2019

AUTOMATED NETWORK SECURITY WITH EXCEPTIONS USING SDN

Sergio A. Rivera Polanco
University of Kentucky, sergiorivera88@gmail.com
Digital Object Identifier: https://doi.org/10.13023/etd.2019.342

AUTOMATED NETWORK SECURITY WITH EXCEPTIONS USING SDN

DISSERTATION

A dissertation submitted in partial fulfillment of
the requirements for the degree of Doctor of
Philosophy in the College of Engineering at the
University of Kentucky

By
Sergio A. Rivera Polanco
Lexington, Kentucky
Co-Directors: Dr. Zongming Fei, Professor of Computer Science
and Dr. James Griffioen, Professor of Computer Science
Lexington, Kentucky
2019

ABSTRACT OF DISSERTATION

AUTOMATED NETWORK SECURITY WITH EXCEPTIONS USING SDN

Campus networks have recently experienced a proliferation of devices ranging from personal use devices (e.g. smartphones, laptops, tablets), to special-purpose network equipment (e.g. firewalls, network address translation boxes, network caches, load balancers, virtual private network servers, and authentication servers), as well as special-purpose systems (badge readers, IP phones, cameras, location trackers, etc.). To establish directives and regulations regarding the ways in which these heterogeneous systems are allowed to interact with each other and the network infrastructure, organizations typically appoint policy writing committees (PWCs) to create acceptable use policy (AUP) documents describing the rules and behavioral guidelines that all campus network interactions must abide by.

While users are the audience for AUP documents produced by an organization's PWC, network administrators are the responsible party enforcing the contents of such policies using low-level CLI instructions and configuration files that are typically difficult to understand and are almost impossible to show that they do, in fact, enforce the AUPs. In other words, mapping the contents of imprecise unstructured sentences into technical configurations is a challenging task that relies on the interpretation and expertise of the network operator carrying out the policy enforcement. Moreover, there are multiple places where policy enforcement can take place. For example, policies governing servers (e.g. web, mail, and file servers) are often encoded into the server's configuration files. However, from a security perspective, conflating policy enforcement with server configuration is a dangerous practice because minor server misconfigurations could open up avenues for security exploits. On the other hand, policies that are enforced in the network tend to rarely change over time and are often based on one-size-fits-all policies that can severely limit the fast-paced dynamics of emerging research workflows found in campus networks.

This dissertation addresses the above problems by leveraging recent advances in Software-Defined Networking (SDN) to support systems that enable novel in-network approaches developed to support an organization's network security policies. Namely, we introduce *PoLanCO*, a human-readable yet technically-precise policy language that serves as a middle-ground between the imprecise statements found in AUPs and the technical low-level mechanisms used to implement them. Real-world examples show that PoLanCO is capable of implementing a wide range of policies found in campus networks. In addition, we also present the concept of *Network Security Caps*, an enforcement layer that separates server/device functionality from policy enforcement. A Network Security Cap intercepts packets coming from, and going to, servers and ensures policy compliance before allowing network devices to process packets using the traditional forwarding mechanisms. Lastly, we propose the *on-demand security exceptions* model to cope with the dynamics of emerging research workflows that are not suited for a one-size-fits-all security approach. In the proposed model, network users and providers establish trust relationships that can be used to temporarily bypass the policy compliance checks applied to general-purpose traffic – typically by network appliances that perform Deep Packet Inspection, thereby creating network bottlenecks. We describe the components of a prototype exception system as well as experiments showing that through short-lived exceptions researchers can realize significant improvements for their special-purpose traffic.

KEYWORDS: Software-Defined Networking, Network Security, Policy Enforcement, Security Exceptions

Students's signature: Sergio A. Rivera Polanco

Date: August 1, 2019

AUTOMATED NETWORK SECURITY WITH EXCEPTIONS USING SDN

By
Sergio A. Rivera Polanco

Co-Director of Dissertation:	Zongming Fei
Co-Director of Dissertation:	James Griffioen
Director of Graduate Studies:	Miroslaw Truszczynski
Date:	August 1, 2019

To my wife Vera, my baby girl Valeria, my parents, and siblings for being my constant support and inspiration throughout these years...

ACKNOWLEDGMENTS

There are several people that I would like to thank for making the completion of this dissertation possible.

I would like to thank my advisor, Dr. Zongming Fei, for giving me the opportunity to pursue my doctoral degree under his guidance. Inviting me as a guest lecturer, providing me with several recommendation letters, and nominating me for student awards made a significant contribution towards my professional career. Next, I would like to thank Dr. James Griffioen for agreeing to serve as a co-chair on my committee. During all these years, Prof. Griffioen has been encouraging and very supportive, constantly providing me with thorough feedback on all of my publications, including this dissertation, and helping me notably improve my presentation skills. I would also like to thank Dr. Jane Hayes, Dr. Hank Dietz, and Dr. Daniela Moga who all had immediately responded to my request to become members of the doctoral committee at a late stage of my research and yet provided very valuable feedback on my work.

I gratefully acknowledge all my colleagues from the different projects in which I was involved for fostering design and implementation discussions, provisioning the adequate infrastructure to deploy system prototypes, and sharing their insights on the systems presented in this dissertation. Especially, members of the Laboratory for Advanced Networking, the Research Computing group, and the Center for Computational Sciences at the University of Kentucky (Mami, Song, Jacob, Charles, Lowell, Bhushan, Pinyi, Satrio, and Nasir), and collaborators of the NetSecOps project from the University of Utah (Prof. Kobus Van der Merwe and Joe Breen). Thanks are also due to the National Science Foundation (under grants ACI-1541380, ACI-1541426, ACI-1642134, and CNS-1346688 subcontracts 1925 and 1928) for providing financial support and funding the research. I am also grateful to the UK Graduate School, especially, Dr. Morris Grubbs, Chad Gilpin, and Ashley Sorrell for organizing dissertation writing camps; the quiet and supportive environment helped me concentrate on my writing.

My personal thanks to my uncle Alvaro and his wife Yolanda who helped me during the application process to the Computer Science program and who kindly opened their home to me during several months of my first year of studies; without their help this would nothave been possible.

I would also like to say a heartfelt thank you to my parents and siblings who always believed in me and supported me through all these years. To my friends, thank you for making this an enjoyable journey.

Finally, words cannot describe how thankful I am to my beloved wife Vera. Ever since I met her, she has been an angel by my side in any success or setback I have faced during my doctoral studies. Whether it was cooking a delicious Ukrainian meal, accompanying me on a trip to a conference, staying awake on a sleepless night writing a paper, or taking me to a salsa dancing event, she made every moment special and helped me keep motivated towards the achievement of my career goals. And to my little Valeria, for being such a peaceful and calm baby since the day she was born, and boosting my motivation to complete this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1. Introduction

Networks provide a wide variety of services needed by a diverse set of environments that include (but are not limited to) enterprise and business operations (e.g. credit card transactions, employee travel bookings, and payroll information access), academic and research workflows (e.g. big data transmission and analysis, virtual and in-class online instruction), healthcare and medical procedures (e.g. patient data access, medical procedures, virtual doctor visits), or housing affairs (e.g. residential network connectivity, and rent payments). Moreover, networks, consisting of routers, switches, Wireless Access Points (WAPs) and middleboxes (e.g. firewalls, load balancers), must interconnect a wide range of devices, including (but not limited to) general-purpose machines like computer desktops and servers; hundreds of personal and corporate mobile devices, for example, phones, laptops, tablets, or smartwatches; appliances that provide monitoring and threat detection; and various kinds of specialized devices deployed at key places over the physical campus that include copiers and printers, badge and biometric readers, intelligent thermostats, motion sensors, IP telephones, surveillance cameras, video-conferencing equipment, or payment terminals, to name a few.

All these devices generate varying types and amounts of network traffic that in one way or another use a part of the underlying network infrastructure. For example, security cameras may generate live streaming data (video/images) to a backup storage system for easy retrieval of images from previous days, payment terminals might send encrypted requests to external credit card franchise systems to check the validity of debit/credit cards, Information Technology (IT) staff could configure printers to only accept printing jobs from members of a particular department or a group of machines, or Intrusion Detection Systems (IDSs) may generate alerts to IT monitoring systems that in turn might send control messages to a number of devices to mitigate malicious or uncommon network activity.

The heterogeneity of campus networks has led organizations [1] to appoint specialized committees that define and establish directives and detailed guidelines regarding the desired and acceptable use of organization's network. In particular, the directives and guidelines are often referred to as *rules* because they describe how devices should operate and communicate between each other (if at all) in terms of who gets to send (or receive) what, when, where and how. A set of rules is usually referred to as *Network Policy* that addresses a wide range of decisions that must be made about the network. For example, network security policies might define the standards for communications between any two systems in the campus network in an attempt to minimize damaging actions like the theft of trade secrets, alteration of data and services, or removal of intellectual property, among others. Even in the absence of malicious actors and cyberattacks, policies are still needed to describe acceptable use behaviors (also known as Acceptable Use Policy (AUP)) and interactions within the network. Network policies also determine how communications within the network should be distributed such that all network devices share the network infrastructure fairly and optimally while aligning with the institution's objectives. Moreover, policies are used to define permissions, prohibited behavior, and the obligations of users and systems based on the organization's hierarchy, personnel expertise/roles, and the services provided.

Policies are of the utmost importance to organizations because they are tightly associated to business objectives, internal processes and workflows, and federal obligations. Unfortunately, technically precise documents describing such policies are either non-existent or scarce at best. Often the only available policy documents were written by Policy Writing Committees (PWCs) in the form of acceptable use and

[1]The terms *organization, enterprise,* and *institution*, will be used interchangeably throughout the dissertation.

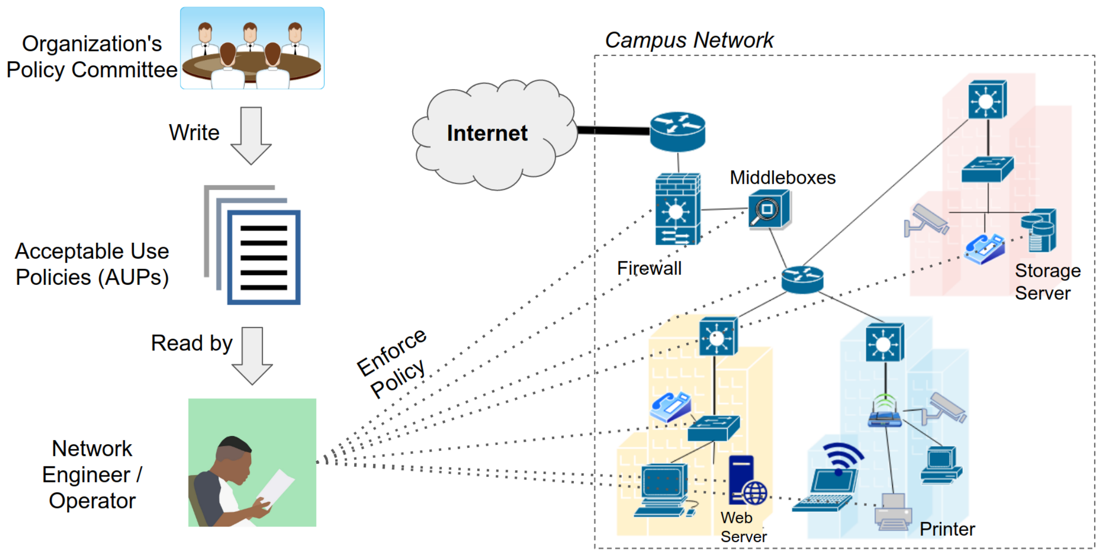


Figure 1.1: A human configuring several devices to enforce multiple Acceptable Use Policies (AUPs)

behavioral guidelines that contain a series of imprecise statements. The statements use a high-level, non-technical language that focuses on organizational procedures and workflows that can be easily understood by network users, board members, and regulatory agencies and fall far short of providing technical information about the actual enforcement of the policy. Hence, mapping vague non-technical statements (understandable by humans) into low-level technical commands and configurations (understandable by machines and network devices) in order to enforce policies is a challenging task. The mapping process requires per-device intervention and heavily relies on the AUP interpretation and expertise of the network operator (or server administrator) in charge of configuring the network system (see Fig. 1.1). Furthermore, the task is error-prone because it involves the management of heterogeneous policy enforcement mechanisms that provide different (possibly conflicting) requirements of the network infrastructure. For example, some mechanisms offer protection of the network border against incoming connections, some allow separation of traffic between groups of users, others authenticate individual users for privileged access to network resources, and still others configure services with custom security. Generally, enforcement mechanisms come with so-called "sane" defaults that do not always align with the organizations' objectives and policies. For example, when users/operators configure, say, a new website or any other service (e.g. mail services, file servers, firewalls, and IDSs), often a default configuration file is provided in case the user/operator just wants the service/appliance to work out-of-the-box. Moreover, by relying on default configuration files, policy enforcement and device/service configuration are conflated in one place (the service/application), potentially leading to policy violations. Furthermore, current enforcement mechanisms are rigid and meant to be in place for large timescales. As a result, policy implementations are rarely dynamic and are usually revised, at best, a couple of times per year. While the revision process may have worked in the past, where all communications could be categorized as general-purpose traffic, institutions now need more dynamic, agile, flexible, and intelligent approaches that can cope with the growing number of complex specialized workflows and activities that take place in a network.

This dissertation describes systems and tools that address the challenges described above: First, it introduces a human-readable **and** technically precise language that addresses the lack of a precise way to document, translate and verify how high-level policies are mapped into their corresponding low-level implementations. Second, it describes an alternative way to process packets in order to enforce network security policies regardless of file misconfigurations or policy violations happening at end systems. Lastly, the dissertation proposes an approach towards network security policy enforcement based on the notion of *on-demand security exceptions*. The exception system allows individual, fine-grained, trusted research flows to temporarily bypass the set of baseline security policies enforced for general-purpose (untrusted) traffic. The results from the experiments show that with a prototype exception system, complex research workflows are able to gain significant performance improvements (more than an order of magnitude) when compared to the regular scrutiny applied to general-purpose traffic in the network.

1.1 Examples and Purpose of Network Policies

Network policies are sets of rules that determine how traffic and communications are treated while they traverse any part of a network. Policies are expressed in a variety of ways that range from high-level human-readable statements–comprehensible by network users and network administrators–to low-level commands and configurations–understandable and executed by network devices and end-systems that offer services to network users. Policies have a profound impact in the interactions that take place within a network because they arbitrate aspects such as network resources that may be involved (e.g. web servers, databases, file systems) in a particular type of connection (e.g. incoming, outgoing); when, how often and for how long communications are allowed to happen (e.g. 30 minutes during business hours a maximum of 3 times per week); which sets of users may be part of the communication (e.g. graduate students of the biology department); who can initiate the communication (e.g. only traffic originating from within the campus network); or what processing/conversion should be applied to the communication (e.g. encrypted communication using secure protocols).

Enforcing policy decisions in the network requires network-level mechanisms that actually implement and carry-out the network policies. Although PWCs define network policies using (high-level) non-technical statements, organizations (through their IT staff) rely on enforcement mechanisms that are commonly deployed as part of the network infrastructure (i.e. they are low-level). After all, communications ultimately boil down to network packets and, in the majority of cases, the information contained inside the packets suffices to apply policy. Since packets traverse switches and routers, network operators have historically used the mechanisms embedded in these network devices to enforce policy. Furthermore, network vendors have also developed dedicated network appliances that are optimized to enforce policies that require more complex functionality than the lightweight packet inspection switches and routers do. The network appliances, also called *middleboxes* [1], rigorously scrutinize network traffic flows and apply actions like blocking or allowing traffic, rate-limiting network connections that consume disproportionate amount of bandwidth, or logging suspicious activity, ensuring that all analyzed traffic is policy compliant.

Over the years, low-level enforcement mechanisms that are based on switches, routers and specialized hardware have helped institutions deploy a wide range of network policies. In the following, we list example network policies that use low-level mechanisms to ensure policy compliance and are typically found in university campus networks.

- *All traffic coming into the campus must go through an Intrusion Detection System* (IDS). IDSs are network appliances that inspect network traffic and generate warnings/alerts if suspicious or anomalous activity is detected. An IDS that inspects all incoming traffic reduces the chances of compromising local network assets from outside (untrusted) sources. In case of detection, an IDS helps network administrators isolate compromised equipment in a timely manner and apply appropriate controls to prevent similar events from occurring again.

- *Web traffic should only reach registered primary (centrally-managed) and secondary (managed by units) web servers* [2]. Web services are one of the principal avenues for cyberattacks from external sources because they are “open-doors” into the local network from the Internet. Network operators should implement access control policies that prevent web traffic to reach unregistered servers because oftentimes these servers are not properly maintained (e.g. running insecure services, or lacking the latest security updates) and increase the the chances of security exploits in the network.

- *In order for a vendor to gain access to a server from off campus, they must be assigned a Virtual Private Network (VPN) account, and use the university's VPN service* [3]. At times, university's IT department cannot fully manage specialized equipment in the network. For example, in High-Performance Computing (HPC) environments, vendors might offer remote administration and management services of the high-speed, low-latency, and complex Infiniband network [4] due to a lack of trained staff in the university. VPNs establish secure (and private) channels that protect university information traversing shared/public networks (like the Internet) to reach authorized third-parties.

- *Printing jobs in the computer science department may only be submitted by members (faculty, staff, students) of the department.* Network operators need to limit access to network printers in order to preserve the estimated lifetime of printers and the goods needed for the equipment to work properly (e.g. cartridges, paper, memory, etc).

- *Payroll systems should only be accessed by members of the human resources department and designated payroll officials from units across campus.* Payroll systems contain sensitive information about employees like IDs, phone numbers, addresses, or financial information (e.g. salaries, account numbers). Ensuring that access to such information is only granted to an authorized group of individuals is vital to prevent harmful events such as identity theft and fraud.

- *Traffic containing medical records traversing the campus must be encrypted.* Some network policies derive from rules found in federal regulations (HIPAA[2] in this case) that must be fully abided by in order to avoid penalties and sanctions.

[2]Health Insurance Portability and Accountability Act

- *No individual service or system running on the wired/wireless network should send or receive more than 10 GB of data per day* [5]. Network utilization policies ensure that network infrastructure resources (e.g. links, servers, etc) are fairly used among network users. Limiting the amount of traffic a user may send/receive per day prevents network systems from being overloaded and the services offered (e.g. web, mail, or file sharing applications) from being disrupted.

1.2 How Network Policy Implementation is Done Today

In the same way that defining and revising network policies in an organization are important practices to ensure that only legitimate operations occur in the network, so are the mechanisms used to implement the policies.

When it comes to network security, one of the first objectives an organization defines when setting up a network is to protect its local users and network assets (e.g. services, databases, source code repositories, etc) from external malicious actors. Network vendors have developed varying tiers (in terms of cost, features, performance) of network firewalls that serve as the first line of defense against network attacks coming from the Internet. Firewalls are strategically placed at the edge of the network with the goal that outgoing (trusted) traffic is protected and all incoming (untrusted) traffic is inspected. Firewalls make decisions on whether to allow or drop packets based on a set of security rules that minimize the risk of cyberattacks from external sources. Unfortunately, just guarding the borders of the network from external malicious actors is not enough to protect a network today. With the massive adoption of personal (network-capable) devices, the chances for an entity within the organization to cause a security violation drastically increase. For example, a personal laptop that has been compromised before joining the enterprise network (e.g. by unknowingly downloading malicious software) may trigger unnoticeable attacks on local systems once it has been assigned a private campus IP address (e.g. via WiFi). In this case, the laptop completely invalidates the protection offered by border firewalls. Deploying multiple firewalls at various network locations could certainly decrease the chances of internal attacks from succeeding. However, the approach is overly expensive, inefficient, and non-scalable in the long term.

To complement the network edge protection offered by firewalls, network operators started to leverage built-in features and protocols found on traditional network equipment as cheaper alternative approaches to enforce policies within the internal network. One prominent example is the modified use of Virtual Local Area Networks (VLANs). VLANs were originally designed to group hosts in a network regardless of their physical location. Today, VLANs enforce security and privacy policies (e.g. separating traffic between groups of users like professors and students), provide simplified access control between regions of a network (e.g. aggregating groups of addresses into a unique VLAN number for a more concise notation), and enable decentralized network management (e.g. delegating management of a VLAN to each department's IT staff) [6]. Despite being widely deployed in campus networks as enforcement mechanisms, VLANs impose limitations in policy enforcement because they were not designed with network security in mind. For example, the number of possible VLANs one can configure per switch is limited (1-4096); potentially running into scalability issues for organizations with large number of groups of users. Moreover, VLANs do not need to have the same meaning across multiple switches, e.g., VLAN 10 in one department may refer to student traffic while in another department that same VLAN number may be associated with faculty traffic—the lack of a unified source of truth with campus-wide information complicates policy enforcement based on VLANs. As a result, most campuses have defined cross-campus standards for VLAN numbers to ensure VLAN numbers have the same meaning and use everywhere on campus. Lastly, the VLAN information included in network packets can only be used as a lightweight form of identification (like IP or MAC addresses) but do not provide any form of authentication that can prevent spoofing practices.

Firewalls and VLANs examine packet headers to enforce policy. However, as noted above, additional mechanisms are needed to implement policies at the user level. Even though IP and MAC addresses sufficed to identify a network user in the past, today's networks are far more complex and use advanced techniques (e.g. virtualization) to share equipment across multiple users and applications. Consequently, policies must be also applied to whoever is logged into a system and is generating (or receiving) the network traffic; not to mention the complexity when multiple users are logged in and performing distinct networking activities simultaneously. As a result, network devices or appliances often include mechanisms that allow authentication and facilitate dynamic mapping of *users* in the network to low-level identifiers like IP addresses. One such mechanism is the Remote Authentication Dial-In User Service (RADIUS) [7]. In RADIUS, the server acts as a central authentication place for remote connection requests received at other network systems. In a simple deployment, users provide authentication credentials to the network system they want to access (i.e., the RADIUS client). Then, the RADIUS client hands over the user credentials to the RADIUS server. The RADIUS server internally searches the policy that must be enforced on the user (e.g. grant full access, grant limited access, forbid access) and responds back to the RADIUS client with the information found.

Lastly, network operators and system administrators modify the configuration files of the services and applications running on network servers as well as end system utilities provided by the Operating System (OS) in order to implement network policies for servers. Policies such as disabling unnecessary services from a system hosting a database server, rejecting incoming requests from connections that use clear text protocols, or only allowing access to a specialized end system from specific IP address ranges, are example policies that oftentimes are implemented in configuration files.

Configuration files are a convenient place to enforce policy because they are modular (i.e. there is a configuration file per application running on the system), the language and syntax used to modify values and enable (or disable) features is relatively easy to understand (typically in the form of key-value pairs), and configuration files almost never have to be written from scratch, instead, applications oftentimes come with pre-populated (and well-documented) “sane” defaults that suffice to bring a service up in the network and allow for easy modification.

1.3 Problems With Network Policies

Having described the need for network policies and given examples of policies used in campus networks, we also need to mention some of the challenges of realizing policy in today’s networks.

Lack of precise policy definitions and documentation: The first step in the process of realizing network policies is to define them. Defining policies often involves time-consuming and recurring (yet necessary) meetings and discussions where PWCs of organizations revise objectives and procedures in order to establish what constitutes an acceptable use of the network by network users. Ideally, the outcome of these discussions should result in a set of policy documents written at three levels of abstraction. Namely, (1) at a high-level informing users about how they should use the network infrastructure and services, (2) at a middle-level human-readable yet technically-precise manner that tells IT personnel what the policy means in terms of network configuration (e.g. network ranges, groups of systems, network traffic), and (3) at a low-level that can be used by policy enforcement mechanisms that ultimately will implement the policy.

Today, policy definitions exist at the higher level. Looking at several universities' network policies online [8, 9, 10, 11, 12], one can find statements and documents indicating the permissible, mandatory, and prohibited actions that users may and may not do if they use the campus network infrastructure. From a technical point of view, high-level policies are vague and imprecise due to their target audience (users, board members, regulatory agencies) and hide details about the underlying equipment implementing the policies (i.e., the low-level statements).

Unlike high-level policies, middle-level statements that should be targeted at IT staff are non-existent or scarce at best. Even worse, instead of carefully deriving the middle-level information from the high-level policies, network operators are on their own (using their own intuition) when it comes to deciding how to map vague high-level statements onto the network. The problem with this approach is that the lack of technicality in high-level statements constrains the implementation of the policy to the interpretation and expertise of the network/server administrator—policies become hard to accurately translate, deploy/implement and verify. Even a perfect configuration where multiple policies are manually added and sanity checked, the distributed nature of some policy enforcement mechanisms (e.g. recall VLANs above) and the technical details found while configuring network devices complicate tasks associated with network policy maintenance. Mapping high-level statements straight into low-level configurations hinders the possibility to verify and ensure that a policy is active over a period of time. Moreover, because of the dynamic nature of campus networks, nothing prevents other members of the IT staff to issue new configurations to support an emerging service while unknowingly violating/overriding a policy that was already in place.

Dangerous impact of server configuration in policy enforcement: Various high-level statements found in university network policy documents (like AUPs) are expressed in terms of the services campus infrastructures offer and the systems that host those services (e.g. web, mail, file, or video-conferencing servers). For the most part, hardware and software powering services is highly configurable, flexible, and capable of enforcing various types of policies via simple modifications to configuration files. Albeit possible, there are adversities of using server configuration as the **only** mechanism to implement policy. Policy enforcement requires careful box-by-box configuration that does not scale well (e.g. manual intervention for hundreds or thousands of systems in a campus network), is error-prone (e.g. potential for typos in configuration files), and conflates device/service configuration with policy enforcement. For example, system administrators conflate functionality and policy enforcement when the network policy requires a web server to be contacted over secure a connection (e.g. via HTTPS and not HTTP), when administrators set up a blacklist to certain nodes to forbid access from specific outside entities (e.g. pairing Secure Shell (SSH) with `fail2ban` [13]), or when administrators limit access of a system to a predefined set of networks, IP ranges, network users, or hosts in the local network. In all three cases, the policy is implemented as part of configuration files overlooking the fact that network equipment (i.e. routers/switches) could enforce the desired behavior. In addition, it is challenging to enforce policy at end systems because users, that now have the ability to join internal networks with their personal equipment, might be system administrators themselves. While a group of conscious users might responsibly follow security guidelines published by the organization, the chances are high that most users oftentimes end up relying on “sane” defaults that come with the applications they use.

Although default configuration files allow for fast deployments (i.e. services working out-of-the-box), the default functionality found on the files depend on decisions made, and best practices defined, by the developers of the software as opposed to organization AUPs.

The need for on-demand policy exceptions when the enforcement mechanism negatively impacts legitimate workflows: Network policies are typically written and defined around the notion of general guidelines governing all network traffic. However, in highly dynamic environments like campus networks, one-size-fits-all policies are restrictive and impose limitations to some specialized workflows that actually align with the organization's goals and objectives. In order to circumvent these limitations, a limited number of organizations have established a procedure to add exceptions to general-purpose policies. Unfortunately, a network user in need of an exception has to perform a series of time-consuming human-dependent steps such as filling out long forms justifying the exception request, collecting signatures from high-ranked employees (e.g. Chief Information Officer (CIO), Chief Technology Officer (CTO)), scheduling meetings with IT staff and PWCs, mailing hard copies of forms to multiple offices, to name a few. In addition, the (static) policy exceptions a user can requested are intended for static workflows that are expected to remain in place for long periods of times (on the order of months). However, exceptions can be given a more important role and can be used as a tool to provide context that could allow user trusted workflows to temporarily bypass the general policies defined by network providers. Recall the first sample policy we presented in Section 1.1: *"all traffic coming into the campus must go through an IDS"*. While IDSs offer security features to network traffic, their internal packet processing mechanisms and intensive analysis oftentimes result in adverse effects to common research-oriented operations. For example, science workflows often require short-lived, high-throughput transmissions when working with large datasets (e.g. to share data sets with collaborators at other institutions). Enforcing the aforementioned security policy via heavy IDS inspection poses a major performance bottleneck to big data transfers and ignores the fact that network users can share characteristics of the network traffic they will send before the transfer starts.

With information about the transfers, individual workflows could be verified and marked as trusted. After trust establishment, a valid exception for a data transfer workflow would be to only allow the specific data transfer to bypass the deep scrutiny of the IDS while the remaining general-purpose traffic would still be subject to the general inspection policy, even if it comes from the same end-host.

1.4 Dissertation Contributions

Part of the complexity involved in defining technically precise policies, conflating server configuration and policy enforcement, and allowing trusted exceptions, lies in the distributed nature of network equipment used to perform these tasks. It is hard to describe and specify global network-wide policies when the entities and protocols used to enforce them behave in distributed and independent ways. Even in reduced environments (e.g. per-department policies), unawareness of the regulations that are enforced elsewhere in the network could lead to unexpected violations to security policies or undesired network performance for regular operations.

As noted above, the problems operators face with respect to network security policies are: (1) Policy Writing Committees write network policy documents using vague, non-technical, imprecise statements that are challenging to implement in the network; (2) network operators configure network devices to implement policies largely relying on personal interpretation of the policy documents and personal expertise and skill set; (3) policies that are targeted at servers are not enforced in the network but usually implemented in configuration files of the server, thereby conflating policy enforcement with service functionality; and (4) traditional one-size-fits-all policies introduce performance and behavior issues to certain flows that do not need enforcement.

In this dissertation, we will address these problems and provide the following contributions.

Human-readable translatable policy language: We introduce a human-readable yet technically precise policy definition language (called *PoLanCO*) for network operators to write network security policies and request exceptions based on information and details of the network gathered from multiple sources and protocols (e.g. types of traffic, static files, RADIUS server authentications, Simple Network Management Protocol (SNMP) data, OpenFlow). The network information includes specifics about the equipment that is used, the built-in functionalities each network device has that can be used to enforce policies, or the paths available to traverse the network. The proposed language serves as a middle ground that fills the existing gap when translating vague and imprecise high-level statements (typically tailored to network users) into low-level configurations and commands of network equipment (used to enforce policies). We show that with the human-readable language it is possible to describe and document a set of policies describing the permitted, prohibited and mandated behaviors users must abide by in several environments of a campus network. Particularly, we show examples of how policy excerpts publicly found on a large number of university websites can be written as simplified human-readable statements using PoLanCO. Moreover, we also show that operators may use PoLanCO to write policy statements that are not found in AUPs that enforce policies regarding the interactions of components within specific environments (e.g. printers in a department, the edge of the network, emerging HPC systems, etc.).

Separation of policy enforcement and server configuration: We propose a network policy enforcement layer called *Network Security Caps* that leverages OpenFlow packet processing to separate the enforcement of policies from network server configurations. Unlike related work in the field, the approach does not require a clean-slate network, ad-hoc modification of middleboxes to enforce policy, or expensive network appliances to be deployed – significantly reducing the Capital Expenditures (CapEx)/Operational Expenditures (OpEx) when compared to a full overhaul of the network infrastructure. Moreover, the enforcement layer can be extended to add customized services that require decision making on a per-flow basis and are not necessarily tailored towards only security. We reduce the potential danger to cause a security policy violation by implementing policies not only at the end-system level but also in the network equipment in charge of forwarding traffic. With this approach, we do not abolish policy enforcement at end-systems, but rather ensure that there are multiple layers of security controls placed throughout the network system to minimize exploits. Under *Network Security Caps*, if a server is misconfigured, the network will still ensure that actual policy is enforced to the traffic sent to/from the vulnerable server.

Short-term on-demand security exceptions: We present a new approach towards network security based on the notion of *short-term on-demand security exceptions*. We developed an exception system where network providers and network users can establish trust relationships in order to temporarily deploy fine-grained policy exceptions that can be associated with individual workflows. The exception system prototype requires users (or their applications) to authenticate using university credentials and provide information about the characteristics of their flows (e.g. target destinations, types of traffic, project numbers, department affiliation). The flow information provides context to the decision making process of our automated system in order to deploy (or reject) an exception. Researchers are able to achieve performance improvements of more than an order of magnitude for some of their research flows while letting general-purpose traffic be regulated by traditional network appliances used to enforce baseline security policies on the regular campus network.

Although the work in this dissertation applies to any type of Autonomous System (AS), the focus of this work is on campus networks. We developed the systems presented in this dissertation under the assumption that a campus network is centrally controlled and managed by an IT group. With current advancements on intent-based systems, campus networks have started to move to solutions that use a centralized controller allowing us to apply (distributed) policy from a central IT system.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows:

- Chapter 2 presents background and existing work related to the applications, technologies and systems presented in this dissertation,
- Chapter 3 proposes a network policy language that is human-readable yet technically precise that translates network policies into low-level configurations,
- Chapter 4 describes the *Network Security Caps* enforcement layer that protects end systems by enforcing policy in spite of misconfigurations,
- Chapter 5 describes our approach towards network security that uses trust relationships to promote exceptions as first-class entities for policy enforcement,
- Lastly, Chapter 6 discusses future directions, further research opportunities, and a summary of the work presented in this dissertation.

Chapter 2. Background and Related Work

In this chapter, we describe the state-of-the-art of services and approaches that relate to the design, concepts, applications, and prototypes described in the subsequent chapters of the thesis. First, we give an overview on how the traditional mechanisms used to enforce security in computer networks. Then, we describe the challenges that network operators face when trying to enforce security policies in networks. Next, we review historical efforts that address those challenges that lead to the emergence of the Software-Defined Networking architecture. Lastly, we summarize existing applications, frameworks and efforts addressing (part) of the problems we introduced in Chapter 1. Specifically, we refer to approaches used to secure networks in Software-Defined Networking (SDN) settings, and we report on network programming languages built for SDN that are intended for network operators to develop programs – as opposed to Command-Line Interface (CLI) – that map high-level policies to low-level configurations for policy enforcement.

2.1 Traditional Network Security

Networks consist of a set of distributed switches and routers where packets are forwarded on a hop-by-hop basis. Network devices make independent forwarding decisions based on network information gathered via several distributed protocols such as the Link-Layer Discovery Protocol (LLDP) [14] that advertises device identity and capabilities to adjacent peers, the Spanning-Tree Protocol (STP) [15] that helps preventing packets from looping in a single domain, the Routing Information Protocol (RIP) [16] that exchanges routing tables between neighbors, the Open Shortest Path First (OSPF) [17] and Intermediate System to Intermediate System (IS-IS) protocols that share the state of router interfaces to other routers, or the Border

Gateway Protocol (BGP) [18] that exchanges paths and reachability among edge routers of ASs.

Historically, network operators have manually configured networks by logging into each node and issuing vendor-specific CLI utilities to configure the nodes in hopes of a implementing a consistent policy across the network. More recently, because the number of services, systems and devices in a network have increased exponentially over the last decade, there has been a push from network vendors to offer centralized controllers that network administrators can use to configure the network in a single place. Examples of these controllers include SDN controllers (see Section 2.3.1), Intent-based network controllers (e.g. Cisco Digital Network Architecture [19]), or Wireless Local Area Network (LAN) Controllers (e.g. Aruba Controllers [20]).

Albeit these centralized solutions provide network configuration consistency, they do not solve the problems we highlighted in Chapter 1 and the demands to protect the infrastructure from increasingly complex network attacks, provide optimal performance for bandwidth/latency sensitive applications, and make efficient use of the network resources. Moreover, the lightweight control and management features embedded in general-purpose network equipment cannot efficiently enforce policies that require heavy processing of network packets. Network vendors started to develop dedicated intermediary appliances (so called middleboxes) that provide more complex services and heavy processing than what is commodity routers/switches. Middleboxes shortly became have become the go-to mechanisms where network operators specify and enforce organizational policies based on sets of rules and event descriptions. Moreover, some of the appliances also contribute to the security of the network. With the increasing number of Bring Your Own Device (BYOD) systems joining networks, more avenues for security threats are opened. It is not uncommon to find these specialized devices at various key locations in the network intercepting traffic, replicating it for further offline monitoring, or performing any local action based on the packet payload contents.

Network security has become a major priority in today's networks given the significant effects (including financial losses) a security breach – such as loss of confidential or personal data, unavailability of a service, theft of intellectual property or harm to network users – may cause to an organization.

In a distributed computer system such as a network, security involves the protection of the resources that make up the infrastructure including the transmission medium (e.g. channels and connectors); network equipment like access points, switches, or routers; end-systems such as servers, mobile devices, IP telephones, desktops, Internet of Things (IoT) devices, etc. as well as the files and information stored in them. Over the years, security experts from both industry and academia have developed multiple security enforcement technologies and defense mechanisms in the form of network appliances and dedicated systems to ensure that network infrastructure elements are secured from various attacks. Below, we describe some of the most common security approaches found in traditional networks (also referred to as *legacy networks*).

2.1.1 Firewalls

Firewalls are arguably the most well-established technology for protecting networks from unauthorized access. They are typically the first line of defense to the outside world (i.e. the Internet) inspecting all incoming and outgoing traffic to and from the local private network (Fig. 2.1). The are several implementations of firewalls in the market ranging from in both software (e.g. iptables [21], pfSense [22], PF [23], Windows Firewall [24]) and hardware (e.g. Palo Alto [25], Dell SonicWALL [26], Fortinet FortiGate [27]). Firewall solutions filter traffic and label it as legitimate or malicious based on information found in network packets and a set of (prioritized) rules that represent the policy of an organization. Firewall policies are typically consolidated in two forms:

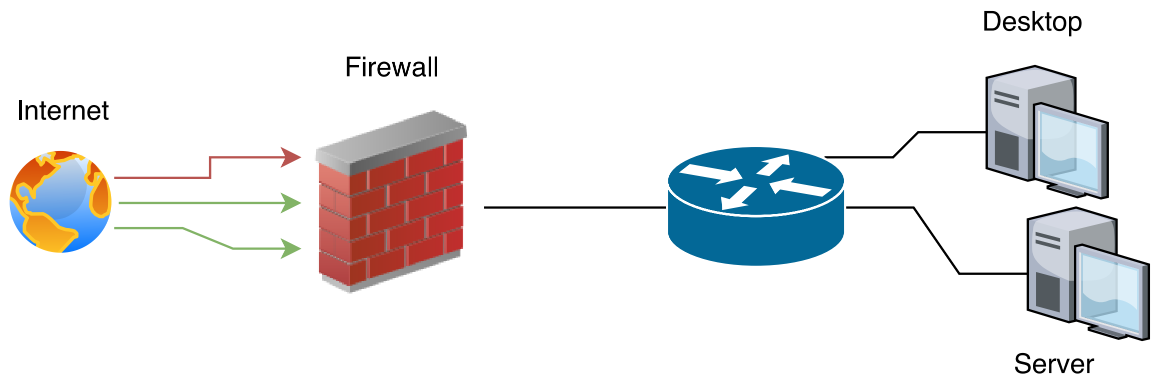


Figure 2.1: Firewall as a perimeter defense of a network

- *Deny-everything-not-specifically-allowed*: Also known as whitelisting. Under this setup all traffic is denied by default with the exception of a few allowable connections. This model is highly conservative, severely limiting access to the internal network from the outside.

- *Allow-everything-not-specifically-denied*: Also known as blacklisting. A more permissive model that flags bad actors, placing them on a "forbidden" list and letting all other communications go through.

In addition to the allow/deny policies, current firewall solutions also include other services such as alert generation, system logging, network address translation, connection tracking, and proxy functionality. Although firewalls are normally associated with perimeter defense, they can also be found at other locations in the internal network. For example, in front of dedicated public servers, data centers, or even running on critical end systems given that almost every OS provides an easy-to-use firewall utility.

2.1.2 Intrusion Detection and Prevention Systems

Intrusion Detection and Prevention Systems (IDPS) are software or hardware systems that monitor the events occurring in a computer system or network. IDPSs analyze packet captures and events to identify signs of security problems [28] such as suspicious activities (e.g. using BRO [29]) or by matching network packets against well-known patterns and signatures (e.g. using SNORT [30] or Suricata [31]).

IDPSs can be described in terms of three main components:

- *Information Sources:* The information used to determine whether an intrusion took place or not. For example, data collected while monitoring the network, individual hosts, or application processes.
- *Analysis:* IDPSs dissect and organize the events obtained from the information sources and determine when a particular sequence of events relate to ongoing intrusions or compromises that already happened.
- *Response:* The actions that should be executed whenever an intrusion is detected. The set of actions include passive measures such as logging, reporting the incident to the network operator for further action, or some active operations such as dropping a connection, or redirecting it to a honeypot.

While IDPSs were historically present in the systems they protected (called Host-based Intrusion Detection Systems), network operators can achieve a significant cost reduction by deploying these types of enforcement mechanisms as standalone devices that can monitor targeted portions of the network (Network-based Intrusion Detection Systems). Today, most of IDPSs are network-based that perform local analysis of the traffic and report detected anomalies to a central management console.

IDPSs can be deployed at various locations (Fig. 2.2). For example, an IDPS can be passively attached to a switch port that mirrors traffic coming from other interfaces for offline monitoring, actively deployed on the network path such that packets are analyzed before they reach their destination, or activated on critical systems that require enhanced security and are critical to the business operations (e.g. web site, file server, mail server, etc.).

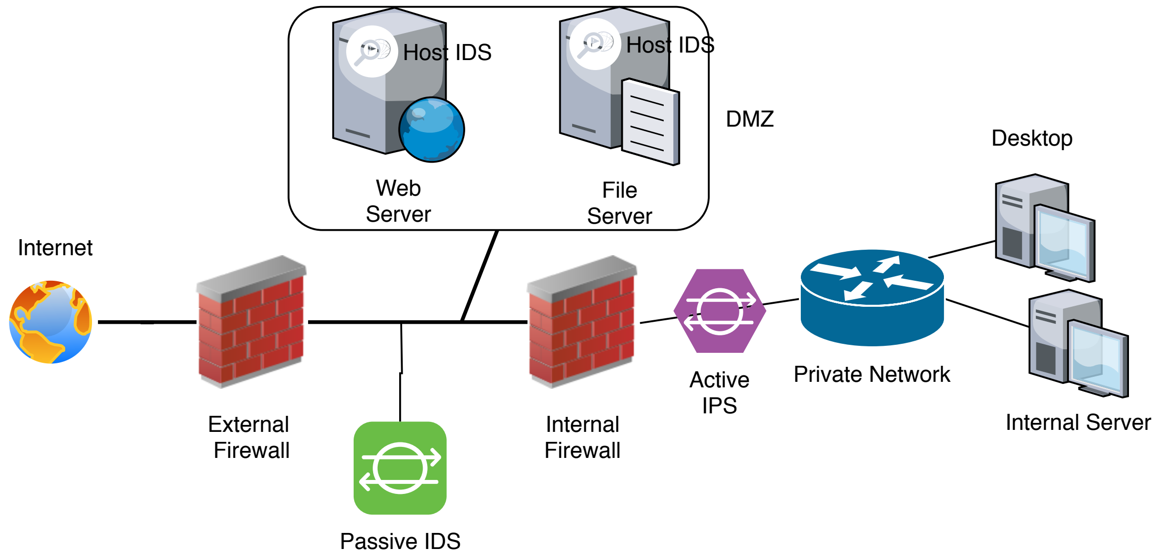


Figure 2.2: Deployment of IDSs and IPSs

2.1.3 Virtual Private Networks

A Virtual Private Network provides a mechanism to protect data that is being transmitted over the Internet (insecure) to a private network (assumed to be secure). VPNs allow users to remotely (e.g. from home, say during a trip) access systems and resources that are not accessible from the Internet because the campus firewall (Section 2.1.1) normally blocks access to those resources from external locations.

In order to access resources via VPN, network users must install an authorized client on their local machine to authenticate to a Remote Access Server (RAS) that is normally located at the edge of the campus network. The RAS contacts a directory service like a RADIUS or Lightweight Directory Access Protocol (LDAP) server to verify that the user is allowed to use the VPN service. The RAS then creates an encrypted tunnel over the Internet connecting the user to the campus edge so that the user's computer appears to be part of the campus network.

Once a network user is authenticated, every individual packet the user sends is encapsulated in a new packet with new header information. The new packet provides routing information so it can traverse the public network before reaching the tunnel endpoint. Upon tunnel endpoint arrival, the packet is unwrapped and traverses the internal network until it reaches the desired resource. Tunnels are established using technologies such as the IP Security (IPSec) [32] protocol, the Point-to-Point Tunneling Protocol (PPTP) [33], the Layer 2 Tunneling Protocol (L2TP) [34] or the Secure Sockets Layer (SSL)/Transport Layer Security (TLS) protocol.

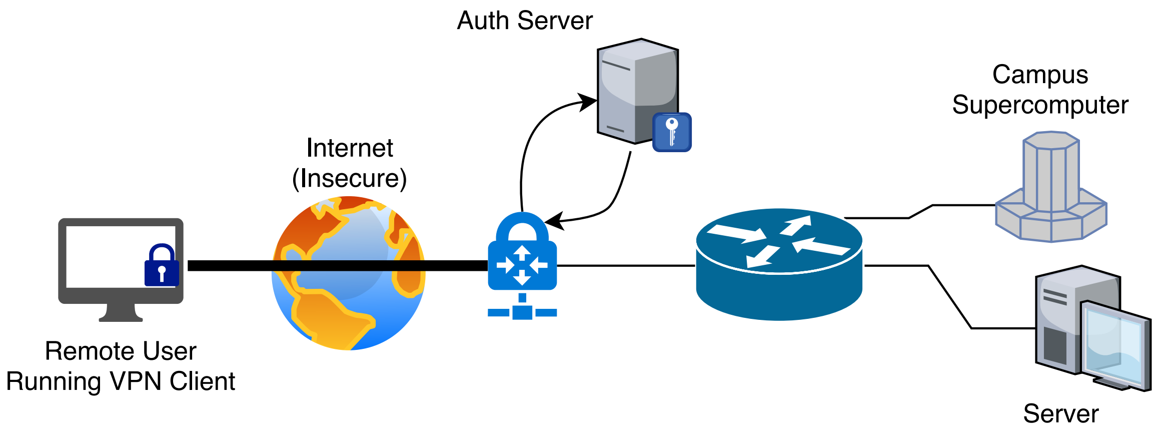


Figure 2.3: Using a VPN to access campus resources

2.1.4 Honeypots

A honeypot is a collection of servers or systems that is typically separated from the production network and whose purpose is to lure attackers into interacting with them (i.e. act as traps or baiting systems). The goal of a honeypot is to gather, log, and analyze the operations and tasks executed by an attacker in order to learn her *modus operandi* and how to best protect the actual production resources from potential vulnerabilities.

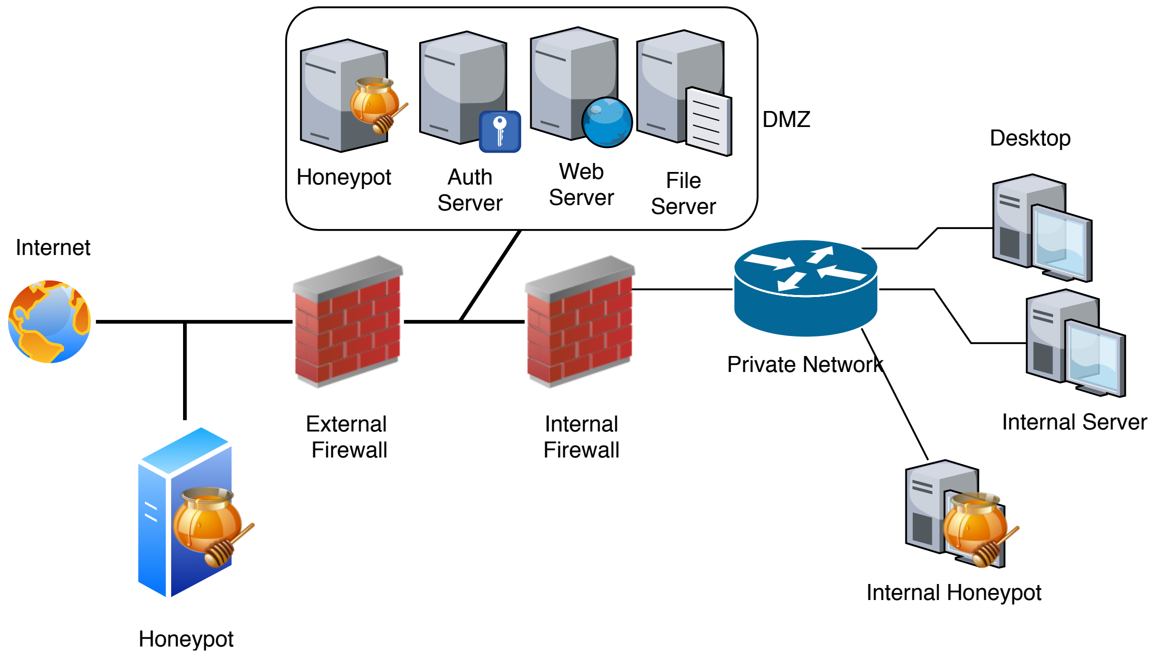


Figure 2.4: Multiple locations to deploy a honeypot

Honeypots do not enforce network security policies *per se*. However, they are a frequently used security analysis tool that help network security policy writers with the definition of newer controls for emerging attack trends. Since the Domain-Specific Language we propose in Chapter 3 allows network operators to write statements that send traffic to honeypots, we describe below the locations in the campus network where honeypots are found.

Honeypots [35] can deployed at possibly strategic places throughout the campus network (Fig. 2.4). A honeypot deployed at the edge, outside the campus external firewall, can attract a great number of external attackers since no pre-filtering is done. However, honeypots at the edge do not monitor attacks coming from the inside of the network. Another place where honeypots can be deployed is alongside Internet-facing servers (e.g. web server, or file server) that are typically located in a Demilitarized Zone (DMZ)—A “service” network located between the campus private network (secure) and the Internet (insecure) where servers are carefully tuned and locked down to receive access from the general public. Because of their exposure, a honeypot at this location could help network operators learn from potential insider and external actors trying to access public facing servers. However, because DMZs typically sit between two firewalls (i.e. an external firewall and an internal firewall), they only receive traffic that either firewall let through and therefore do not see all possible malicious connections. Lastly, honeypots can also be deployed inside the private network to log and analyze behaviors and insider attacks. The major difficulty of a honeypot is its initial setup as it still needs to be a controlled environment that has to resemble a real network yet should not be used to attack production systems.

2.2 Limitations of Traditional Networks

In recent years, campus networks, enterprise networks, and data center networks have grown in size, been used to carry a wide range of traffic, and become increasingly complex, expensive and hard to manage [36]. Below we outline some of the challenges network operators face when managing and trying to secure the growing number of routers and switches that comprise today's traditional networks:

Extensive Protocol Support: Router/Switch vendors often add support for a wide range of protocols. While a large feature set may look appealing, it complicates network management because some protocols can cause conflict with each other or with features enabled other equipment thereby introducing unexpected behaviors for network users.

Cross-Vendor Incompatibility: The incompatibility across multi-vendor equipment leads to unusual workarounds to manage and troubleshoot the network. The use of non-standard proprietary protocols makes it difficult for an operator to understand unexpected traffic behavior (e.g. packets being dropped, low speeds over data transfers) reported by network users.

Middlebox Processing: The extra layer of complexity added by middleboxes; albeit being pervasive and critical in medium and large networks, from a network management point of view, they are significantly more expensive than general-purpose equipment. These devices increase the CapEx/OpEx of organizations, introduce a new set of vendor-dependent CLI commands and management dashboards that oftentimes require extensive training or outsourcing management, add to existing box-by-box configuration workflow, and, like any proprietary solution, packet processing is susceptible to bugs and misbehavior because unlike general-purpose equipment, these middleboxes are typically blackboxes that only the vendor can fix.

2.3 Software-Defined Networking

The problems outlined above are not new or even recent, dating back to the early-mid 1990s [37]. Since that time, the goal to enable programmable networks that simplify network management and lower the barrier to deploy, new, more efficient, more secure, services in the network has been an active research area. Caesar *et al.* proposed a logically centralized Routing Control Platform (RCP) [38] that separated the IP forwarding plane from the process of route selection and BGP route advertisement for every router within an AS. Their work showed that RCP could emulate a full-mesh Internal Border Gateway Protocol (IBGP) configuration while substantially reducing the overhead on the routers. Greenberg *et al.* proposed 4D [39], a clean slate architecture that generalized concepts introduced in RCP to achieve network-level objectives (like efficient BGP route advertisement) instead of individual router configurations. 4D had four components, namely: (1) a *Decision plane* in charge of making all decisions driving the network control based on a realtime view of the network topology; (2) a *Dissemination plane* that provided a communication channel to install packet-processing rules (derived from decisions made) into the network switches/routers; (3)

a *Discovery plane* responsible of collecting topology and traffic information in the network from neighbors and physical components; and (4) the *Data plane* whose main function is to process individual packets based on state pushed by the decision plane. While 4D was designed as a clean slate environment, the work of Casado *et al.* with the Ethane [40] architecture demonstrated a concrete deployment of programmable network infrastructure on Stanford's University campus network. The deployment proved that previous efforts to separate the decision making process from packet forwarding would indeed not only simplify network management, but cause a reduction of expenses due to the use of commodity hardware. Moreover, Ethane had the potential to develop innovative workflows in the network based on fine-grained policies, enable network virtualization, and serve as the basis to raise abstractions of network concepts/elements that could yield the development of network programming languages. SDN could be considered one of the most effective demonstrations towards enabling programmability in traditional IP networks [37], particularly with examples of publicly successful deployments like Google's data center network [41], the Global Environment for Network Innovations (GENI) [42] a federated virtual testbed for research experiments, AT&T's take on how to integrate SDN with traditional technologies to enable an SDN Internet Service Provider (ISP) backbone [43], or Facebook's SDN solution to efficiently deliver content to its users [44].

In addition, higher-level abstractions for networks such as network programming languages (Section 2.4), network operating systems, security applications, new paradigms like Network Function Virtualization (NFV) and cloud computing show the rapid evolution of SDN over the last decade. Both industry and academia have been involved in a myriad of projects tackling on-going deployment and research problems under this architecture (e.g. self-driving networks [45, 46], Internet eXchange Points [47, 48], security attack mitigation [49, 50], portability of applications [51, 52], policy and intent mapping [53, 54], etc.). SDN is still an emerging architecture and due to the well-known rigidness of IP (and its narrow waist) and the cost associated in the replacement of existing infrastructure, SDN has the potential to replace the distributed architecture on which the Internet was built. Nevertheless, we consider SDN can coexist with legacy technologies and more importantly, gives the research community an opportunity to explore alternative approaches towards problems in legacy networks.

2.3.1 Architecture

SDN separates the control plane from the data plane into a logically centralized entity (Fig. 2.5). By decoupling the data and control planes, network devices become simple forwarding elements that carry out the decisions made by the centralized entity (called the *Network Operating System (NOS)* or *SDN controller*).

The NOS uses a well-defined interface called the *Southbound Interface* to modify the forwarding state of network devices and determine what actions should be applied to specific network flows via control messages. Internal modules in the NOS generate the messages that make network decisions based on information such as network load, status of the network topology, network policies, or requests from network applications built on top of the NOS.

The ability to develop network applications (shown in the management plane of Fig. 2.5) is what makes SDN networks programmable. The NOS abstracts the complexities found in the forwarding plane (e.g. diversity of vendors, supported protocols) and defines Application Programming Interfaces (APIs) (the *Northbound Interface*) that network applications can use to provide specialized services (e.g. traffic engineering, Quality-of-Service (QoS), security, and routing) similar to the features offered by traditional routers.

In SDN, every plane has its own specific functions, can evolve independently, and communicates with other layers via APIs that are described below. This separa-

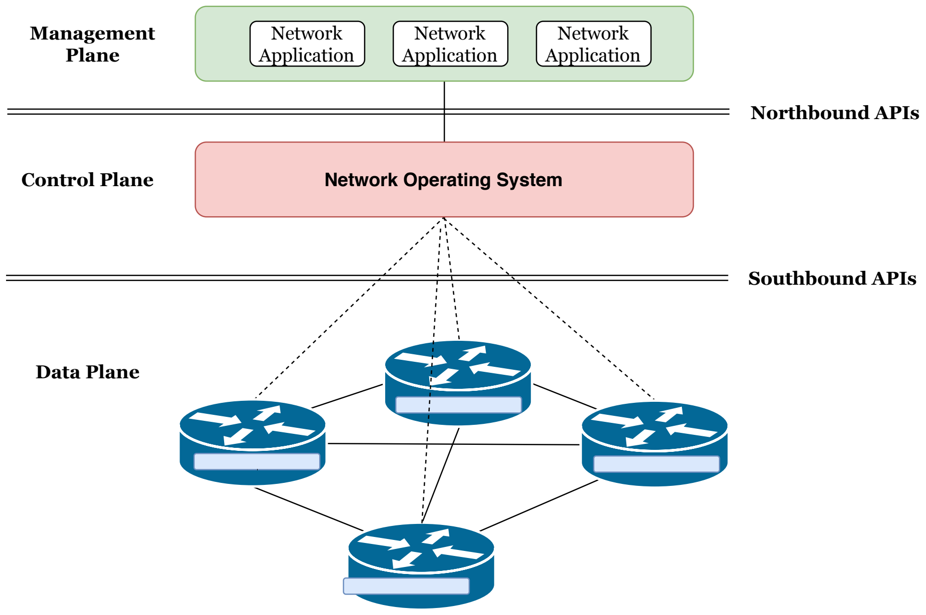


Figure 2.5: SDN architecture

tion of concerns has fostered network innovation from both academia and industry. Some vendors have developed highly-capable SDN-compliant forwarding equipment, several open source communities promote and develop NOSs projects, and a variety of research projects constantly tackle complex problems that were hard to address under traditional networking (e.g. network orchestration, global policy management, holistic network troubleshooting). The systems we describe throughout this dissertation are mainly network applications in the management plane of the SDN architecture. Our prototypes used the Aruba VAN [55] as control plane and hardware OpenFlow-enabled switches deployed at various locations of the campus network of the University of Kentucky as the data plane of our SDN network.

2.3.2 SDN Northbound and Southbound APIs

Application Programming Interfaces (APIs) are the means that allow a layer in SDN to exchange information with, and access features provided by, another layer. While specific implementations of SDN might have several APIs within an individual layer (e.g. built-in modules or internal processes communicating with each other), there are two major APIs in the overall SDN architecture, namely, the Southbound Interface and the Northbound Interface.

Southbound Interface: The Southbound Interface is used for communication between the controller and the forwarding devices. In OpenFlow, the NOS acts as a server whereas the switches behave as clients. Note that this role is only relevant for the establishment of the communication channel (i.e. the three-way handshake). In reality, OpenFlow messages can be sent in either direction in an asynchronous fashion as long as the control channel is active. Some example control messages include OpenFlow version negotiation (bidirectional), request for packet counters (controller-initiated), or network event reporting (switch-initiated). Over the years, several NOSs have added support for both newer and legacy protocols in addition to OpenFlow. For example, OVSDB [56] can be used to configure virtual Open vSwitch (OVS) nodes, NETCONF [57] and SNMP [58] enable compatibility with non-OpenFlow devices, and OpFlex [59] provides policy control for Cisco-only deployments. In this thesis, we focus our implementations around the OpenFlow protocol. Specifically, we use the NOS to modify the *Flow Table(s)* of OpenFlow-capable devices and determine how traffic flows are processed based on the contents of the packet headers (e.g. source IP, destination IP, incoming port, protocol). Section 2.3.3 discusses in more detail the ways of processing network packets using OpenFlow. We leave implementations of SDN using legacy devices as future work (although some of our initial efforts are included in [60]).

Northbound Interface: Unlike the Southbound Interface, the Northbound Interface does not have well-defined protocols binding the NOS with network applications. Instead, the Northbound Interface uses technologies from the software world such as REST APIs [61], gRPC [62] or Software Development Kits

(SDKs)[1] to allow network applications consume information and features offered by the NOS. While there have been some efforts to standardize the Northbound Interface [63], currently, NOS developers define the technology, API calls, and data models that management applications must use to fetch/push network information data. As a result, vendor lock-in scenarios at the control plane are not uncommon when developing network applications. We addressed this problem in [52] by adding a lightweight translation layer on top of the control plane. The translation layer (described also in Section 3.4.3) serves as a unified Northbound Interface for all the systems we describe in Chapter 3 and Chapter 5.

2.3.3 SDN Packet Processing: Reactive vs Proactive

In an OpenFlow-enabled SDN network traffic is processed primarily based on two components (1) the information present in the header of each packet and (2) the set of active *flow table* entries (also called OpenFlow rules) in a switch at the moment of packet arrival.

When a packet arrives at a switch, the switch's processing pipeline determines the modifications and actions that must be applied to a packet. In the initial experimental versions and version 1.0 of the OpenFlow protocol, the pipeline was composed by one individual flow table (the minimum number of tables for a switch to be OpenFlow compliant). Newer versions of the protocol (optionally) allow for more complex processing letting the packet be modified multiple times throughout the processing of linked flow tables.

Packets are matched against existing OpenFlow rules in the current processing table. Should a match occur, the actions found in the matching rule are executed. The actions may explicitly direct the packet to another table for further processing,

[1]A collection of libraries used for developing applications for a specific device or operating system.

Table 2.1: An example flow table with four OpenFlow rules

Match	**Priority**	**Counters**	**Actions**	**Timeouts**	**Cookie**
protocol=tcp, tcp_dst=80, ip_dst=3.4.5.6	100	10 pkts	set_ip_src=10.0.0.1, set_ip_dst=10.8.8.2, output=port 2	idle=120 secs	0xab
protocol=udp, udp_src=53	50	400 pkts	controller	0	0xae
ethertype=arp	1	10 pkts	go_to_table 200	0	0xff
*	0	3000 pkts	drop	0	0xff

modify some of the packet headers before exiting the pipeline, send the packet out of a particular port, or simply drop the packet. The OpenFlow specification [64], provides in great detail the list of (mandatory and optional) actions and instructions that can be applied to packets.

A flow table consists of multiple prioritized rules. The structure of each rule is shown in Table 2.1. The *match* specifies the field values a packet header must have for the rule to be activated (e.g. the third rule matches Address Resolution Protocol (ARP) traffic The *priority* establishes the order in which rules are evaluated in the table (higher priority rules are evaluated first). The rule *counters* contain statistical information about the number of packets (and bytes) that have matched the rule. The rule *actions* determine the operations or pipeline processing applied to a matching packet. Possible actions include (but are not limited to) rewriting packet header fields in the (rule 1), sending traffic to the controller (rule 2), forwarding packet to another table in the pipeline (rule 3), or simply dropping a packet (rule 4). Each rule has two *timeouts* that define the lifetime of a rule. A *hard timeout* specifies a fixed lifetime for the rule. An *idle timeout* determines for how long a rule may last without a packet match. For example, in Table 2.1 rule 1 expires after two minutes with no matching packets and the rest of the rules do not expire unless explicitly instructed by the controller. Lastly, each rule has a *cookie* set by the controller when the rule was installed. The (internal modules of) NOSs use cookies to simplify rule management. For example, in Table 2.1 the cookie value `0xff` could be associated to default rules pushed by the controller upon switch connection whereas any rule starting with `0xa` could represent a rule pushed as a consequence of a Northbound Interface request.

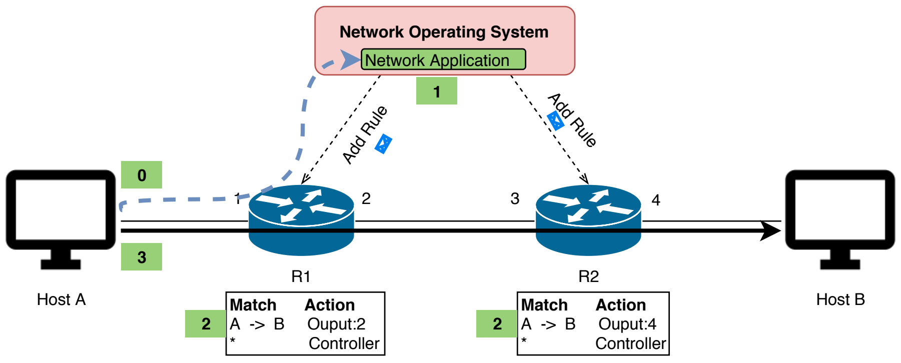
A rule that has all of its fields wildcarded is called a *table-miss* rule and determines the default packet processing for traffic that did not match any other rules. A table-miss rule is similar to the default route in a routing table in legacy networks where in the absence of a prefix match, a default route is selected to forward a packet to. In an SDN network, where SDN controller defines the decision logic of the network, it is important to determine what and when rules will be installed in the switch based on the packets that traverse the network. There are two basic *default* modes of packet processing, namely, reactive and proactive processing. We describe both modes below using the examples shown in Fig. 2.6.

Reactive: Under reactive processing (Fig. 2.6a), once a switch (e.g., $R1$) receives the first packet of a flow (step 0), the table-miss rule instructs the switch to send the packet to the controller. Then, at the controller, internal network applications get access to the contents of the packet and issue control messages based on the packet information (step 1). In the example shown in Fig. 2.6a, the network application installs two flow entries at $R1$ and $R2$ (step 2) that ensure subsequent packets of that same flow reach HOST B without contacting the controller again (step 3). The main benefit of the reactive processing mode is its flexibility to make decisions via internal controller applications for every unknown packet. For example, the *on-demand security exception* system (Chapter 5) inspects Domain Name System (DNS) responses to deploy exceptions "on-the-fly" for data transfers to cloud storage providers that adopt moving target defense practices. However, the major drawback of reactive processing is that in the presence of multiple flows the controller can get severely overloaded. A buggy network application could fail at handling bursts of requests introducing unexpected behavior to the network users.

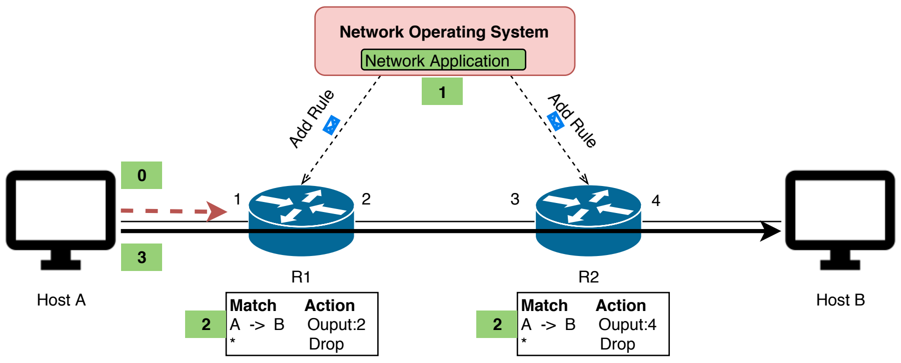
Proactive: In proactive processing, in order to prevent NOS overload, the default table-miss drops all packets unless there is a rule matching the traffic. In the example shown in Fig. 2.6b when a flow from HOST A is initiated, it is possible for the packets to get dropped (step 0) because of the table-miss action. Under proactive processing, network users need to wait until a controller module (possibly after an external request from a management plane network application) installs some flow entries at the switches (step 1 and 2). Upon rule installation, packets may traverse the network and reach their destination. The proactive processing mode is a conservative approach that is generally adopted when it is possible to know before-hand the flow characteristics and the actions that should be applied to the packets of such flow, and when controller overload is a concern. The systems presented in this dissertation that leverage a controller NBI assume flows (involving types of traffic, end points) are, for the most part, known in advance; therefore, proactive is the main mode of packet processing of the proposed systems. The main drawback of proactive processing is that on-the-fly dynamic packet processing is not possible because controller applications never make decisions (i.e., issue control messages) based on packet contents.

2.3.4 Leveraging SDN to Enhance Network Security

Under the SDN architecture, network application developers (e.g., researchers, vendors, technology companies) leverage the NOS's global view of the network and event reaction to develop a variety of network security solutions ranging from use-case-specific protection applications (e.g. the DNS protector app from HP [65]) to more complex frameworks that require custom agents running on middleboxes, control



(a) Reactive



(b) Proactive

Figure 2.6: Types of packet processing in SDN-enabled networks

plane Deep Packet Inspection (DPI), or unconventional application coding to provide security to a network.

In terms of network security policy enforcement, there are systems and frameworks that leverage OpenFlow to follow the same definition of a policy (*a set of rules*) that we used in the systems described in the following chapters.

Ben-Itzhak *et al.* [66] proposes the *EnforSDN* architecture based on the observation that middleboxes (e.g. Firewall, Intrusion Prevention System (IPS), IDS, etc.) are consist of three logical processing layers: (1) *Configuration plane* derived from a high-level policy description, (2) a *resolution plane* that uses concrete policy rules derived from the configuration, and (3) the *Enforcement* plane that corresponds to the low-level data plane instructions applied to a traffic flow. In *EnforSDN*, the resolution and enforcement planes of middleboxes are separated to reduce middlebox overload, network load, and latency. Fig. 2.7 shows the architecture and control plane flow of *EnforSDN* in a simple network with a firewall and two switches. Policies are fed into the middlebox as configurations. Then, once the source device initiates a flow, the first packet is sent to the network appliance for policy resolution, i.e., how to handle subsequent packets of that flow. Once a middlebox decides how to process packets in the flow, the decision is informed to the SDN controller (via a custom API) for enforcement using OpenFlow rules such that further packets of the same flow do not go through the appliance again. Unlike *EnforSDN*, our work derives policies from AUPs that are fed into the working memory of a separate decision system (Business Rule Management System (BRMS)) that generates the OpenFlow rules that must be installed in the network. We argue that middlebox modifications is expensive and infeasible in campus networks. Instead, we leave middlebox enforcement "as is"for general-purpose traffic and propose an exception system that can deploy middlebox-free paths on a per-flow basis for trusted users that share information about their traffic before communications start.

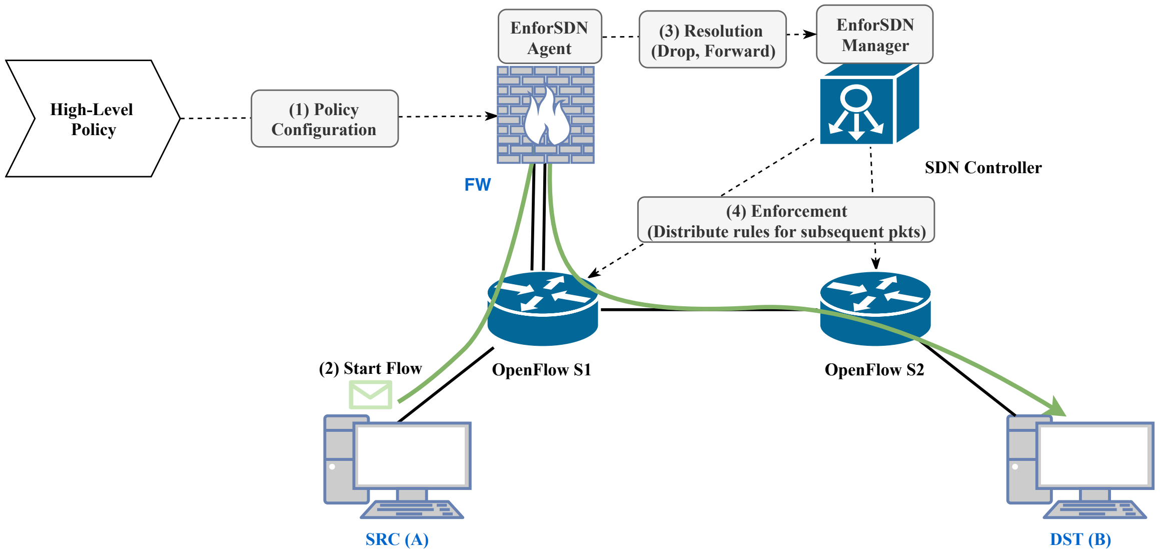


Figure 2.7: Architecture and control plane flow in *EnforSDN*

Bakker *et al.* [67] propose an OpenFlow-based network-wide virtual firewall that extends the functionality of traditional (perimeter) firewalls by allowing traffic filtering not only at the boundaries but within an OpenFlow-network. Fig. 2.8 shows an example workflow on how policies are enforced with the network-wide firewall application.

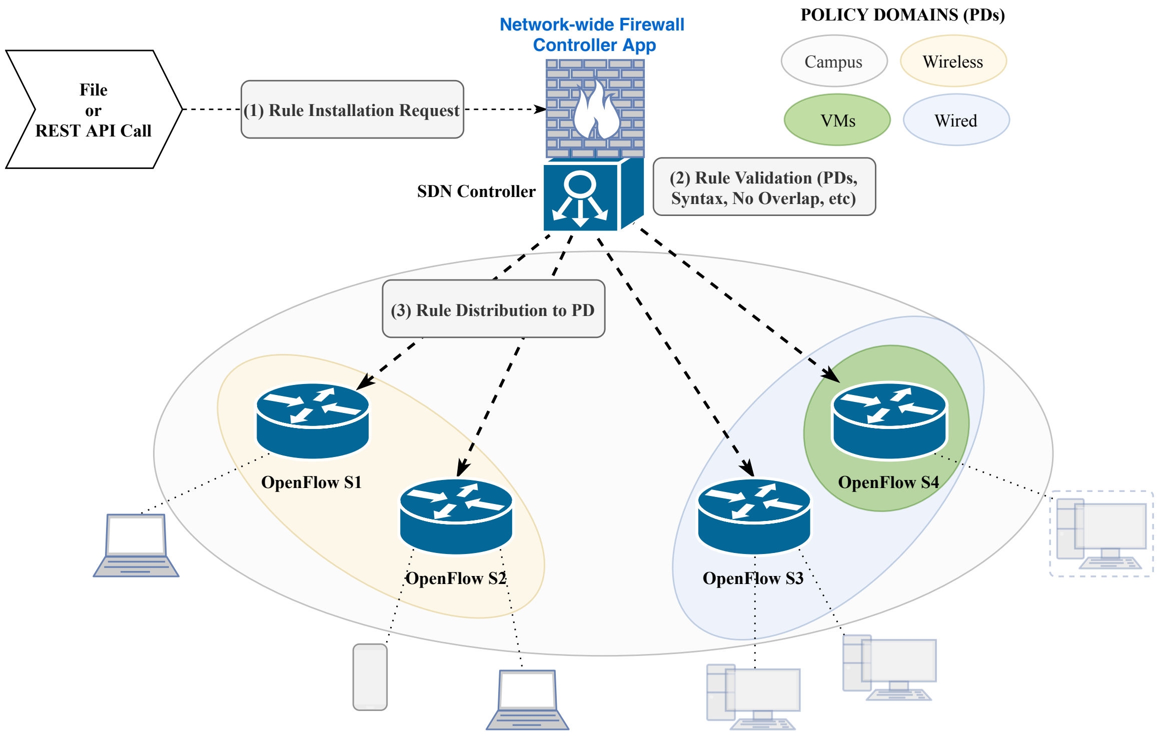

Network-wide Firewall Controller App
POLICY DOMAINS (PDs)
Campus
Wireless
VMs
Wired
File or REST API Call
(1) Rule Installation Request
SDN Controller
(2) Rule Validation (PDs, Syntax, No Overlap, etc)
(3) Rule Distribution to PD
OpenFlow S1
OpenFlow S2
OpenFlow S3
OpenFlow S4

Figure 2.8: Network-wide Firewall example deployment and rule installation workflow

Before any policy enforcement takes place, policy domains should be defined. In the example, there are four policy domains, namely, campus-wide, wireless, Virtual Machines (VMs), and wired. After switches have been assigned a policy domain, the network operator may start defining policies using the rule syntax shown below. Then, the network-wide firewall application checks the rule syntax, the existence of the policy domain, and the absence of potential policy conflicts with already deployed rules. After validation, the system schedules the deployment of the policy. Policies are defined as singleton rules (Access Control List (ACL) rules) that use the following OpenFlow-like syntax:

```
Rule r = {src_ip, dst_ip, protocol, src_port, dst_port,
↪  policy_domain, action, time_start, time_duration}
```

The *policy_domain* field enforces policies across a group of switches whereas the *time_start* provide basic policy scheduling functionality. PoLanCO (described in Chapter 3) does not have scheduling capabilities, however, Drools, the decision system used to generate rules, does. We leave adding scheduling functionality to PoLanCO as future work (Chapter 6). On the other hand, not only PoLanCO enforces policies on a group of switches but also PoLanCO can enforce policies that are expressed in terms of the varying types of end systems and middleboxes in the network (e.g., web servers, laboratory machines, firewalls, IDSs). Lastly, our work allows for the deployment of *on-demand security exceptions* as well as more complex policy actions than the basic filtering (i.e., allow and block) actions supported by the network-wide firewall application.

In Policy Graph Abstraction (PGA) [53] network operators from various units in a campus network specify policies simultaneously using network graphs. The input graphs are consolidated in one unified graph that holds the allowable communications between systems in the network. Policy graphs may include middlebox processing requirements as well. middleboxes are modeled using pyretic [68] programs that describe a middlebox's behavior. Rezvani et al. [69] extended PGA to let network operators specify priorities in policy graphs as well as blocking policies since PGA is mostly a whitelisting model. The added PGA extensions address problems found in current OpenFlow rule compilers where redundancy anomalies are common thereby reducing the number of actual rules being pushed to the network switches. We argue that both systems could be incorporated as part of the rule generation phase in PoLanCO's translation pipeline. PGA treats the network as "one big switch" making it challenging to deploy on-demand exceptions that allow specific flows traverse the network over an alternative (local) path. Although we share PGA's goal to automate the way network operators translate high-level policies into low-level network configuration commands, PGA and our work address the problem from two perspectives. PGA resembles diagrams network operators draw when designing policies, our work focuses on the definition of human-readable and technically-precise statements derived from AUPs that use imprecise language.

FlowTags [70] proposes an extension to middleboxes to support *tags* and a dedicated API that helps middleboxes interact with the NOS. FlowTags enforces network-wide policies by controlling the route of a packet in the network using the Type of Service (ToS)/Differentiated Services Code Point (DSCP) field of the IP packet header. We argue that modifying middleboxes is challenging because of the quantity and diversity of middlebox solutions available today. Moreover, reusing the ToS/DSCP IP packet header field could yield to unexpected behaviors of network traffic based on existing Differentiated Services (DiffServ) deployments in the network (e.g. voice, media streaming).

Qazi et al. propose SIMPLE [71], a network security approach the enforces policies by steering traffic through a desired sequence of middleboxes. Similar to our approach, SIMPLE assumes there is a network management mechanism that provides the controller with information about middleboxes (e.g., location, load, and capabilities). Policies in SIMPLE can only be expressed in terms of middlebox chaining (e.g. HyperText Transfer Protocol (HTTP) traffic must go through a firewall, then an IDS, and then a web proxy). Unlike the work presented in this dissertation, SIMPLE does not provide a human-readable policy language and does not allow for the enforcement of policies in the network that are usually embedded in the configuration files of end systems. Although the exception paths we implemented in Section 5.4.2 could be replicated in SIMPLE (i.e. specifying a policy that does not go through any middlebox), SIMPLE policies are specified by network operators for long timescales whereas in our exception system we deploy exceptions dynamically (on-demand) based on established trust relationships between groups of users and network operators.

OpenSec [72] is an OpenFlow-based framework where network operators can specify security policies in a *human-readable language*. Although OpenSec's language is more readable than what could normally be written using network programming languages (Section 2.4), we argue that PoLanCO provides better human-readability because OpenSec's language (1) still uses low-level identifiers (e.g., VLAN numbers, interface numbers, port numbers), (2) does not resemble human-readable sentences found in AUPs, and (3) only focus on the packet processing done by middleboxes and does not consider policies that are embedded in configuration files of end systems. Moreover, OpenSec assumes a topology where middleboxes are removed from the path between the LAN and the Internet which is not always possible in campus networks. Similar to FlowTags, OpenSec requires modification of middleboxes, specifically, an agent that can report the results of the analysis to the NOS. If an agent can be added to middleboxes, PoLanCO could be easily extended to support the same capabilities of OpenSec's language with better human-readability and relaxing the assumption of middlebox placement.

CloudWatcher [73] is a security monitoring framework for the cloud. Similar to OpenSec and SIMPLE, the operator describes the security services (i.e., middlebox functionality) that must be applied to a particular traffic flow specified via an 4-tuple. CloudWatcher focuses on optimal routes to send traffic to Middleboxes in a cloud environment with multiple alternative paths. In contrast, in our exception system (Chapter 5) our goal is to compute paths that actually bypass middleboxes using the Neo4j graph database.

Shin et al. propose FRESCO [74], an OpenFlow programming framework that allows network operators to design, implement, and compose security detection and mitigation modules using a simplified scripting language for the NOX [75] controller. Albeit different in syntax, the scripting language could be placed at the same level of abstraction as the Drools code we described in Section 3.3. We used a mapping function to translate PoLanCO statements into valid Drools code to generate OpenFlow rules that enforce policy. We could change the mapping function to generate valid FRESCO modules. However, policy enforcement in FRESCO requires the analysis of network traffic at the NOS level; therefore, the NOS could easily be overloaded and become bottleneck of the network. Instead, PoLanCO follows a proactive packet processing where policies are enforced at the data plane via OpenFlow rules with minimal intervention of the NOS.

The systems we describe below address specific network security scenarios that are complimentary to the work we describe in this dissertation. We reference them for completeness. Mehdi et al. [76] shows how four traffic anomaly detection algorithms (Threshold Random Walk with Credit Based Rate Limiting [77], Virus Throttling [78], Maximum Entropy Detector [79], and NETAD [80]) can be implemented in SDN using OpenFlow-enabled switches and the NOX controller processing only a small fraction of the packets in any individual connection. OrchSec [81] proposes an architecture to develop security applications in SDN networks that can be spawned across multiple (possibly different) NOSs. The authors discuss increasing iterations of the architecture as well as a detailed example on how to develop an application that addresses problems of DNS amplification attacks (a type of Distributed Denial of Service (DDoS) attack). SDN-Guard [82] proposes a scheme to protect SDN networks against Denial of Service (DoS) attacks by rerouting, adjusting timeouts of rules, and aggregating rules of malicious flows. SDN-Guard relays switch-to-controller packets to an external IDS that informs the threat probability of each flow. FleXam [83] addresses the problem that IDSs have to inspect massive loads of traffic to perform security analysis. The authors propose a sampling extension for OpenFlow in the form of an additional "sample" action. The extension allows the controller to poll a reduced number of packets (based on stochastic or deterministic sampling) and access only specific parts of the packet contents (e.g., only headers, only payload). Neu et al. present a lightweight IPS [84] that utilizes switch counters to prevent port scanning attacks. *Network Security Caps* can benefit from the approach to not only enforce policies that are embedded in configuration files but also protecting end systems from port scanning attacks.

HoneyMix [85] and HoneyProxy [86] are SDN-based systems that aim at defeating honeypot well-known detection techniques [87, 88]. Zhao et al. [89] propose a fingerprint hopping method that uses moving target defense to interact with a network attacker in a honeynet. PoLanCO statements could be written to direct infected hosts to any of these systems. Lastly, Community Connect (CoCo) [90] allows the creation of on-demand VPN services via an easy-to-use web portal without requiring network operators to manually configure network devices. We believe that our exception system could leverage the dynamic nature of CoCo to offer the deployment of on-demand VPN services.

2.4 Network Programming Languages

We described in the previous section various frameworks and systems addressing policy enforcement practices in an SDN network. In the majority of the cases, the network applications were developed using general-purpose imperative programming languages (e.g., java, python, C++) to interact with the services, APIs and abstractions provided by several NOSs (e.g. Floodlight, ONOS, OpenDaylight, etc.). This section presents existing work in a new class of languages called *network programming languages* (NPLs). NPLs are a type of Domain-Specific Languages (DSLs) developed to write network applications where the network operator specifies *what* must happen to arriving network packets rather than *how* it is done. As a result, network programming languages are mainly written on top of declarative programming languages (with the exception of Pyretic [68]) although some NPLs follow a functional reactive programming approach for event-driven applications where network events (e.g. resource discovery, link congestion) trigger policies that are expressed in terms of allow-or-deny actions applied to underlying switches. NPLs are surely a higher-level abstraction than traditional per-device configurations. However, we suggest that NPLs are not suitable for policy definition because they impose a steep learning curve on network operators who historically have not been application developers but rather network application users. We list NPLs below for completeness and acknowledge that most of them could be used as the mechanism that generates configurations (OpenFlow rules) of the policy language we present in this dissertation. NPLs have considerably evolved since the inception of OpenFlow and the adoption of NOX [75] as the default control plane back in 2008. FML [91] was NOX's *de-facto* policy language that triggered policy conflicts due to distributed authorship within a single policy domain. Nettle [92] brought in the manipulation of series of messages rather than individual message processing. Procera [93] let developers express policies based on external events (e.g. QoS, intrusion detection). Frenetic [94] and Flog [95] propose a three-stage approach for NOSs policy enforcement: (1) query network state, (2) express policies, and (3) reconfigure network. Both NPLs hide low-level details relative to rules and network events. Regular expressions were introduced in FatTire [96] to describe network paths (e.g. "*" represent all traversing paths). The authors of NetPlumber [97] introduced Flowexp, a language that also relies on regular expressions and whose goal is to check constraints on the history of flows. Pyretic [68] introduced the parallel and sequential composition operators in python programs to allow for collaborative application development. Netcore [98], based on Frenetic, provides an enhanced compiler that divides programs into two pieces, one that runs on the switches and another that runs on the controller. Flowlog [99] resembles SQL and presents an interface that treats the control- and data-planes as one individual plane. Merlin [100] and Kinetic [101] are more advanced frameworks that provision network resources (e.g., bandwidth) and provide abstractions that automate changes in network policy. NeMo [102] is one of the first attempts for expressing human intent-based policies to create virtual networks via commands that are sent along REST calls. Lastly, MPI [103] proposes a service-oriented policy language characterizing policies as services established between two end nodes (e.g. hosts, subnets) through a traffic specification pattern (e.g. HTTP, SSH) that traverse any number of network functions (e.g. load balancers, byte counters, firewalls). This policy language is heavily based on PGA [53].

Chapter 3. PoLanCO: A Policy Language for Campus Operations

Network policy definition is a key component in network deployment and administration. Defining policies ensures that operations in universities are legitimate and secured, follow standards and specifications, abide by federal regulations, align with business objectives and procedures, and ensure adequate provisioning of services. Committees or groups of business experts consisting of high-ranked senior employees, referred to as Policy Writing Committees (PWCs) in this dissertation, are usually the authors of network policies. However, the points of view of PWCs typically focus more on the global requirements and procedures of the organization rather than the low-level configuration details needed to adapt network devices to support the network policies. The documents that PWCs create are oftentimes called Acceptable Use Policies (AUPs). AUPs use generic terms and vague wording (business jargon) and are commonly the only reference point and source of truth for network operators to configure network devices to enforce policy. Consequently, network operators, based on their expertise and interpretation, have to carefully read the AUPs to understand the purpose of the policy, extract relationships and conflicts among the AUPs, and derive the technical details and enforcement mechanisms required to deploy the policies. For example, network operators issue CLI instructions on network devices (e.g. switches, routers, middleboxes, etc.), and server administrators modify application-specific configuration files to enforce AUPs.

Clearly, there is a substantial gap between the definition of a policy and its enforcement that makes the policy translation process manual and error-prone for network operators and system administrators. Nevertheless, with current advancements in network architectures and existing practices based on business rules and procedures, it is possible to develop software-based intermediate mechanisms that can automate the translation from of imprecise AUPs into valid and correct network configurations.

In this chapter, we introduce a *Policy Language for Campus Operations (PoLanCO)* based on (1) current technologies used in businesses for policy definition via Business Rule Management Systems and (2) the Network Security Caps policy enforcement layer presented in Chapter 4. Network administrators and system administrators can use PoLanCO to precisely define a large number of policies with a simple human-readable syntax. PoLanCO provides a high-level abstraction that hides the low-level details and syntax of device CLI instructions and configuration files, and automates the deployment of policies on the network infrastructure based on the fine-grained flows diverse applications generate. We present examples on how to write PoLanCO statements that represent the AUPs that PWCs write and show how the simplified PoLanCO statements enforce network policies in various environments and locations of the campus network.

3.1 Motivation

In Chapter 1 we highlighted the importance of defining network policies in campus network infrastructures. Network deployments are becoming larger and more complex, requiring not only constant physical infrastructure renovations and hardware refresh cycles (i.e., device installations, cabling) but also revised rules and policies regarding how all these new devices should interact with each other and with any existing system/service in the network.

Normally, the outcome of an organization's PWC is a set of non-technical policy document(s) (AUPs) that use business-like language to inform network users about the current regulations governing IT infrastructure-related activities. Unfortunately, the statements found in policy documents are predominantly imprecise and make it hard for network operators to understand or know how to encode the policy into the network equipment. Moreover, the current policy translation process does not provide feedback to the PWCs in charge of writing the AUPs, in particular, feedback from the network operators in charge of deploying the policies. PWCs do not necessarily understand the low-level mechanisms used to enforce policies (e.g. commands, configuration files, protocols) and thus do not understand the challenges (and sometimes impossibility) of implementing the policies they write. Network operators, though issuing syntactically valid commands and configurations, could partially and/or incorrectly enforce high-level policy due to misinterpretation of the written policy statements.

Take for example the short excerpt from the Internet-facing web server security guidelines found on the Carnegie Mellon University's website [104] and shown in Fig. 3.1. While PWCs are likely able to understand (most of) the contents of the high-level policy document excerpt, there is hardly a direct association between guidelines expressed in the AUP with the low-level details (i.e. variable definitions, port numbers, syntax) found in the configuration files of web servers. As new, more complex, technologies continue to be developed, associating policy documents with enforcement mechanisms becomes harder. For example, not long ago Apache [105] was the *de facto* solution for hosting a web site. However, newer solutions such as Flask [106] or Nginx [107] are commonplace today due, in part, to their "easy-to-use" setup nature and other capabilities they offer (e.g. efficient load-balancing, proxy services). Fig. 3.2 shows a snippet of an Nginx web server configuration file implementing the policy. After comparing the policy with the server configuration file, we can see that there is no direct mapping between the high-level statements (the policy definition) and the configuration details (the policy implementation). In fact, they could be in conflict. Take for example the guideline shown in line 1 of Fig. 3.1: *"Configure web server to meet recommended vendor best practices"*, system and network administrators responsible for setting up the server could interpret the portion *"recommended vendor best practices"* as configuring the web service software with the default settings. However, the default settings normally open port 80 (i.e. HTTP) in both IPv4 and IPv6 (see lines 2-3 in Fig. 3.2) in the server. Enabling port 80 may be undesirable as it allows the transmission of web traffic in clear text, possibly exposing authentication credentials or other sensitive information. Moving forward in the policy document, line 3 requires that the server *enables necessary web services and disables all others*. Though the configuration file does allow an administrator to specify the port on which the web service will listen to incoming connections, disabling other unnecessary services is typically performed via OS utilities and not the web service configuration file.

```
1  4. Configure Web server to meet recommended vendor best practices
2    - Install the Web server software on a dedicated host
3    - Enable necessary web services; Disable all others.
```

Figure 3.1: CMU web server policy excerpt

nginx.conf

```
1  server {
2      listen         80 default_server;
3      listen         [::]:80 default_server;
4      server_name    example.com www.example.com;
5      root           /var/www/example.com;
6      index          index.html;
7      try_files $uri /index.html;
8  }
```

Figure 3.2: Example web server (Nginx) configuration excerpt

Moreover, it is hard for network operators to recall what policies caused the current state of a configuration file. It would be beneficial if network operators could use technically-precise documentation to not only track changes in configuration files but also as a way to obtain feedback from the PWC–that is, do these precise statements match what the PWC intended in the original policy? Answering the question is even more challenging if we take into account that configurations could have been written in the past (months or years ago) by other network operator (with varying expertise and skills) and perhaps under different policies.

Besides the lack of feedback between PWCs and network operators, the translation of high-level policies into low-level configurations today is for the most part manual. Even under the assumption that network operators are able to accurately abstract out technical details and commands from AUPs, configurations need to be individually pushed to the set of network devices enforcing the network policies. Albeit possible—after all, this is how policies have been implemented thus far—this approach to policy enforcement is error-prone, time consuming and difficult to verify. In fact, others [108, 109] have reported that human errors occurring during device configuration are the main cause for network outages and unexpected behaviors in networks.

As shown in Section 2.3, challenges associated with human/manual configuration in network management has been one of the main motivations behind the development of emerging network architectures that try to simplify and centralize configuration/management [40, 38, 109]. The ability of an SDN controller to push out configuration (in the form of rules) to routers/switches has the potential to reduce the intervention of operators during the implementation of AUPs. Further, the integration of SDN with automated systems that implement business policies present an opportunity to adapt solutions from the business world to network configuration challenges, bringing PWCs and IT staff together in AUP enforcement tasks.

We introduce a "middle-ground" mechanism in the form of a DSL (called *PoLanCO*) to tackle the challenges network administrators face translating policy into configuration. Specifically, network operators are now able to convert high-level guidelines and directives into a series of short, human-readable and technically precise (PoLanCO) statements. The produced list of statements could be used to validate if the abstracted “middle-level” policy (i.e. PoLanCO statements) is indeed what the PWC expressed in the high-level policy documents. In other words, feedback becomes a key component of the translation process because it allows network operators to revise and change the derived (PoLanCO) statements to appropriately represent the policies defined by the PWC. In addition, since the derived short technical statements follow a structured human-readable syntax, software that uses emerging technologies in networks and business operations can be developed to ensure that the statements are automatically translated into low-level enforcement mechanisms. Thus, reducing the intervention of operators in the deployment of the policies.

In the following section, we describe the design goals of PoLanCO. We cover aspects such as the need to know the characteristics and roles of network systems, the goal of remaining vendor and protocol neutral, the emphasis on avoiding the use of network-level identifiers to express policies, and the opportunities to improve workflows that depend on the dynamics of the network (e.g. changing the role of a network node, or deploying an exception to global network policies).

3.2 Design Goals

As noted earlier, technically-precise, human-readable language would be useful as a way to provide feedback between PWCs and network operators, and to automate the translation process from policies into low-level configurations. We describe below the set of features and characteristics that such a language should have:

Avoiding Low-Level Network Identifiers: Low-level network identifiers are, unsurprisingly, commonplace when operators reason and analyze how to translate policy documents into configuration files or device CLI instructions. Low-level network identifiers are the type of information that network devices and end systems expect as input in order to process network traffic. However, thinking about policy at such a low level of technical detail is unnatural for humans that better reason in terms of words, characteristics, roles, and the like. Take for example the way we connect to websites. We use human-readable domain names (e.g. `www.google.com`, `www.uky.edu`, `www.wikipedia.org`) as opposed to their corresponding IP addresses (even network operators prefer not to use them!). Instead of using network identifiers, we can assign each identifier a role and/or trait (e.g. group of users, device types, type of traffic) and raise the level of abstraction of technical configurations. Human-readable statements based on such assignments can (1) be easily understood by PWCs, (2) help move towards the documentation of policies that were previously deployed, and (3) be expanded by translation mechanisms into low-level details when needed.

Network Description: The proposed language will need to have terms to describe common networking concepts like file servers, web servers, printers, firewalls, secure channels, blocking, mirroring, etc. and the translation mechanisms used to install low-level configurations will know what portions of the network configurations should be pushed to, and how the configurations should look.

Event-Aware: While some of the policies that need to be enforced on networks do not change over long periods of time, some policies are dynamic, periodic, or based on the current status of the system. The language should allow the specification of the type of actions that must be triggered whenever a relevant event happens, thus providing context to the policy. Events may include situations like the addition of new equipment to the network, detection of a malicious actor or an infected machine, changes in the policy, current time of day, etc.

Extensible and Vendor-Agnostic: The language should be generic enough to hide proprietary, vendor-specific characteristics of the network as well as the complexities of the low-level mechanisms present in the network. Resolving what features or protocols to use in order to enforce the policy **should not** be part of the language used to define the policies but a part of the system that translates language statements into low-level instructions. As we described in earlier chapters, networks have continuously evolved in complexity over the years. Whether it is the type of systems that are part of the infrastructure, or the features supported by existing equipment, we require the language to be easily expandable and allow for new types of policies to be specified.

Exceptions: The language should allow for the explicit specification of authorized exceptions. As we will describe in Chapter 5, exceptions are a common part of a policy ecosystem and the current process to deploy them is largely manual. IT departments often require that users in need of an exception submit a request (say via e-mail or a web form) to obtain permission and install an exception. Typically, a PWC analyze exception requests every few weeks or months. Once the risks and implications of the request are evaluated, the deployment of the exception finally takes place (or the request is rejected). While we agree that pre-authorization is important for the deployment of exceptions, the current process is manual and heavily dependent on human intervention (see Chapter 5). We want to make PoLanCO the middle layer where both policies and their exceptions can be written and automatically deployed. The language will help keep the documentation consistent over time, while handling dynamic workflows that require timely-deployed short-lived exceptions.

3.3 Approach: Leveraging Business Rule-Based Management Systems

Defining and enforcing policy based on a series of facts, events, and characteristics is not unique to networking. For example, businesses define policies in terms of the products and services offered, the interactions with existing and prospective customers, and the federal and state legislation businesses must abide by. With a goal of automating processes that were otherwise manual or error-prone (e.g. profiling customers), business management and IT departments joined efforts to develop decision systems, called *BRMSs*, that make choices based on a given set of facts and events.

At a high-level, BRMSs consist of four main components, namely, a rule repository, a set of tools for rule authoring and maintenance, data objects that represent the current facts in the system, and a runtime environment that invokes the rules. Fig. 3.3 shows how the components of a BRMS interact with each other. The *rule repository* is the centralized place (similar to a database) where all rules are stored. A business rule can be defined as a statement that aims to influence or guide behavior in an organization [110]. A rule is the place where the decision logic (i.e. what to do) is separated from the core production application code (e.g. how to do it). Business rules are created, modified and removed using *rule authoring tools*. Authoring tools help business experts and IT staff in the definition, design, documentation, and modification of business rules. BRMS vendors and/or third-parties develop these tools that vary from simple markup syntax to sophisticated web-based applications. Business rules are defined based on the current facts (also called *working memory*) of the system. Facts may be added to the runtime environment both at startup or as a consequence of other rules being activated. The runtime environment, also called *rule engine*, implements a continuous execution cycle based on the rules and the current status of facts when each rule is being evaluated. The output of the rule evaluation include the addition (or modification/removal) of facts—possibly triggering another set of rules—and the execution of external procedures, for example, production code that enforces the decision that has been made.

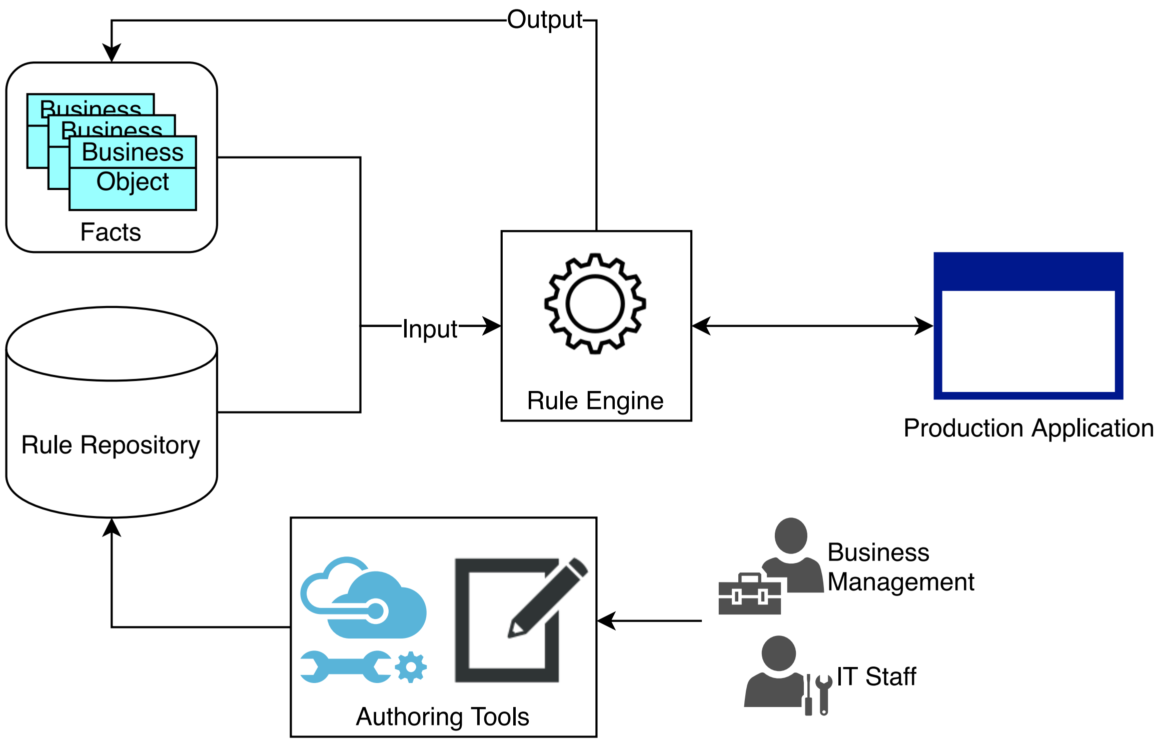


Figure 3.3: High-level architecture of a BRMS

3.3.1 From Business Rules to Network Policies

BRMSs normally come with syntax restrictions to write business rules that are heavily influenced by the underlying programming language used to develop the actual BRMS. Currently, there is no unified or standardized language to specify rules seamlessly across BRMSs solutions. However, the structure of the rules is, for the most part, the same. In addition to programming language preambles (e.g. environment variable definition, library and data structure importing), business rules have two main constructs, namely, a conditional (antecedent) and a body (consequent) [111]. The *conditional*, typically represented by the keyword `when`, is used to express the set of constraints that must be satisfied in order to activate a rule. Constraints are written in terms of the properties of the data objects that are part of the working memory of the BRMS. The *body*, typically represented by the keyword `then`, contains the set of actions that are executed if the constrains are satisfied. Unlike constraints, actions may include API calls to either local processes or external services (e.g. logging systems, monitoring frameworks, network equipment), or modifications to working memory data objects. Some example rules that use the Drools [112] syntax are shown in Fig. 3.4. The first rule (i.e. *welcome.rule*) shows an example banking organization's policy for customers whose loan was approved (line 3). The body of the rule (lines 5-6) includes the actions that must be executed whenever a loan is approved. Note that in *welcome.rule*, the actions do not modify the status of the working memory but rather execute external processes (logging and mailing). However, the action in line 7 of the second rule (i.e. *auto-approve.rule*) does cause a modification to the working memory. The rule changes the status of a loan application when the amount requested is less than 5,000 dollars and the credit score of the applicant is greater than 675 (lines 3-4), thereby causing the welcome rule to be triggered whenever a loan is auto-approved.

Business rules are lists of statements that determine whether some action should happen (or be avoided) based on well-defined constraints. As presented earlier, the rules are often expressed in BRMSs using a specific syntax that is typically provided by the BRMS vendor. BRMSs are flexible mechanisms that could be leveraged to enforce policies in networks in the same way that it enforces business policies and automates processes. The flexibility of a BRMS comes from the fact that any data object can be part of the working memory. Based on the examples provided above, a banking institution would include relevant information like customer data (e.g. age, credit score, active products, etc), or product information (e.g. type of loan, amount requested), as facts in the BRMS's working memory. Moreover, facts associated with events (e.g. applying for a product such as a credit card or a loan) can also be registered in order to expand the type of policies that can be written as a result of an event. Network policy management adheres to the same model. Some of the facts needed to enforce network policies can be found in information about welcome.rule

```
rule "welcome message"
  when
    $app: LoanApplication( status == status.APPROVED);
  then
    LOGGER.debug("Sending welcome e-mail to customer  for loan ",
    ↪  $app.custID, $app.ID);
    mailService.send($app, mailType.WELCOME);
end
```

auto-approve.rule

```
rule "auto-approve loan"
  when
    $a: LoanApplication(amountRequested <= 5000)
    $c: Customer( id == $a.custID, creditScore >= 675)
  then
    LOGGER.debug("Auto-approving loan for $ to customer ",
    ↪  $a.amountRequested, $a.ID);
    modify( $a )   setStatus ( status.APPROVED )
end
```

Figure 3.4: Example business rules written in a BRMS (Drools)

the nodes and connections of the underlying topology. For example, consider the policy commonly found at various university websites (shown in Fig. 3.5) [113] that requires network operators and system administrators to disable insecure protocols from network devices and systems in the network. A BRMS can automate the implementation of the policy. Fig. 3.6 shows an example Drools code that enforces the policy. Assuming nodes are part of the BRMS working memory, the conditional of the rule (line 3) selects all the nodes where the policy will be enforced (i.e. nodes that are of type `NETDEVICE`). Then, the body of the rule includes configuration details need to enforce the policy. For example, line 5 specifies protocols numbers of insecure protocols such as File Transfer Protocol (FTP), telnet, and HTTP; in line 6 `Config` objects are created using information about each selected node in the conditional, the protocols, and the action representing the policy decision; and line 7 launches an internal function that uses the created `Config` objects to push actual configurations (e.g. OpenFlow rules) into the selected network nodes.

```
Applications which transmit sensitive information over the net-
work in clear text, such as telnet and ftp, are prohibited and
will be blocked
```

Figure 3.5: Excerpt from the network usage policy of the University of Missouri-St. Louis

```
1 rule "disable-insecure-protocols"
2   when
3     $n: Node( type == type.NETDEVICE);
4   then
5     protocols = new ArrayList<Integer>(Arrays.asList(21,23,80));
6     cfg = new Config($n, protocols, PolicyAction.BLOCK);
7     ConfigPusher.push(cfg);
8 end
```

Figure 3.6: Example Drools code that generates configurations for network devices in working memory

As described in subsequent sections, BRMS code has the potential to enforce a large number of policies provided the working memory contains specific details and characteristics about the network such as the role a node has in the network (e.g. firewall, L3 router, L2 switch, printer, etc.), the enforcement mechanisms every node supports (e.g. different OpenFlow versions, remote CLI commands, NETCONF), the type of node (e.g. network device, end system), the status (e.g. infected, quarantined, up, down), and the way nodes are connected with each other (e.g. link capacities, VLAN information).

3.4 Implementing a Domain-Specific Language

BRMSs provide tools (typically out-of-the-box) for writing business rules that hide the complexities and verbosity found in traditional programming languages. Nonetheless, even though the rule-definition syntax is greatly simpler than regular programming language code, it contains several elements (e.g. symbols, keywords, identifiers, annotations) that are not typically found in human sentences; therefore, readable statements are difficult to construct using BRMS-specific code. For example, the body part of a Drools rule is particularly hard to read because it often contains (multiple lines of) java code to launch other application processes (e.g. line 6 in Fig. 3.4 or lines 5-7 in Fig. 3.6).

By contrast, PoLanCO was intentionally designed to severely restrict the token namespace that operators may use to write human-readable policies. Yet, PoLanCO is translatable to machine code. Fig. 3.7 shows a concrete example of a top-down translation pipeline (i.e. set of steps) that starts with the AUP written by a PWC [114] and ends with the generated Drools code that pushes network configurations onto the network. Unlike the contents of policy documents written by PWCs (e.g. the security policy at the top of the pipeline), network operators write PoLanCO statements that are human-readable, technically precise and derived from the contents of an AUP. The statements allow network operators to have an unambiguous interpretation of the policy that is being enforced without getting involved in the complexities of complex Drools (and java) code.

Once a network operator writes a PoLanCO statement s, s is divided into groups of words $w_0, ..., w_n$ where each w_i is passed as parameter to a translation function $T(w)$ that generates valid Drools code. Table 3.1) shows five w inputs and their corresponding Drools code. While the first two groups of words (i.e. *policy* and *policy priority*) are simple word replacements, the rest of the groups of words generate more verbose Drools code that is hidden from an operator's perspective.

The resulting Drools code for the firewall policy example is shown at the bottom of the pipeline. The generated code not only interacts with the working memory of the BRMS (i.e. identifying firewall nodes) but also launches back-end methods that build OpenFlow rules (i.e. `ConfigGenerator`), resolves types of traffic and end points (i.e. `aliasEvaluator`), and issues REST requests to the NOS to push the configurations (i.e. `ConfigPusher`).

Table 3.1: Translation function $T(w)$

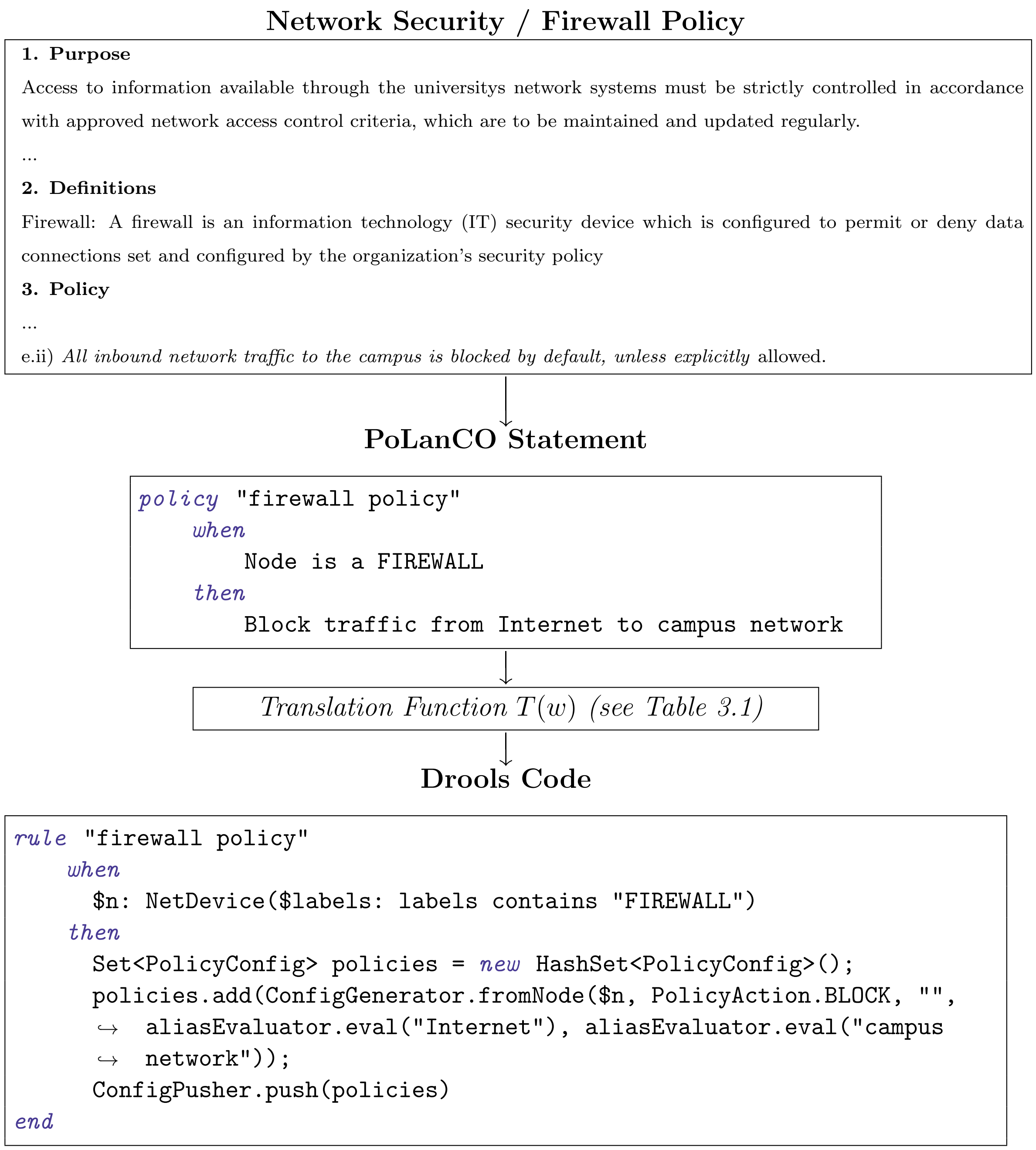
PoLanCO Grammar w	**Drools Code** $T(w)$
`policy`	`rule`
`policy priority`	`salience`

`[Nn]ode is a {type}`	`$n: NetDevice(`
	` $labels: labels contains "{type!uc}");`

`{action} {param} traffic`	`policies.add(`
	` CfgGen.fromNode(`
	` $n, PolicyAction.{action!uc}, {param},`

`from {src} to {dst}`	`aliasEvaluator.eval("{src}"),`
	`aliasEvaluator.eval("{dst}");`
	`CfgPush.push(policies);`

Translating PoLanCO statements into executable Drools code is only a portion of the complete workflow that starts with PWCs writing AUP documents and ends with low-level device configurations enforcing the policies. Assuming an operator has read and interpreted the contents of AUPs, Fig. 3.8 shows the steps that happen when a policy is actually enforced in the network using PoLanCO (the translation process is skipped as it was already described). First, the team of network operators must agree on a descriptive meaning for every low-level identifier needed to enforce a policy. For example, associating (1) IP ranges such as 10.10.0.0/20 to a group of devices like mobile devices connected in the university library, (2) individual IP addresses (123.456.1.3) to university authorized servers such as DNS or Dynamic Host Configuration Protocol (DHCP) servers, or (3) port numbers (80, 443, 9100) with types of traffic such as web or print jobs. The association between identifiers



Network Security / Firewall Policy

1. Purpose

Access to information available through the universitys network systems must be strictly controlled in accordance with approved network access control criteria, which are to be maintained and updated regularly.

...

2. Definitions

Firewall: A firewall is an information technology (IT) security device which is configured to permit or deny data connections set and configured by the organization's security policy

3. Policy

...

e.ii) *All inbound network traffic to the campus is blocked by default, unless explicitly* allowed.

PoLanCO Statement

```
policy "firewall policy"
    when
        Node is a FIREWALL
    then
        Block traffic from Internet to campus network
```

Translation Function $T(w)$ (see Table 3.1)

Drools Code

```
rule "firewall policy"
    when
      $n: NetDevice($labels: labels contains "FIREWALL")
    then
      Set<PolicyConfig> policies = new HashSet<PolicyConfig>();
      policies.add(ConfigGenerator.fromNode($n, PolicyAction.BLOCK, "",
      ↪  aliasEvaluator.eval("Internet"), aliasEvaluator.eval("campus
      ↪  network"));
      ConfigPusher.push(policies)
end
```

Figure 3.7: Translation pipeline from a high-level network policy into Drools code

and high-level descriptions needs to be first fed into the system before any PoLanCO statement is written. A mechanism to specify identifier-description associations is described in Section 3.4.1. Additionally, in order for the policies to be implemented in the network, the translation system (that translates PoLanCO to SDN configurations) must discover the network topology and role of each node in the network graph. The underlying SDN network is assumed to be capable of discovering the campus network topology and learning the role of each node in the topology. Once the topology is discovered, it becomes part of Drools' working memory; at this point, network operators can start writing multiple policies using PoLanCO.

All the PoLanCO statements are evaluated and mapped into Drools code following the procedure explained above. The back-end components (written in Java) launched by the generated Drools code contact the SDN Northbound Interface via standard REST API requests. The SDN controller processes these requests by sending *FLOW_MOD* OpenFlow messages that instruct the switch (or switches) to add entries to the flow table; thereby, enforcing the policy (Chapter 4 describes the enforcement layer in more detail).

3.4.1 Network Information Gathering

One of the reasons why it is difficult to express campus network policies in terms of low-level identifiers is that often it is not straightforward to map those technical details into high-level entities. Even though operators might be familiar with low-level networking addresses, protocols, services, etc. using such jargon as a means of feedback with PWCs hampers the opportunity to establish a two-way dialogue between both parts.

In order to improve the expressiveness and precision of a policy, every low-level network identifier must have a high-level meaning associated to it. The association is what enables the construction of precise human-readable network policy statements. For example, MAC and IP addresses can identify individual users, subnets could identify groups of users, VLANs and port numbers could represent types of traffic. Note that these associations are already being made when operating a campus network. However, hardly ever the meaning of low-level identifiers is used in AUPs.

In order to incorporate the mappings into PoLanCO, we require the topology

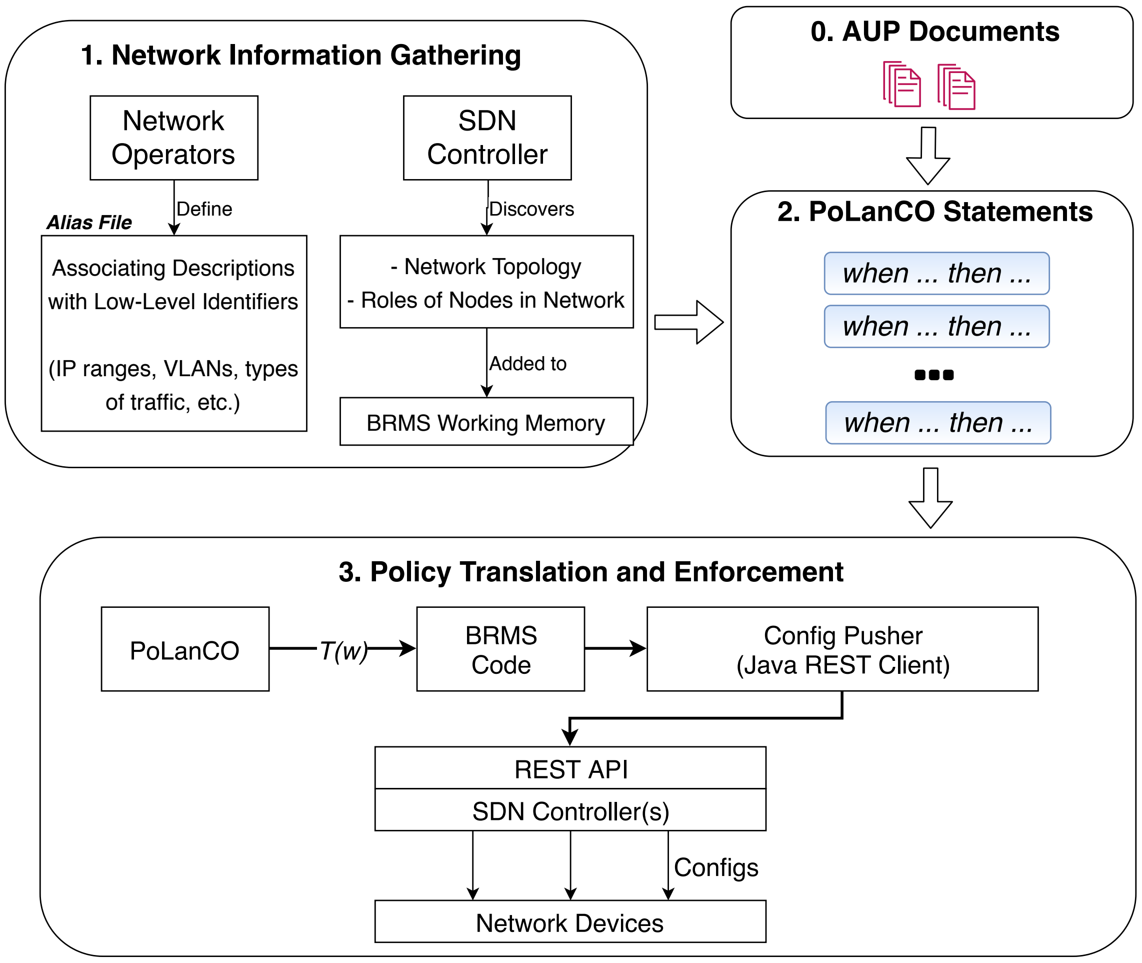


Figure 3.8: Workflow to enforce policies using PoLanCO

discovery to be performed before any policy is defined. Gathering the network information can certainly be done in multiple ways. For example, Cisco DNA [19] relies on their *Identity Services Engine* to map network user IDs to various end systems (a similar approach can be achieved using alternative implementations of RADIUS servers). For simplicity, in our prototype we perform information gathering on two fronts. Namely, an *alias* file and the discovery capabilities of SDN controllers. Network operators must define the associations in a static file that we refer to as `alias` file. The file is formatted in YAML [115]. Fig. 3.9 shows an excerpt from an example alias file.

The format used to represent each piece of network information is simple yet *template.alias*

```
1  alias: private-addresses
2  specs:
3    - ip: 10.0.0.0/8
4    - ip: 172.16.0.0/12
5    - ip: 192.168.0.0/16
6  ---
7  alias: netlab-net
8  specs:
9    - ip: 123.100.22.0/27
10 ---
11 alias: ceph-storage
12 specs:
13   - &ceph
14     ip: 123.100.22.3
15 ---
16 alias: amazon-web-bucket
17 specs:
18   - &amz
19     ip: 54.231.0.0/17
20 ---
21 alias: storage-systems
22 specs:
23   - *ceph
24   - *amz
25 ---
26 traffic: web
27 specs:
28   - port:
29     - protocol: tcp
30     - number: 80
31   - port:
32     - protocol: tcp
33     - number: 443
```

Figure 3.9: Excerpt from an example alias file

extensible (in accordance with the language design goals). The alias file can be either modified manually or generated as output of a more sophisticated service (like RADIUS). Each item of the alias file is separated by three dashes (e.g. lines 6, 10, 15, etc.). Currently, there are two main components per item, namely, an *alias* (or a *traffic type*) which is the actual word that is to be used in PoLanCO statements, and a list of *specifications (specs)* that contain information regarding the low-level identifiers associated with a given alias. The code generated by the transformation function $T(w)$ maps the aliases back to the corresponding low-level identifiers. YAML allows reusing objects that were already defined in the file (avoiding duplication) by using anchors, where & is the anchor symbol. For example, the alias "storage systems" (line 21) comprises the ceph storage address (line 11-14) and a subnet of Amazon Web Services S3 buckets (line 16-19). In that way, it is possible to define an individual policy statement for transfers to different types of storage systems (as we will show in the evaluation of our exception system in Chapter 5) even though one system is located in the campus network whereas the other is a public cloud service.

The second piece of information used during the policy translation is the discovered topology. While a group of network operators could potentially draw a sketch on how nodes are deployed over the network, the approach would not only take a long time but also does not take into account dynamic events that occur in a network (e.g. a server reboot, a switch disconnect, a damaged link, or a new mobile device connecting to the wireless network). NOSs provide built-in topology discovery features that create an *initial topology sketch* comprised of OpenFlow-enabled devices, the connections between them, and the connections to attached devices discovered when the NOS inspect ARP or DHCP packets coming from the end systems.

Unfortunately, NOSs discovery features only distinguish between two types of nodes, OpenFlow switches and end systems when, as described in Chapter 1, networks are composed of many heterogeneous systems (e.g. firewalls, printers, badge readers, virtual machines, mobile devices, storage nodes, IDSs, credit card terminals, etc.). The `alias` file described above could be used to add properties to the discovered nodes. For example, hosts discovered in the IP address range 123.100.22.0/27 can be marked as "netlab" machines. In addition, operators may use traditional network management protocols (e.g. SNMP) to extract information from network devices (both OpenFlow and non-OpenFlow) and add it as properties to specific nodes in the discovered topology. Optionally, operators may also specify middlebox information if the NOS discovers the interface(s) of the middlebox as independent end systems in the initial topology sketch.

Fig. 3.10 presents an example topology stored in a graph database, specifically a *(Neo4j [116])* database. The NOS discovered the portion of the network enclosed in red. Then, say, an automated system added a "FIREWALL" label to the top node. Additionally, on the right-hand side of the figure, a portion of the non-OpenFlow network (surrounded in blue) was discovered using traditional protocols such as SNMP and LLDP, and labeled nodes for further use by the BRMS.

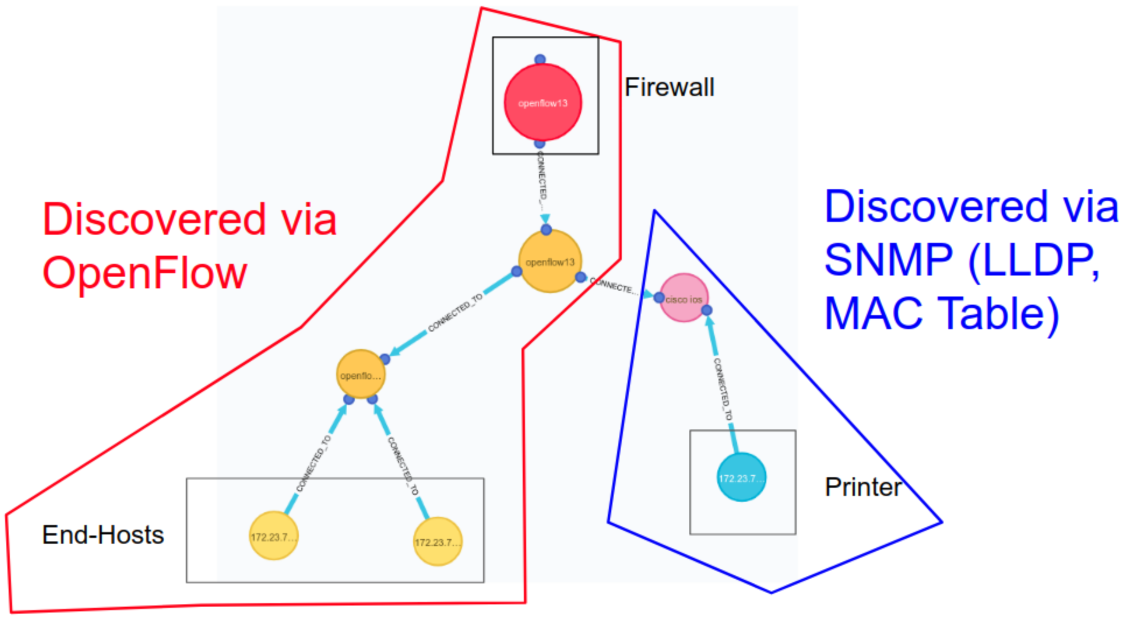


Figure 3.10: Augmenting topology information using OpenFlow and traditional protocols

3.4.2 Writing Policy Statements

Once the network information gathering is completed, network operators can write policy statements using the PoLanCO syntax shown in Fig. 3.11.

PoLanCO Rule Template

```
1 policy "policy-name"
2
3 [policy priority n]
4
5 [when
6   Node is [connected to] a device-type
7 then]
8   Action [param] [traffic-type] traffic [from end point A] [to
    ↪  end point B]
```

Figure 3.11: Syntax of the Policy Language for Campus Operations

Table 3.2: List of values for each PoLanCO token

TOKEN	**TOKEN Values**
device-type	Firewall, Web Server, Switch, Printer, etc.
Action (param)	Allow, Allow-Only, Block, Send to (Controller, IDS, HoneyPot), Mirror to (Port)
traffic-type	Web, FTP, Video, Print Jobs, etc.
end-point	netlab-network, storage-systems, authorized DNS, etc.

A policy always starts with the keyword `policy` followed by an operator-defined name assigned to it. Policies can be assigned a priority that influences the order of the policy processing by the BRMS rule engine; higher priority policies are evaluated first. PoLanCO statements are written following the syntax shown in lines 5-8. The structure resembles the way business rules are defined using the keywords `when` and `then`. In Fig. 3.11 there are four tokens that can be replaced with multiple values (see Table 3.2).

The values of the *device-type* correspond to the labels added to the nodes in the graph database during network information gathering. At present, PoLanCO supports actions that include allowing (i.e. forwarding) legitimate traffic, blocking (i.e. dropping) packets of a particular flow, sending packets of the matching flow to an external entity (specified as a parameter) for further processing, and mirroring (i.e. copying packets) to a particular port (if applicable) for out-of-band analysis.

As we can see, gathering network information is vital for the definition of PoLanCO statements. In terms of the syntax of the statements themselves, policies should be written following the conditional-body structure of conventional business rules. For policies that have to be applied in **all** the devices in the network, the conditional of the PoLanCO statement can be omitted.

Last but not least, the power of the PoLanCO syntax is that with simple combinations of words that describe the types of nodes or their relationships it is possible to identify the appropriate set of devices where network policies can be applied. We present some examples in Section 3.5.

3.4.3 Translation Layer

One of the design goals defined in Section 3.2 is to make PoLanCO agnostic to low-level mechanisms used to enforce the policies. SDN controllers can send messages using protocols like OpenFlow (and NETCONF) that facilitate the distribution of rules and device configurations to equipment built by various vendors. However, PoLanCO, as a network application, is tightly bound to the vendor-specific North-bound Interface APIs and is hampered by the lack of a standard NBI among SDN controllers.

Even though implementations of the NBI are intended to offer a common programming abstraction to network applications, the interfaces vary **widely** across NOSs (proprietary and open source); existing implementations include ad-hoc and RESTful APIs, multilevel programming interfaces, file systems, or more specialized APIs [109]. The NOS-specific designs for the NBI make it difficult for network applications (like PoLanCO) to evolve independently. Typically, applications end up tied to a specific NOS even if the application is not using any NOS-specific concepts or features, just common abstractions such as a flow, a port, the topology, etc. In reality, what makes the application portability process hard is the fact that API methods and data structures vary, as opposed to the underlying conceptual abstraction.

Consequently, with state-of-the-art NOS solutions it is not possible to perform policy translation in two NOSs software (given the same topology and same set of high-level policies) without having to re-write the BRMS application code that pushes configurations, learn new data models, APIs, and NOS-specific conventions.

To passively circumvent the network application portability problem, and instead, allow PoLanCO to enforce policies in networks that are either controlled by multiple controllers or change controller software regularly, we developed a REST API TranslaTOR (RAPTOR) for OpenFlow controllers [52]. RAPTOR serves as a single common NBI that is located amid the NOS and the BRMS components capable of pushing switch-specific and SDN-protocol specific configurations into the network.

RAPTOR unifies the network information found in controller-specific data models of multiple NOSs and puts them into one common generic data model that can be used to make PoLanCO (or any other application) portable.

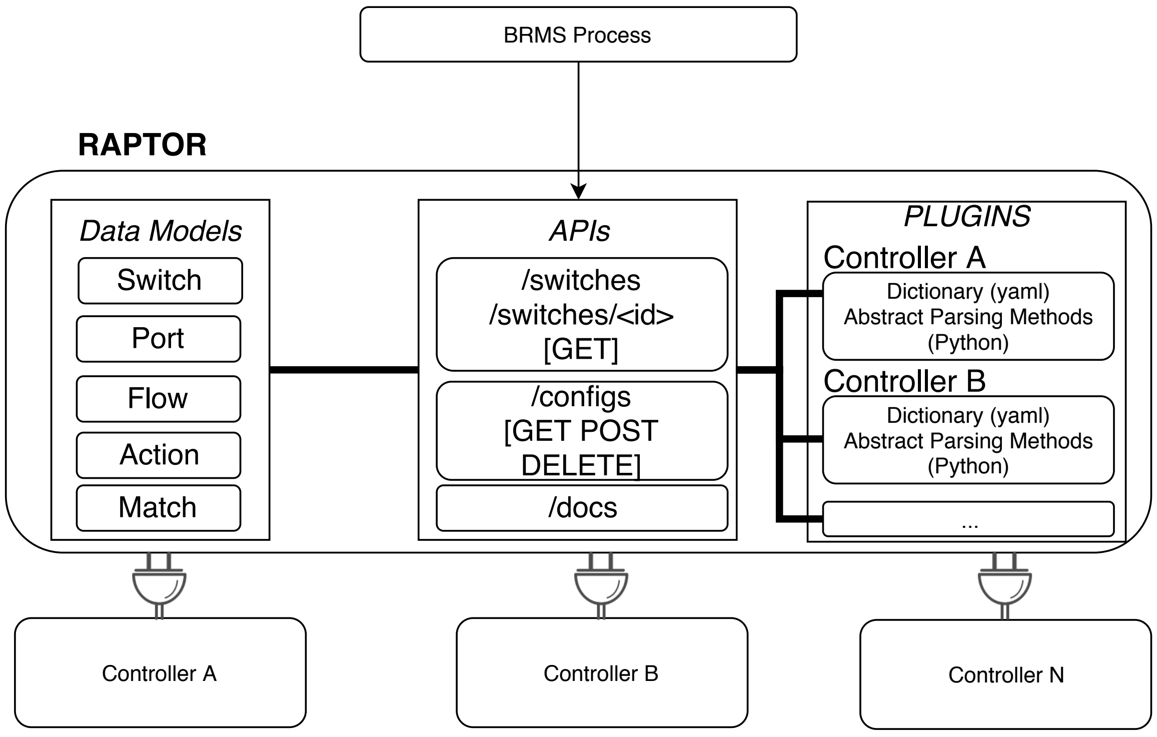


Figure 3.12: RAPTOR layer to provide language portability

Fig. 3.12 shows where RAPTOR is placed along the NBI of existing controllers to provide portability for PoLanCO (to any underlying SDN controller). RAPTOR is lightweight in the sense that it is stateless and does not prevent network applications from using NOS APIs directly. RAPTOR creates object instances on-demand every time a call is made to a controller, and returns the responses back in *JavaScript Object Notation (JSON)* format to the BRMS process. Unlike the REST API of some controllers, RAPTOR APIs follow the guidelines prescribed in the literature [61, 117] and perform various types of translations including Uniform Resource Identifiers (URIs), HTTP verbs, and data models.

Fig. 3.12 also shows the internal building blocks of RAPTOR. First, RAPTOR defines a set of common abstractions that includes switches, ports, flow entries, actions, flow match, etc. The data models representing the abstractions are common across all components of RAPTOR and the backend service of the BRMS that generates the low-level configurations. Second, RAPTOR also defines a list of APIs (Uniform Resource Locators (URLs)) that high-level applications (e.g. the BRMS component that pushes configurations) access via REST calls in spite of the NOSs controlling network devices. Lastly, each controller supported by RAPTOR is included as a "plugin" composed of two elements, namely, a dictionary (in YAML [115] format) of equivalences between RAPTOR and the controller's conventions (i.e. field names and values), and a file where *abstract* methods ought to be implemented in order to adhere to the translation layer.

3.5 Examples

This section presents examples of how high-level imprecise policies found on several academic institutions can be translated into simplified, human-readable statements using PoLanCO. All the examples described below would need an associated alias file that defines the port numbers of certain types of traffic (e.g. DHCP, DNS, FTP, or

HTTP traffic) as well as IP addresses of known end systems (e.g. printers, DHCP servers). However, to keep the examples short, the alias file is generally not shown. In most cases, the file is rather straightforward to define (see Fig. 3.9 for an example alias file).

The following examples show various types of campus network security policies. When possible, excerpts of the high-level (vague) policy found on the university websites are displayed along with the PoLanCO statements written to represent the high-level policies.

3.5.1 Disabling Insecure Protocols

Recall the example presented in Section 3.3 that introduced a *clear-text prohibition policy* commonly found in university websites:

```
1  Applications which transmit sensitive information over the
2  network in clear text, such as telnet and ftp, are prohibited
3  and will be blocked
```

The relevant information needed to write PoLanCO statements appears in lines 2 and 3 where types of traffic in the form of protocols (i.e. telnet, FTP) and the corresponding action (i.e. prohibit, block) are specified. The PoLanCO statements derived from the contents of the policy and the alias file generated to write the statements are shown in Fig. 3.13.

The conditional part of the two PoLanCO statements is omitted because all network devices on the campus network need to enforce the policy. Fig. 3.14 shows an example topology where the policy is enforced at various places, namely, a firewall, a router, and two switches. The definitions in the alias file cause the BRMS to generate OpenFlow rules that drop incoming packets from any interface whose destination is the either of the FTP control or data ports (20 and 21) and the telnet port (23).

PoLanCO Statements

```
Block ftp traffic
Block telnet traffic
```

insecure-protocols.alias

```
traffic: ftp
specs:
  - port:
    - protocol: tcp
    - number: 21
  - port:
    - protocol: tcp
    - number: 20
---
traffic: telnet
specs:
  - port:
     - protocol: tcp
     - number: 23
---
```

Figure 3.13: PoLanCO statements and alias file that implement a clear-text prohibition policy

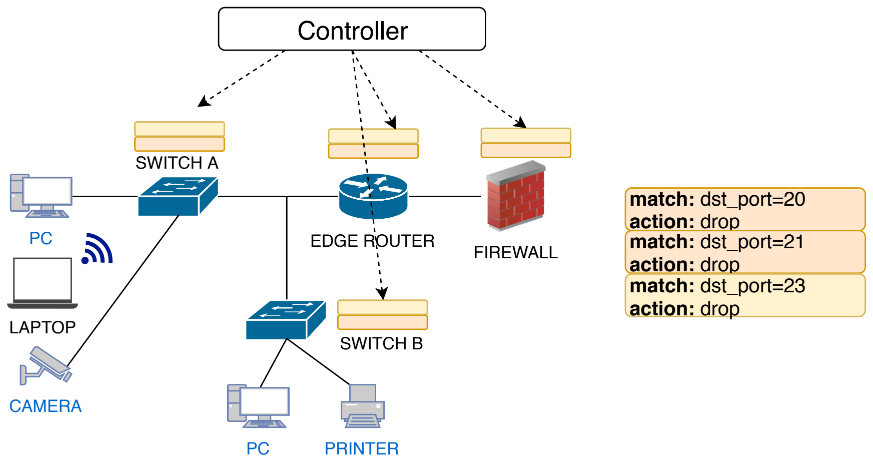


Figure 3.14: Enforcing a campus-wide policy that disables insecure protocols

The network configurations shown are in the form of OpenFlow version 1.3 rules. However, during the network information gathering phase, where every topology node is added as an object to the working memory of the BRMS, each type of device can be assigned a mechanism for policy enforcement (e.g. any version of OpenFlow, NETCONF/Yang, iptables, remote SSH commands, etc).

3.5.2 Securing Network Printers

Most printers come with a default configuration that allows users to start using them out-of-the-box once the printer is plugged in to the network. Though out-of-the-box functionality is appropriate for a home network, carelessly plugging printers in to an enterprise network poses various risks because multiple unnecessary services are enabled, an easy-to-guess administrative password is set by default, and printers can be accessed from outside the network if they get a public IP assigned by mistake. If compromised, a network printer could be used to steal data (e.g. contents of print jobs), attack other systems in the network (e.g. DoS) or print an excessive amount of spam messages causing waste in resources.

Some universities have published policies on how to secure network printers. Fig. 3.15 shows a printer policy found on the University of California–Berkeley's website [118].

```
To secure your printers from unauthorized access, print
configuration alterations, eavesdropping, and device compromise
follow these printer security best practices:

- Campus printers should not be exposed to the public Internet.
- Use encrypted connections when accessing the printers
↪  administrative control panel.
- Do not run unnecessary services.
```

Figure 3.15: Printer policy of the University of California–Berkeley

The statements in the policy are not technically precise. In fact, the statements are guidelines and not *mandatory* policies. However, in order to protect network printers and avoid the security risks entailed due to misconfiguration, network operators can write PoLanCO statements that enforce the practices outlined in the guidelines.

For example, the guideline in line 5 could be addressed in two ways. First, if every end system with a publicly reachable IP (e.g. 128.163.4.50) is labeled such that it can be distinguished from systems with private IP addresses, then the assigned label could be included in PoLanCO statements that block traffic to/from a misconfigured printer. Second, since some network printers are configured with an IPv6 address that is globally unique, there is a potential risk for the printers to be reachable from the Internet. Given that normally SDN controllers cannot push configurations to end systems, the network operator could write a PoLanCO statement that blocks IPv6 traffic destined to a printer at the closest network device where the policy can be enforced.

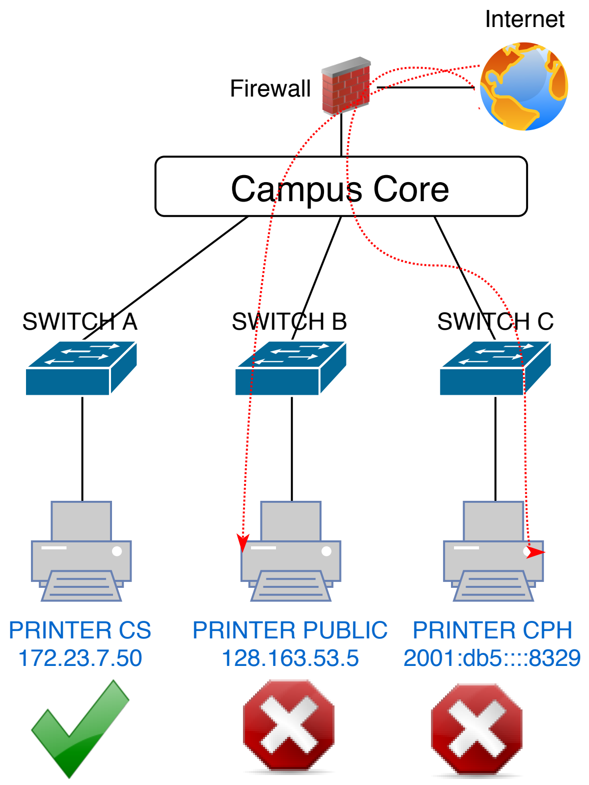


Figure 3.16: Three printers in the network with their IP assignments

Fig. 3.16 shows a topology where three printers were (mis)configured and labeled during the topology discovery phase. Naturally, the common label across all the printers is `PRINTER`. However, note that the printer attached to `SWITCH B` was also marked with the label PUBLIC as it was assigned an IP address in the 128.163.0.0/16 public network. While the printer attached to SWITCH C was labeled with CPH (the College of Public Health), the printer is still misconfigured and susceptible to being accessed from the Internet via IPv6. Fig. 3.17 shows the network statements that would enforce the printer security guidelines.

```
1 when node is connected to a PUBLIC PRINTER
2 then block all traffic
3
4 when node is connected to a PRINTER
5 then block ipv6 traffic
```

Figure 3.17: PoLanCO statements securing printers from external access

The first statement (lines 1-2) enforces the policy only at SWITCH B because it is the only network device that is connected to a node with the labels PRINTER and PUBLIC. The generated OpenFlow rule uses the printer's IP address (128.163.53.5) to block all traffic coming in and going out of the public printer. Similarly, for the second statement (lines 4-5), the policy is enforced at switches A, B, C because they are all connected to a node with the label PRINTER. In this case, a generic OpenFlow rule blocking IPv6 traffic to the printers enforces the policy.

The rest of the guidelines such as disabling unnecessary services are similar to the previous example (Section 3.5.1) where insecure protocols are disabled campus-wide. For the printer example, the conditional part would not be omitted and instead should identify only the nodes that have a printer connected to them. In addition, the body part of the statement should block what the network operator consider as an "unnecessary" service (e.g. HTTP, FTP, SNMP, etc.) from a printer management perspective.

3.5.3 Firewall for External Connections

Firewalls are often the first line of defense of any network, including campus networks. It is not uncommon to see policies and guidelines for network traffic that is destined to/from the Internet. Consider an excerpt from a policy involving the perimeter firewall at the University of Missouri-St. Louis [113] shown in Fig. 3.18.

```
1  All UMSL network traffic to and from the Internet must go through
 ↪  the firewall.
2  Any network traffic going around the firewall must be accounted
 ↪  for and explicitly allowed by the Computer Security Incident
 ↪  Response Team (CSIRT).
```

Figure 3.18: UMSL Firewall Policy

Enforcing the core of the policy (line 1) is straightforward in PoLanCO. Assume the topology discovered during the network information gathering is the one shown in Fig. 3.19. There are switches *inside* the network that send traffic out of the network and switches *outside* the campus network that forward data into the network.

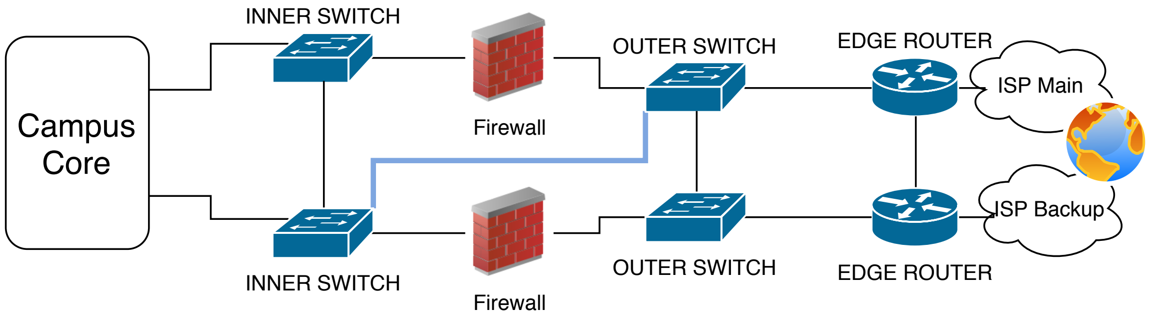


Figure 3.19: Example topology discovered at the edge of a campus network

The PoLanCO statements that enforce the policy are shown in Fig. 3.20

Both PoLanCO statements would subsequently be translated into the appropriate network configurations for both `INNER` and `OUTER` switches. The translation process will identify information such as the designated interface where packets must be forwarded, and the IP addresses the rules need to match on (e.g. the campus network is 134.124.0.0/16). Moreover, the blue link connecting the upper `OUTER SWITCH` with the lower `INNER SWITCH` offers an alternative path to traditional routing protocols (e.g. OSPF, BGP) that bypasses the firewalls. The PoLanCO statements force all traffic to avoid the alternative route and appropriately send all traffic through the firewall.

However, note that network operators can use a *policy priority* in the PoLanCO statements to explicitly allow exceptions to the policy and allow the usage of the path that bypasses the firewall. Likewise, the rest of the policy (line 2 in Fig. 3.18) that discusses the allowance of exceptions to the policy (i.e. bypassing the firewall's inspection) could be enforced via an exception system such as the one described in Chapter 5.

3.5.4 Rogue Servers

PoLanCO can enforce policies that forbid the deployment of rogue servers—a system that is providing services to the network that IT staff is not managing. Take for example the policy found at the Oberlin College and Conservatory [119] shown in Fig. 3.21.

```
when node is an INNER SWITCH
then send to PERIMETER FIREWALL traffic from campus network to
↪  Internet

when node is an OUTER SWITCH
then send to PERIMETER FIREWALL traffic from Internet to campus
↪  network
```

Figure 3.20: PoLanCO statements enforcing a firewall policy

```
In no case shall the following types of servers (except those
maintained by CIT for the express purposes delineated) be
connected to the network: DNS, DHCP, BOOTP, or any other server
that manages network addresses.
```

Figure 3.21: Network-based Intrusion Prevention Policy of the Oberlin College and Conservatory

The key to enforce the policy is to distinguish between authorized servers and regular hosts use two labels. By distinguishing servers from hosts it is possible to block DNS and DHCP traffic destined to the latter. Note that it does not suffice to solely block all of DNS and DHCP packets to enforce the policy because legitimate end systems would be unable to resolve names or request a private IP addresses from authorized DNS and DHCP servers. Instead, the BRMS should produce configurations that only allow responses issued by authorized servers and block messages issued by any other device (i.e. a rogue server). The PoLanCO statements are presented in Fig. 3.22.

```
1 when node is connected to a REGULAR-HOST
2 then allow-only dns-response traffic from authorized dns server
3
4 when node is connected to a REGULAR-HOST
5 then allow-only dhcp-response traffic from authorized dhcp server
```

Figure 3.22: PoLanCO statements prohibiting traffic from rogue servers

Fig. 3.23 shows the translation of PoLanCO statements into OpenFlow rules. First, the BRMS selects all the network devices that are connected to a `REGULAR-HOST` node (i.e. `SWITCH A` and `SWITCH C`). Then, the `allow-only` action of the PoLanCO statements (lines 1 and 4) produces two OpenFlow rules per selected switch. For clarity, Fig. 3.23 only shows the rules enforcing the policy on (rogue) DNS servers but the procedure is similar for DHCP servers. Specifically, a rule with priority n drops *all* DNS response traffic, thereby preventing messages originating from rogue servers from reaching end systems; and another rule (with higher priority, say, $n+1$) that explicitly allows response traffic coming from authorized servers to reach end systems for legitimate name resolutions.

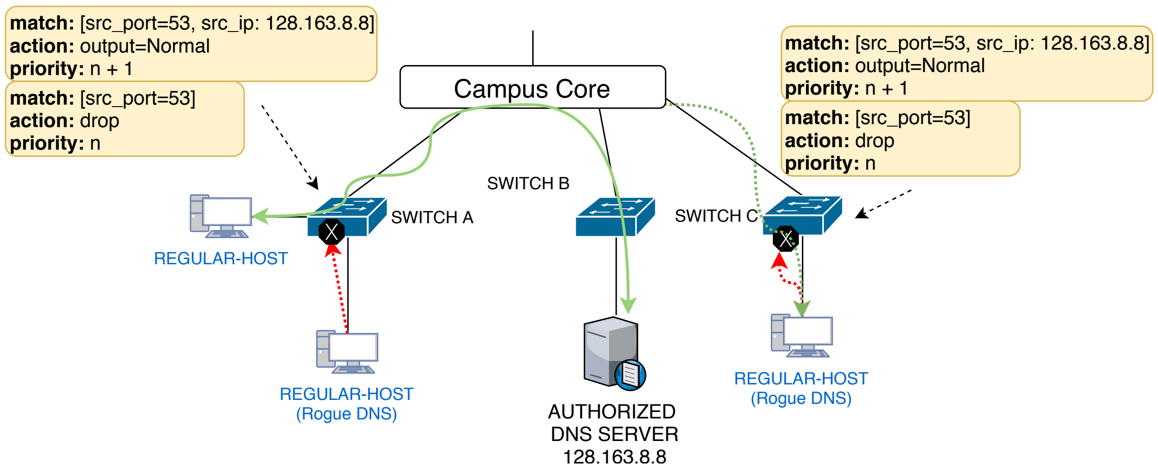


Figure 3.23: Rules installed to prevent rogue servers

3.5.5 Other Policy Statements

We provided a detailed description of some of the policy statements found at various university websites that can be written in PoLanCO in order to document, automate, and trace network policies. This section presents other statements that were not found in online policy documents but are typically enforced in a campus network.

Accessing SSH Servers: Besides VPNs, certain nodes in the campus network might be allowed to be accessed from the Internet over an encrypted SSH channel (possibly from specific IP ranges). Access to authorized SSH servers can be explicitly specified using PoLanCO (see Fig. 3.24)

Deep-Packet Inspection: Even though so-called "Next-generation Firewalls" typically perform DPI, they are sometimes cost prohibitive for institutions that

```
when node is a FIREWALL
then allow ssh traffic to campus network ssh servers
```

Figure 3.24: PoLanCO statements for access to SSH servers

```
when node is a FIREWALL
then send to IDS web traffic from Internet to authorized web
↪  servers
```

Figure 3.25: PoLanCO statements enforcing IDS inspection

have tight budgets for network infrastructure. As an alternative, dedicated software-based IDSs (e.g. Snort [30], Bro [29]) are deployed at multiple places in the network on commodity hardware that perform the DPI at slower speeds but with a high-accuracy of detecting intrusions.

Fig. 3.25 shows a PoLanCO statement that sends traffic aimed at web servers to an IDS once it has successfully pass through the campus firewall.

Offline Traffic Monitoring: There are occasions that, instead of doing inline DPI that may become performance bottlenecks, network operators do offline analysis of the network traffic traversing various portions of the network. For example, network traffic is analyzed for capacity planning, usage trends, or outage detection. Network operators may specify PoLanCO statements to copy network packets out to a specific port. Fig. 3.26 show an example to analyze local DNS traffic:

```
when node is connected to an AUTHORIZED DNS SERVER
then mirror to port 0 dns traffic from campus network
```

Figure 3.26: PoLanCO statements copying DNS traffic to a switch port

Locking-Down Access to Printers: Section 3.5.2 described an example on how to enforce policies that prevent printers from being accessible from the Internet. The policy also required network operators to disable any unnecessary services that could be potentially active by default. However, the policy did not prevent *any* network user from sending print jobs to *any* printer on campus. In reality, campuses typically impose tight restrictions on access to network printers (and other types of devices such as cameras, IP telephones, streaming systems, etc.). Access to printers is normally limited to members of the department where the printer is physically placed. For example, the PoLanCO statements shown in Fig. 3.27 that any printer in the Computer Science network may only be accessed by members of the Computer Science departments using desktops (i.e. no laptop, or mobile printing), blocking everything else. In short, two PoLanCO statements are needed: (1) drop any incoming requests from a device that is not part of the department's network and (2) drop "print" requests coming from the Computer Science network but are not desktops (e.g. a laptop over wireless).

```
when node is CS FIREWALL
then block traffic to cs printers

when node is connected to a CS PRINTER
then allow-only print-jobs from cs desktops
```

Figure 3.27: PoLanCO statements locking down access to printers

Interactions in Emerging HPC Environments: Historically, HPC systems have been governed by relatively simple security policies based on a combination of authentication, VPN and firewall technologies. However, in recent years local cloud environments (e.g. OpenStack [120]) and distributed storage systems (e.g. Ceph [121]) have emerged to complement the traditional offer of large supercomputer clusters provided by HPC centers. These emerging systems are oftentimes formed with separate components that are connected either through a single shared network or via multiple dedicated networks that maximize system performance and scalability to their users. The architecture of two of those systems is briefly described. Namely, an OpenStack deployment and a Ceph storage system. Then, the types of access policies that are needed are discussed. Specifically, policies that define and restrict valid access within the systems' components, and between the systems and the network users.

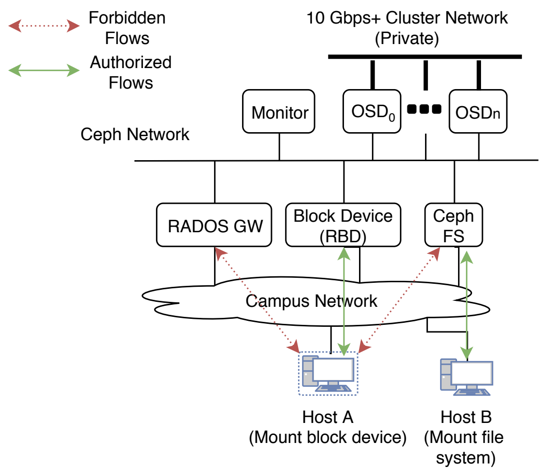
Ceph: In a Ceph environment, the main distributed resources are called Object Storage Daemons (OSDs) and Object Storage Monitors (OSMs). The former are responsible for storing objects and perform replication and recovery tasks, whereas the latter manage cluster membership and state. The resources can be interconnected in many ways, for example, they could be connected to a general-purpose network (like the campus network), or they could be deployed on an additional network (e.g. using VLANs to create a "Ceph network"). Ceph resources rely on three interfaces (e.g. the RADOS gateway, block device interface, or Ceph's file system) that act as proxies that translate requests into Ceph-protocol messages from network users to store/retrieve data from/to the campus network (see Fig. 3.28a). In either case, network operators need to establish appropriate policies to prohibit unauthorized access from hosts on the campus network to Ceph's underlying resources and web interfaces. For example, the network policy should only allow authorized CephFS clients to reach CephFS server, only authorized hosts to reach block devices (i.e. devices that can be mounted), or ensure that the cluster network only moves traffic across OSDs. The policies can be implemented in the network using SDN, and therefore, can be expressed using PoLanCO statements.

OpenStack: Like Ceph, OpenStack can be deployed either sharing the general purpose network or by using a private management network used to provision and control OpenStack components. In the former, any machine on campus would be able to reach/attack any component of the system, potentially including VMs deployed and managed by other users. In the latter (see Fig. 3.28b), network separation can help by making key components of the system only accessible via the management network. In addition, it is also possible to create independent external "provider" networks (i.e. paths to the Internet) that ensure the traffic from VMs is isolated to an appropriate VLAN. While the deployment of various physical (or virtual) networks helps with OpenStack's network security needs via isolation, the approach falls short when it comes to the implementation of fine-grained network security policies required to allow/prohibit interactions (e.g. systems, communication mechanisms) between (and within) components in the management network (e.g. user A's VM can be accessed from the Internet via Remote Desktop Protocol (RDP)) and network users.

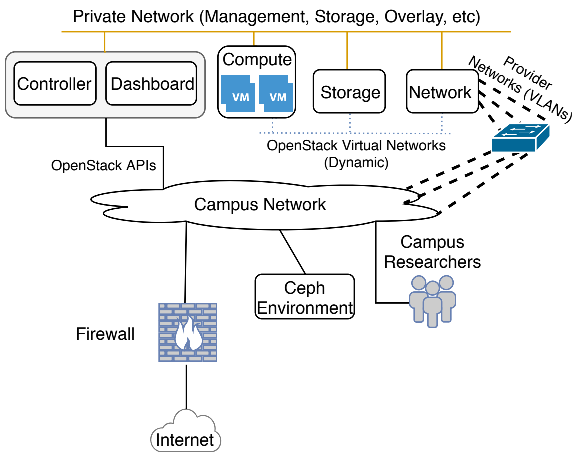
Dynamic environments such as Ceph and OpenStack have the potential to experience a wide variety of network interactions within and between both systems. Fig. 3.29 presents some of the policies that can be written in PoLanCO that could ease the management of such interactions and ensure that components are accessed by specific entities.

3.6 Final Remarks

We presented a translatable policy language called PoLanCO that allows network operators to specify network policies in terms of human-readable, but technically precise, statements when compared to the long acceptable use policies available online on several university websites. The statements are written in an easy-to-read syntax



(a) Ceph



(b) OpenStack

Figure 3.28: Ceph and OpenStack architectures

```
when node is an OSD
then allow-only traffic from RADOS Servers

when node is an OSM
then allow-only traffic from RADOS Servers

when node is an OpenStack VM
then allow-only vnc traffic from campus addresses

when node is a firewall
then allow ssh traffic to openstack-network
```

Figure 3.29: PoLanCO statements expressing access policies in OpenStack and Ceph systems

that leverages advancements in technologies such as BRMS and SDN. PoLanCO uses a network information gathering phase to appropriately label network nodes and utilizes the information as facts (i.e. source of truth) to enforce various types of network policies. We showed several real-world example university policies, describing how they can be written with simple PoLanCO statements, and how the statements are translated into OpenFlow rules to ensure policy enforcement.

Chapter 4. Network Security Caps: Separating Policy and Device Configuration

4.1 Introduction

The traditional way of enforcing network security policies in current networks is largely done by means of device *configurations*. Because network policies are often needed to protect network servers from attack or to protect communication to/from a network server (e.g., prohibiting cleartext protocols like HTTP to web servers or FTP to file servers), network policies often end up being enforced by servers via their configuration files, rather than by the network.

This approach towards network security enforcement should not come as a surprise given that most servers already have a configuration file. While the configuration file is primarily there to define or specify the functionality the server should offer, it is relatively easy to add a few statements to the configuration file that also specify security policies. Besides, who better to protect a server than the server itself.

Enforcing network policies via configuration files conflates network security enforcement with server functionality configuration, and it comes with some drawbacks.

Independent Server Configuration: Every server must be configured (and secured) independently. Therefore, securing the network via configurations requires more work (i.e. it is challenging to scale and maintain independent servers) and increases the chances for a server to be secured incorrectly (on accident). On the other hand, doing security enforcement in the network only requires securing the network (forwarding devices) and applies the same policy to all servers.

Users as Server Administrators: With personal equipment and user-managed VMs deployment being commonplace in today's campus networks, network users cannot be trusted to enforce security policies on end systems (including servers). For example, students in charge of configuring their laptops may not be concerned about security when joining the network, students may also prefer to use default configuration files to have services available out-of-the-box, or students may not want to maintain their systems regularly. Having network administrators be responsible of network security centralizes trust where it should be – with a trained professional who cares.

Functionality and Policy: Because the configuration file is used to specify both functionality and security, it is easy to enable (or disable) features that create security vulnerabilities. For example, during the initial setup of a server, the owner could forget to disable a feature that violates the network policy (e.g. removing the HTTP port from the listening ports in web server configuration file shown in Fig. 4.1). Whereas, network administrators think about security independently of the current functionality that has been enabled (or disabled). In other words, network administrators plan for all attacks, regardless of whether the server functionality is susceptible to them or not.

Mitigating Attacks at the Server: If network security is enforced *only* at the server, then attacks can only be stopped or blocked when packets reach the server. This enables DoS or DDoS attacks that not only affect the performance of the server, but may also affect the performance of other services on the network. By securing the network, attacks can often be blocked far before reaching the server itself.

Emerging network architectures such as SDN have enabled new opportunities for the development of applications that could aid and ease network security man-

httpd.conf (Apache)

```
Listen 80
ServerName localhost:80
ServerRoot /var/www/example.com

<Directory />
    Options FollowSymLinks
    AllowOverride None
    Order allow,deny
    allow from all
</Directory>

```

Figure 4.1: Example configuration file of an Apache web server

agement (Section 2.3.4). As an example of potential uses of SDN technology to assist with security, we introduce and describe the concept of *Network Security Caps*. The main idea behind Network Security Caps is to use OpenFlow-enabled devices at multiple places in the campus network to separate the enforcement of security policies from configuration files. Specifically, the separation is done by adding the ability to intercept traffic going to or coming from end systems and apply policy to the network packets of those traffic flows before they arrive at the servers that need to be protected. Under this approach, we add a security layer to the network that *protects servers from violating security policies regardless of server misconfigurations.* Rather than replacing the way network policies are enforced today, we propose the security layer as an additional control to minimize potential vulnerabilities in the network and can be automated by using DSLs such as PoLanCO (described in Chapter 3).

4.2 Network Security Cap

Traditional networks rely heavily on configuration files and CLI commands to set up network services and devices (switches/routers) in terms of the functionality they provide and the way they should enforce high-level network polices defined by the organization. However, conflating functionality and policy enforcement poses a security risk for the network.

Take for example the network device shown in Fig. 4.2. The network device has multiple interfaces that are used to communicate with other devices and end systems in the network. Once a network device has been successfully installed in the network, it needs to be appropriately configured before any interface is actually enabled to forward packets. Network operators configure network devices via CLI instructions specifying operational data such as the device's name, availability of SNMP Management Information Bases (MIBs), the network domain the device belongs to, initial entries in the routing table (if providing L3[1] capabilities), administrative credentials for future management and maintenance task, logging directives, VLAN association per port, to name a few. After the operator configures a device, whenever traffic arrives at an interface, the device's internal configuration (referred to as *normal pipeline*) modifies and forwards network packets based on the current state of the device's internal data structures. The contents of the internal data structures change over time depending on the type of device, the protocols that were enabled per interface, and network information received from neighboring devices.

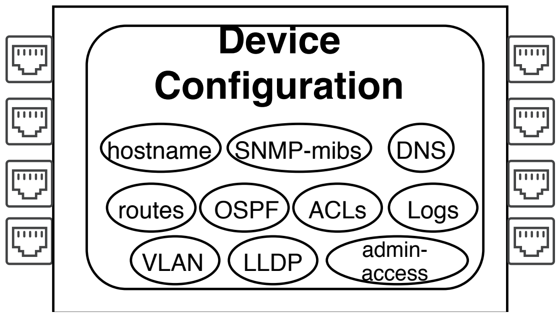


Figure 4.2: A generic network device

Moreover, any modification to the high-level network policy requires the network operator to change the internal configuration of the device (e.g. updating VLANs or default routes for certain destinations). Additionally, for policies that

[1]L3: Refers to the network layer of the OSI model [122]

are solely implemented with configuration files or host-based utilities, there is no mechanism in place to enforce the new policy on the network while the end system is appropriately tuned and its configuration files updated to comply with the new policy directives.

In order to aid with the timely enforcement of policies in the network, even if they depend on the configuration of end-systems, we introduce the concept of a *Network Security Cap (NSC)* enforcement layer. A NSC (Fig. 4.3) is an intermediate security policy enforcement layer located inside network devices (switches/routers) between the device interfaces—in charge of receiving and transmitting network traffic—and the device internal configuration—used by the device's data plane to process packets and make decisions based on information inside the packet headers and internal data structures. The NSC layer intercepts all packets arriving at a device **prior** to any packet reaching the *normal pipeline*. Then, policy enforcement rules evaluate packets and determine whether a packet should be (1) dropped if it is not policy compliant (e.g. when traffic that uses protocols that transmit information in clear text), (2) forwarded out (and possibly modified) off an specific device interface (e.g. when traffic is part of an authorized exception), or (3) sent to the *normal pipeline* if the packet does not violate the policy (e.g. when hosts are resolving addresses via ARP messages).

A NSC is initially empty, thereby relying entirely on the policies enforced at the end systems and the network devices as it happens today. As a result, NSC can be incrementally deployed as *enforcement points* that initially do not interfere with ongoing operations in the network. Nevertheless, a switch's NSC also allows the dynamic addition of enforcement rules. Rules can be of any type (e.g. OpenFlow, Policy-Based Routing (PBR) [123]) as long as they precede the device's *normal pipeline* processing. As time goes by and policy changes, multiple security policies can be enforced across portions of the network that have devices with NSCs.

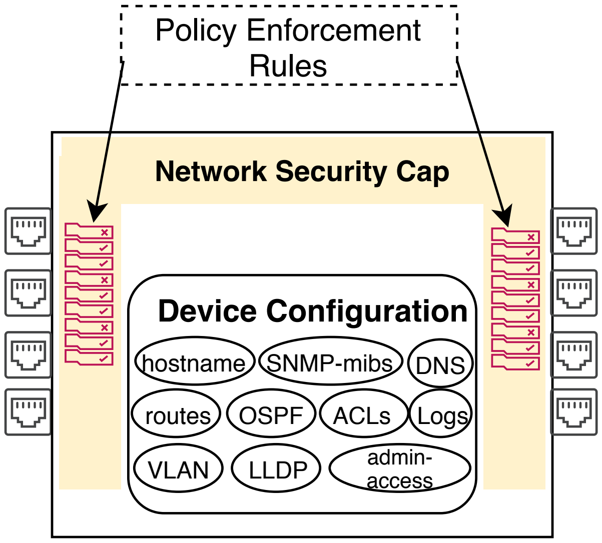


Figure 4.3: The Network Security Cap

Network policy definition tools can leverage NSC-capable nodes to deploy network-wide policies. For example, an operator writing network policies in a DSL such as PoLanCO (described in Chapter 3) can generate various types of network configurations and push them to NSC-capable devices for enforcement purposes. In that way, any arriving packet at a device will be first evaluated against the policy before it can be handled by the regular processing of the network device (if at all).

4.3 Deploying Enforcement Points in the Network

Network Security Caps can follow an incremental deployment approach that does not disrupt ongoing communications. A *Network Security Cap* treats security enforcement as an added service (as opposed to a security replacement) to the infrastructure. As mentioned earlier, every deployed SDN device could be seen as a policy enforcement point that protects a portion of the network from potential exploits arising from system misconfigurations.

In order to achieve a seamless transition towards an NSC-enabled infrastructure, OpenFlow powered switches were utilized to realize the NSC enforcement layer. Specifically, the OpenFlow pipeline is used as the NSC layer where decisions about policy compliance are made on a per-flow basis using OpenFlow actions (e.g. dropping, modifying, forwarding, mirroring, rate-limiting) before the internal device configuration does any further processing.

Unfortunately, the OpenFlow specification [64] has many features marked as *optional*. Vendors may choose what parts of the protocol to implement in their hardware, and consequently, make it harder to deploy SDN-enabled equipment in networks due to the lack of standardization across many OpenFlow implementations.

Nevertheless, in general, there are two types of OpenFlow switches, namely, OpenFlow-only and OpenFlow-hybrid. The former, which is typically used in prototypes and systems reviewed in parts of Section 2.3, are considered “dumb devices” that do not make any local decisions beyond those based on the rules present in the flow tables. The latter are devices that support both OpenFlow operation ***and*** traditional forwarding that involves and relies on traditional protocols to perform L2 [2] Ethernet switching and L3 routing. Naturally, OpenFlow-hybrid devices help realize *Network Security Caps* in a campus network. They ensure that existing communications are not disrupted and an incremental approach is feasible. The deployment OpenFlow-hybrid devices builds a campus-wide NSC enforcement layer that is viewed as a an added security service to the network’s standard forwarding capabilities. In addition to the operation in hybrid mode, the OpenFlow implementation embedded in the switches must support the reserved port `NORMAL` that, unfortunately, is marked as *optional* in the OpenFlow specification. The `NORMAL` port allows a switch to process all packets received in any interface to go through the OpenFlow pipeline **before** the normal pipeline (and internal configurations) takes over.

Given an OpenFlow-hybrid device that supports the `NORMAL` port, a new mode of packet processing that differs from the reactive and proactive modes (see Section 2.3.3) can be introduced in our SDN-enabled network . Unlike the reactive and

[2]L2: Refers to the data link layer of the OSI model [122]

proactive packet processing approaches, which are oriented towards networks with OpenFlow-only devices and where the default policy is to either send packets to the controller or simply drop them, the default policy using NSCs is to send all policy-compliant traffic to the `NORMAL` pipeline, thereby way guaranteeing no disruption of communications. Fig. 4.4 shows how packets are processed when they reach a newly deployed NSC at the access layer of the topology. The OpenFlow rule that matches on all types of packets and forwards them to the `NORMAL` port (i.e. to the normal pipeline) represent the default policy. Consider an HTTP request that is sent from another system to the server on the left-hand side of the figure (green arrows). First, the NSC process the packet matching it against the default policy since no other policies are installed in the switch. Then, the packet is sent to the normal pipeline. The device configuration determines the output interface for the packet based on the device's internal data structures (e.g. MAC table, routing table). Lastly, the packet is delivered to the the web server process running on the server.

Note that a NSC does not affect packets that are used for route and topology discovery using traditional distributed protocols (blue arrows). The only difference in this case is that the packet processing ends at the normal pipeline with updates to the device configuration.

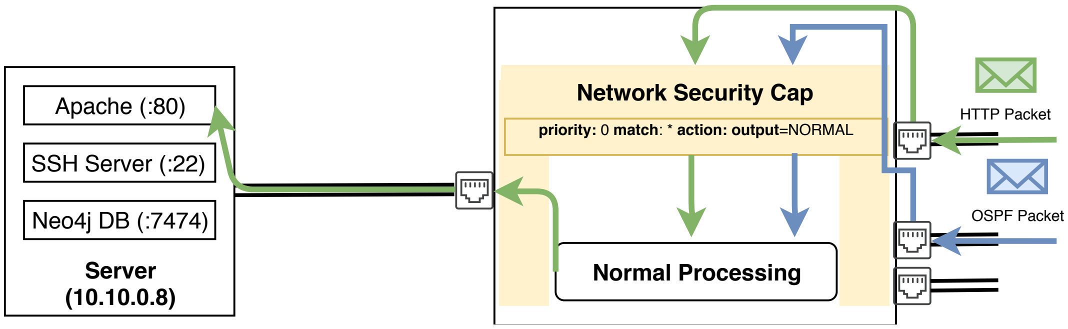


Figure 4.4: Normal processing of packets at the access layer

Although NSC-compliant devices can be deployed at any place in the network, we argue that the access layer is an ideal place for policy enforcement due to its proximity to the end systems (i.e. servers and clients).

4.4 Example Policy Enforcement

All of the examples that were presented in Section 3.5 leverage the NSC concept described in this chapter as a way to translate human-readable policies into policy enforcement rules. The examples were mostly focused on the expressiveness of PoLanCO and therefore addressed an individual policy (possibly spread out onto multiple devices) at a time. For the sake of completeness, the example in this section addresses an scenario where multiple policies can coexist in an individual NSC and can help mitigating the risks of rather frequent server misconfigurations.

Continuing with the interactions described in the previous section, Fig. 4.6 shows a NSC that receives various types of traffic and enforces multiple high-level policies that were specified using the PoLanCO statements shown in Fig. 4.5.

```
policy "disable telnet"
Block telnet traffic

policy "disable clear text web server access"
when node is connected to a WEB-SERVER
then block http traffic

policy "secure web server access"
when node is connected to a WEB-SERVER
then allow-only https traffic

policy "1 hour backup exception"
policy priority 10000
when node is connected to a SERVER A
then send to BACKUP-PATH traffic from server-a to aws for 60
↪ minutes
```

Figure 4.5: Example PoLanCO statements enforced by a Network Security Cap

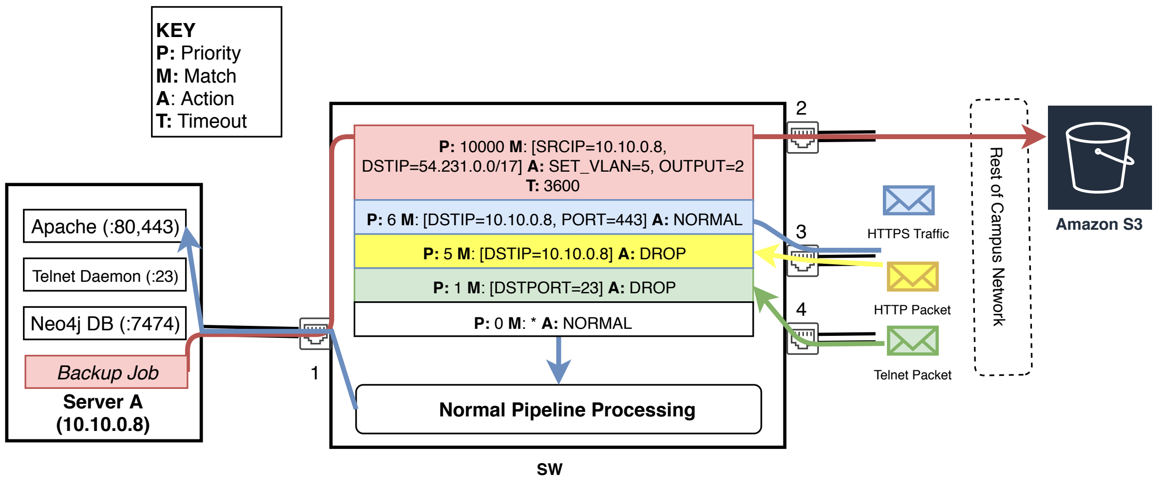

KEY
P: Priority
M: Match
A: Action
T: Timeout
Apache (:80,443)
Telnet Daemon (:23)
Neo4j DB (:7474)
Backup Job
Server A
(10.10.0.8)
P: 10000 M: [SRCIP=10.10.0.8, DSTIP=54.231.0.0/17] A: SET_VLAN=5, OUTPUT=2 T: 3600
P: 6 M: [DSTIP=10.10.0.8, PORT=443] A: NORMAL
P: 5 M: [DSTIP=10.10.0.8] A: DROP
P: 1 M: [DSTPORT=23] A: DROP
P: 0 M: * A: NORMAL
Normal Pipeline Processing
SW
1
2
3
4
HTTPS Traffic
HTTP Packet
Telnet Packet
Rest of Campus Network
Amazon S3

Figure 4.6: A Network Security Cap enforcing high-level policies

On the left hand side, the box representing *Server A* shows the applications (services) that are currently running on the server.

The *Server A*'s primary purpose is to host a website. The system administrator installed the Apache server on the system and changed its configuration file to add HTTP over TLS (HTTPS) support because (1) HTTPS is disabled by default in the software version she downloaded and (2) she is aware that web services should be reached via secure protocols. However, the administrator forgot to disable HTTP support in the configuration file. Therefore, there is a security risk for connections trying to contact the server reaching on that port (many). Additionally, *Server A* has a `telnet` daemon that (unknowingly) was turned on by the OS during its initial setup. Lastly, the server is also running a database server that uses port 7474 to provide a Graphical User Interface (GUI) (Fig. 3.10 and Fig. 5.5 are examples of such interface) for running queries and executing database transactions. Access to this GUI should be local (i.e. restricted to the localhost address 127.0.0.1), but, by default, it is turned on and available for access from other addresses in the network (i.e. the database service is misconfigured).

Let us consider the following network security policies that the server should have implemented but because of misconfiguration is currently violating. First, protocols that transmit information in clear text such as `telnet` should be disabled (lines 1-2 in Fig. 4.5). At present, state-of-the-art OSs disable `telnet` by default. However, nothing prevents a system administrator to misconfigure the server (or use an old OS launches a `telnet` daemon by default) and accidentally enable the service. The PoLanCO statement enforces the policy generating the (green) NSC rule that prevents `telnet` requests to succeed even if the server has the service turned on. Secondly, consider a policy stating that all web servers on campus should only be accessed over a secure HTTPS connection (i.e. port 443). Since the system administrator's intention was to enforce the policy via configuration, she should have disabled the HTTP feature (i.e. port 80) during server setup. However, if network operators use PoLanCO and NSC to implement the policy (regardless of the the system administrator), NSC rules that either drop HTTP (lines 4-6) or allow only HTTPS packets (lines 8-10) can be installed in all network devices that have a web server connected (the yellow and blue rules in Fig. 4.6 implement the latter). Now, since in this example the system administrator forgot to disable port 80 in the configuration file, the network protects *Server A* because the NSC separated policy implementation/enforcement from server configuration.

Unfortunately, OpenFlow does not support negations to the traffic match portion of a rule (e.g. matching on all traffic that is not port 80). However, installing two (or more) rules in the NSC circumvents the limitation. Specifically, a rule with priority P that drops all traffic to a particular node (e.g. using its IP address) and multiple finer-grained rules with priority $P + 1$ that include the services and sources that can access the node. In Fig. 4.6, the yellow rule forbids any request to reach *server A* whereas the fine-grained blue rule enforces the policy defined in lines 8-10 by sending to the normal processing pipeline traffic that is targeted at the server on the HTTPS port. Note that such rules implicitly protect the server from other misconfigurations. For example, attempts to access the GUI of the neo4j database (port 7474)—that was mistakenly configured for public access—are denied. Last but not least, let us consider a short-lived policy (lines 12-15)[3] that allows servers to bypass the scrutiny of middleboxes during one hour in order to achieve high throughput using a dedicated path for data backups to cloud storage services (e.g. AWS S3).

In this case, the exception could be deployed with a temporary high-priority rule (10000).

The rule may perform transformations to traffic packets that are more complex than what regular devices do before packets are sent out of an alternative interface (e.g. the red rule rewrites VLAN number to 5 and forwards traffic out of interface 2 for the matching packets). Note that unlike the HTTPS flow, the policy writers wanted to avoid using the normal path for some reason – say because under the default policy (i.e. priority 0) would have forwarded traffic out an interface that further in the path contain a series of performance limiting middleboxes. This example shows that both policies and temporary exceptions can be deployed in the campus using NSCs and serves as a preamble of the exception system described in Chapter 5.

[3]duration is not part of the PoLanCO syntax presented in Chapter 3 but can be easily added

Chapter 5. Enabling Short-Lived On-Demand Security Exceptions

Network security policies have been traditionally enforced by specialized network appliances. Enforcement devices, often referred to as middleboxes [1], analyze and process packets that traverse the network in a variety of ways to protect network users and end systems from malicious actors. For example, it is not uncommon to find devices such as firewalls, IDSs/IPSs, load balancers, or Network Address Translation (NAT) boxes at various locations to enforce policies that mitigate attacks, and prevent a variety of exploits. While middleboxes help network operators deploy policies, they have two main drawbacks. First, middleboxes tend not to run at line rate due to their complex packet processing pipeline, and second, they apply the same degree of scrutiny to all packets in the network even though not all flows require the same level of scrutiny as other. While there are high-end commercial solutions that solve the former, oftentimes commercial appliances are cost-prohibitive for organizations that have budget constraints. As a result, middleboxes often become bottlenecks that cause network users to experience severe performance degradation and/or unexpected behaviors to their flows.

Moreover, even if middleboxes were affordable and capable of handling line-rates, their heightened scrutiny of user traffic has led to an adversarial relationship between users and network providers. To ensure security, providers try to found out what users are doing by performing Deep Packet Inspection (DPI. On the other hand, users do not appreciate this meddlesome inspection and other downsides such as going through a lengthy approval process to access a site/server, being subjects of invasive monitoring, or experiencing poor performance.

We argue that this escalating arms race between users and providers is detrimental for both parties. Instead, we introduce an approach towards network security enforcement based on the concept of *short-lived on-demand security exceptions*. We bring network providers and **trusted** users together by (1) implementing coarse-grained security policies in the traditional way using conventional in-band security approaches (i.e. via affordable middleboxes) and (2) handling fine-grained policy exceptions outside of the data plane using context information provided by the users (or their applications) when a network flow is initiated or during the connection establishment between both ends in the communication.

Under an exceptions approach both parties benefit. On one hand, trusted network users divulge context information to network providers to receive special treatment for their flows. On the other, by allowing security exceptions, network providers not only reduce the load on traditional policy enforcement middleboxes, but focus inspections on general (untrusted) traffic. This chapter describes the requirements and design of a system that can allocate security exceptions on demand and in real-time, as opposed to the manual, close-to-static, committee-driven process to allow exceptions on campus networks. We highlight the relevant data structures and technologies needed to establish trust relationships between providers and network users, as well as the mechanisms and systems needed to deploy exceptions onto the network infrastructure. Lastly, we report performance improvements we obtained using a prototype exception system tailored to the transmission of big datasets to various storage systems and research facilities.

5.1 Middleboxes

The traditional way to enforce network security policies in a network requires the physical deployment and individual configuration of dedicated appliances (also called middleboxes). These network security devices have become highly sophisticated over the years and are now pervasive across many types of networks including campus, enterprise, cloud-based, and provider networks. Middleboxes provide several services and process network traffic in a variety of ways to ensure users and their applications are protected. Typically, network appliances, which are deployed at key places of the network infrastructure (e.g. at the network edge where campuses connect to their regional ISP, in front of a server pool, as point of entrance to critical resources like databases or file servers), intercept and perform some form of DPI on all traffic moving through the network. Specifically, once packets are intercepted, a middlebox analyzes the content of each packet (including not only headers but possibly the payload) to find matching packets based on predefined network patterns or well-known signatures, and then apply middlebox-specific processing to the matched packets. For example, firewalls, that are deployed inline on the physical network path, drop or forward packets matching certain (header) fields (e.g. IP addresses, port numbers, protocol flags, payload structure); IDSs/IPSs perform DPI looking at payloads to identify suspicious traffic and log, alert or block the activity based on packet frequency or packet payload; NAT boxes serve as an interface between the Internet and a private LAN where all hosts in the LAN have local/hidden IP addresses that must be translated into one or more public addresses when talking to Internet endpoints; load balancers distribute traffic across multiple servers ensuring that no single server is severely overloaded, thus, contributing to the mitigation of DoS attacks and aiding high availability.

Middleboxes are an integral part of network infrastructures because they not only they keep the network secure, but they often provide additional services to network traffic (e.g. content caching, QoS, rate-limiting). Although they have become the *de facto* mechanism to enforce policies the complex processing these devices have to go through—involving DPI—oftentimes causes them to be incapable of performing at line rates.

In recent years new commercial solutions (e.g. Palo Alto [25], SonicWall [26], FortiGate [27]) have emerged that can achieve (close to) wire speeds and offer traffic monitoring and policy management. These boxes are often cost prohibitive for organizations with tight budgets (e.g. schools, community colleges, startups and small organizations). Consequently, the only options left is to acquire inexpensive middle- to lower-tier devices or even deploy software-based solutions on commodity hardware (e.g. Bro [29] or Snort [30] IDSs running on a Linux box). While these affordable solutions are easy to deploy, they often become choke points in the infrastructure where user flows can encounter a performance hit or unexpected behavior due to variable packet throughput rates from the slow DPI.

In addition to this problem, note that even if these middleboxes were both affordable and capable of handling line-rates, the fact that they have historically been the preferred mechanism to enforce policies has led to an ongoing escalating arms race between network users and providers. For example, providers deploy middleboxes to block traffic destined to certain ports or addresses. In response, users tunnel traffic over open ports (e.g., port 80) to get around these limitations. Providers, in turn, deploy DPI to better identify the traffic traversing the network. Users then employ encryption to thwart the providers' DPI efforts, further intensifying the arms race. Providers then rate limit traffic that exceeds a certain rate, threshold, or otherwise looks suspicious, thereby continuing the escalation.

The deteriorated relationship between user and provider is harmful for both parties. Providers have to inspect all traffic; both acceptable (legitimate) and offensive traffic. Users feel their privacy is under constant invasion and their flows are significantly degraded. To de-escalate the arms race we argue that both providers and users should move away from an adversarial *modus operandi* where complex techniques are continuously developed to obstruct each other's needs, thereby making network management harder, and legitimate network workflows unnatural. Instead, users and providers should cooperatively participate in the compliance of security policies.

Specifically, if users share information about what they are doing, in return, providers will allow user flow(s) to avoid the normal policy compliance checks. We define these mutually agreed on safe flows *policy exceptions*.

Mapping higher-level policies perfectly onto network-level abstractions is a hard task where at times middleboxes may not be able to precisely implement a high-level policy (e.g. a policy for flows working on an NSF grant) thus, resulting in collateral damage; middleboxes end up either over-protecting—blocking traffic that does not need to be blocked, thereby limiting functionality—or under-protecting—passing traffic that should have been blocked, thereby increasing risk.

A classical under-protection example is the deployment of a specialized DMZ for research traffic—called a *Science DMZ* [124])—at the edge of the campus network (i.e. the box hanging off of the campus edge router in Fig. 5.1). Unlike traditional DMZs that are generally placed in between two firewalls (as shown in Fig. 2.2 and Fig. 2.4) and whose purpose is to separate public access to corporate resources such as web, DNS, e-mail servers from access to the LAN, the primary purpose of the Science DMZ is to eliminate any bottleneck that involves DPI from the network path in exchange of significant gains in network throughput. As a result, machines that join the Science DMZ have to deal on their own (e.g. via software solutions) against potential attacks from external (untrusted) entities as the protection offered by firewalls and IPS/IDS is forfeited and there is a direct exposure to the Internet. Additionally, campus network acceptable use policies are not enforced for devices in the Science DMZ; thus, nothing prevents a user from abusing the granted privilege and reaching unauthorized sites that otherwise would have been blacklisted by a campus firewall.

Ensuring an appropriate degree of network security is always a trade-off among various costs and benefits such as CapEx/OpEx, user (in)convenience, and performance. We showed above that , at present, it is hard for a user to share information

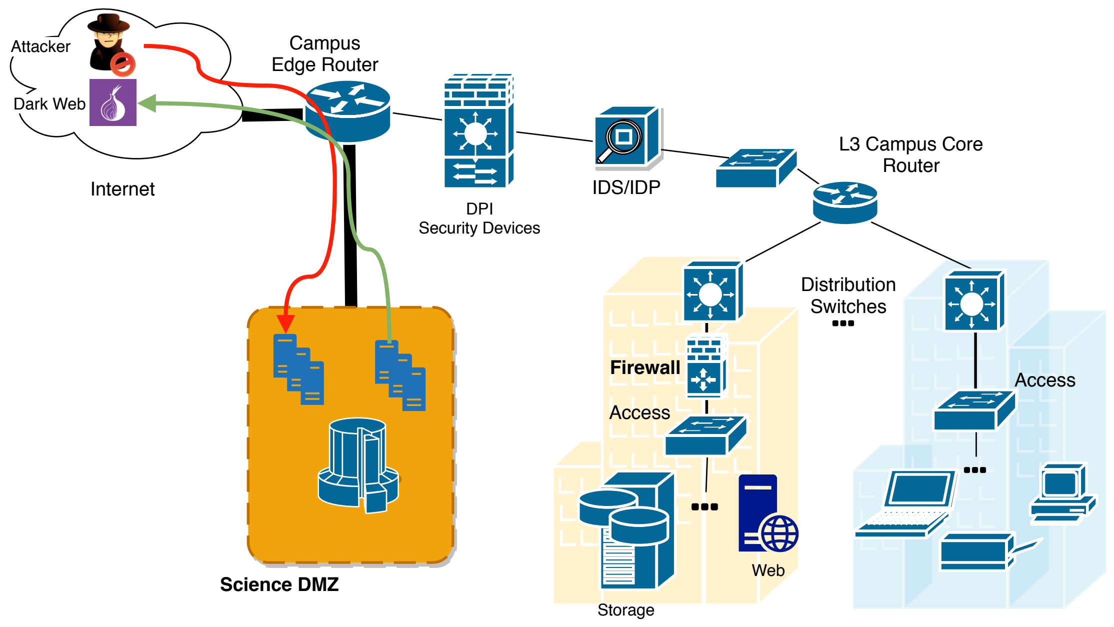


Figure 5.1: Security implications of a Science DMZ

about the context of her workflows in exchange for special treatment against network policies, first, by following an extensive and manual procedure that requires signatures, justifications, forms, committee approvals, that for the most part is static; and second, by registering a machine as part of a Science DMZ to achieve high-performance throughput at the risk of being exposed and unprotected by otherwise helpful middleboxes.

We re-examine the trade-offs involved in the current coarse-grained, middlebox-based approach to enforce network security taking into account that emerging technologies that enable programmability in existing campus networks can cope with research environments that are complex and rather dynamic that expect the network to adapt to their needs. Specifically, we propose a new approach towards network security based on the concept of *short-lived on-demand security exceptions* that leverages the network packet processing we described in Chapter 4 and uses the control plane to remove the context unawareness of the middlebox-based policy enforcement approach for exceptional flows.

The fundamental concept of our proposal is to use traditional (and affordable) security appliances (that have been optimized and hardened) to provide a base level of coarse-grained policy enforcement on untrusted traffic, noting that the performance costs increase with increasing policy complexity as well as traffic volume. To address the issue of invasive scrutiny for legitimate traffic, we support the ability to establish trust relationships between users and network providers using an authorization tree to hierarchically divide segments of the network flow space into trusted regions. These trust relationships can then be used to create fine-grained, short-term, trusted, on-demand exceptions to the base policies. The exceptions are implemented using well-defined protocols in SDN such as OpenFlow while keeping the cost of the base-level down.

In addition, some enforcement decisions (i.e., whether to grant an exception) are abstracted out of the data plane (i.e. the middleboxes) and moved to the control plane (flexible and driven by software and automated mechanisms), where they can be based on authenticated information provided by users indicating conformance (or not) to higher-level policy. In other words, if trusted users are willing to inform network providers about the type of traffic they will be using, their traffic may be eligible to bypass the middlebox compliance checks—and associated costs—applied to the traffic of other (less-trusted) users. More precisely, negotiated security exceptions can allow users to bypass certain middleboxes, allow otherwise prohibited traffic to temporarily traverse the network, or offer some level of QoS to authorized flows. Exceptions might also be made to stop normally-allowed traffic as well, for example to block or rate-limit unwanted traffic that would otherwise be delivered to a user based on network conditions such as time of the day, or changes in role/affiliation.

5.2 Exceptions as First-Class Entities

Most network policies are written in Acceptable Use Policy (AUP) documents describing allowable and expected user behavior, guidelines to protect IT resources in the campus network, and the mechanisms used to reduce risk and mitigate attacks, among others. While in the past such policies sufficed, current workflows are becoming more dynamic and dependent on the services provided by the network, even for disciplines that historically have not needed a significant amount of IT resources. Moreover, there are occasions where compliance to such policies cannot be fully achieved or by actually abiding by them could result in a major limitation (or disruption) of legitimate workflows that are well-aligned with organization's objectives and goals.

Research environments, in particular, have to cope with emerging workflows due to recent technology trends in research (e.g. Big data, machine learning, cloud computing, and IoT). Unfortunately, network security policies were designed for general purpose traffic and have not been appropriately adapted to cope with newer technological paradigms. Most institutions do not have the ability to create network policy exceptions – i.e., grant a flow, or all flows from a machine, the ability to bypass the normal network policy compliance checks.

Several institutions now support the concept of a Science DMZ that allows researchers to move their machine outside the university firewalls into a Science DMZ that is not subject to the university's policy compliance checks – or the performance overheads of applying those checks.

In a few rare cases, the university will allow researchers to request a network policy exception for a certain well-defined flow. The network administrators must review the request, and, if granted, manually configure the network to ensure the requested flow does not go through the normal compliance checks. Although there are slight variations in the actual steps among all institutions, the following procedure shows the steps that need to take place before an exception is granted (modified or removed) on campus.

1. Request form from IT department. While some of these forms are publicly available, some others can only be accessed through a shared folder or via e-mail.

2. Fill out exception form and obtain signatures from department chair(s). The form may include information such as the policy for which an exception is requested, list of systems, networks, and/or data affected by the request, fully qualified name of servers, detailed justification of requests, any attack mitigation controls.

3. Submit form to IT department. This step can be typically completed via an online form, an e-mail, and/or a hard-copy of the form directed to the office of the CIO (or CTO)

4. Once the form is received, IT staff gather background information—not always the user has knowledge of the low-level details of the network–and determine if the case needs to be escalated or an approval/denial recommendation can be made.

5. Contact requester if the information provided is not enough to provide a full assessment of the request.

6. The process is halted until more information provided ultimately by re-submitting an updated request that would require some (or all) of the tasks required in step 2.

7. Final decision is made based on the information provided and feedback collected after every stage. Should a request be denied, departments may appeal the denial by submitting more information (justifying their case). Departments may request extra meetings with IT to make a stronger case.

8. Set a specific duration for the exception (3 months, 6 months, 1 year). It is hardly possible to enable exceptions for short periods of time (minutes, hours, days). The whole process is severely involved and could last longer than the actual duration of the exception.

9. Review requests at time of renewal, ask users if exception need to be reaffirmed or not.

10. If a significant change took place in the network or there was a change in policy, all deployed exceptions need to be reanalyzed for newer approval.

It is evident that all these steps can be completed at best in the order of weeks, and, as we pointed out, involve extensive manual checking, meetings, even paperwork. In the end, the goal of exceptions is to arrive collaboratively at a win/win situation where requests are handled in a fair, appropriate, and timely manner, where both parties get something from working together.

While there are several drawbacks in the process: exceptions are the only way network users can provide context and express their needs to justify partial compliance to the security policies in place, and unlike the traditionally adversarial approach for behavior in the network, exceptions provide a space to develop and foster a cooperative and mutually respectful relationship between them.

The large number of steps to deploy an exception, and even the complexity to maintain them, is what makes exceptions second class entities. In our proposed approach, we treat exceptions as first-class entities. Our proposed security approach splits the enforcement of network security policies into two pieces. First, network administrators define general *base* network security policies to address common security issues and concerns, and deploy them to middleboxes much as they do today. These policies can be simple and even imprecise (e.g., over-protecting traffic and erring on the restrictive side by blocking a wide range of ports) or slow (e.g., employing extensive DPI). They can be deployed over long timescales as they are not intended to change frequently and are meant to handle general-purpose traffic. Moreover, since the types of policies expected at this level are coarse-grained, deployed using straightforward ACLs and lightweight filtering, or even software-based solutions, their deployment is inexpensive (i.e. existing network infrastructure or commodity hardware may be used) and scales well (performance cost is not critical).

The second part of our approach relies on making exceptions "first-class entities" that override the base-level policies defined above. Unlike current practices to treat special-cases in the network involving long timescales and several human interactions, our exceptions are designed to be short-lived and can be requested *on-demand*, thereby allowing the network to quickly adapt policy to meet the current needs of trusted applications, or to address the current security needs of the network. Providers (e.g., network operators, ISPs) grant policy exceptions to users—or their applications—that supply (trustworthy) information about themselves and the network traffic they will be generating. These dynamically created, flow-specific, limited lifetime exceptions can be implemented using well-defined SDN protocols used in programmable networks such as OpenFlow [125], NETCONF/Yang [57], or ovsdb [56].

5.2.1 Example Exceptions

As noted earlier, exceptions enable both network users and providers to come together to "negotiate". Users present their needs and justification for an exception and providers vet those requests allowing them if partial compliance to the general policy is outweighed by the consequences of enabling the exception. We present some motivating example exceptions below where exceptions could simplify and enhance operations in a campus network:

External SSH Access: Consider a policy exception that is dynamically created to allow an authenticated collaborating researcher to use `ssh` from a specific end system in the Internet (e.g. a national laboratory), to punch through a campus firewall and access a private `git` server containing shared data, without the need to set up a VPN. The user might present the network exception system with information about the authorized remote system, the National Science Foundation (NSF) project associated with both the external collaborator and the local researcher, the times of day the collaborator is allowed to access the git server, etc.

Low-Latency Paths: As another example, consider a policy exception that allows a highly interactive distributed application (e.g., a web conferencing application or an interactive game) to utilize low-latency network paths (as opposed to the default paths) among all participants in order to reduce delay, thereby improving the responsiveness of the application. The network should use information presented by the user (similar to the example above) to provision the appropriate network links and paths that would ensure that constraints required by such applications are satisfied.

Big Data Transfers: Consider an application that needs to move a large data set between a national supercomputer facility and the local campus HPC supercomputer. In this case, the user might provide context about the transmissions such as its frequency and duration (e.g. every weekday for two hours), type and size of data, source and destination end points, or project-related information (e.g. NSF grant, department project number, Principal Investigator(s)). Based on this information, the provider might decide to allow a security exception in which such application flows are (temporarily) routed around the network's IDS/IPS system to avoid its throughput-limiting DPI, thus enabling the flow to operate at much higher transfer rates.

All three examples above illustrate cases where general network security policy imposes unnecessary costs on legitimate workflows. Under the traditional approach towards network security, applications and workflows experience scenarios of constant bandwidth rate-limiting, sluggish behavior due to network device overload (causing high latency), or forbidden access to resources that should be shared. On the other hand, on-demand exceptions allow users to disclose beforehand the details of their traffic in order to justify their need for special treatment to the network provider. Given this information, providers no longer need to subject these flows to the general policy enforcement mechanisms and rather, as means of verification, can perform offline passive monitoring for these flows (i.e. not affecting their performance) to ensure granted exceptions are not used in unauthorized ways.

5.3 Exception System Design Requirements

We described in Section 5.2 the procedure that must be followed to request a policy exception on existing campuses that support (static) policy exceptions. The procedure is extensively manual. We consider this lack of automation a major setback to effectively coping with the security needs that arise from the dynamics of research environments. We also observed that emerging SDN network architectures, where the control plane is separated from the data plane to enable network programmability, present an opportunity to develop a policy system that can reconfigure the network *on-demand* treating exceptions as their primary entities, and significantly reducing the amount of work that humans have to go through during the request/approval process.

There are several design considerations that must be taken into account in order to implement such an exception system. We list them below:

Who can request exceptions? Naturally, defining what types of users can request exceptions is of the utmost importance. Letting untrusted users deploy exceptions in the network could lead to security vulnerabilities inside the network.

For example, we found that at times [126, 127], (static) exception requests are accepted via e-mail using publicly available forms. While filters can be defined to block non-institutional addresses, as we noted in Chapter 1, assuming that any internal user is trusted is not a good policy. Instead, a better approach is to authenticate users using their institutional credentials and against the role(s) they are associated with. In that way, it is possible to narrow down the groups of users that might request an exception (e.g. only faculty members and graduate students may request exceptions) and mitigate unintended usage of the exception system by both external and internal users.

Who grants the exceptions? Likewise, as we showed in Section 5.2.1, exceptions must be vetted by the real-world authorities responsible for the network (e.g. IT staff). In that way, the policy enforcement process is moved away from the middleboxes, where every packet is (possibly deeply) inspected, to the control plane, where network applications can make decisions based on high-level information about the request. This is particularly beneficial for the performance of the system because unlike middlebox-based enforcement, where decisions must occur at (or near) line rate, control-plane decisions may take place on slower timescales before exception flows are initiated. However, in order to avoid the bottleneck of potential deliberations between organization units, trust decisions should not be made by a single entity (e.g. campus IT), but rather should be distributed in a controlled way among users to be able to scale the trust management system (see authorization tree below).

What is the scope of traffic that is subject to exceptions? Exceptions should be granted to individual flows. Should a workflow involve the use of an application that requires multiple connections (e.g. bwctl, rclone), multiple exceptions, one per connection, must be instantiated. Exceptions should be associated with **only** the number of connections that are necessary for the legitimate execution of a workflow. Unlike the *Science DMZ* approach, where privilege (exception) is given to **all** the traffic to/from a set of machines (i.e. exception is given out on a per host basis to a set of machines), an on-demand exception system should lock down the privilege of by-passing compliance checks to only a small number of flows; all other flows should still be subject to DPI done by middleboxes. Note that deploying exceptions for fine grained flows is challenging because some header fields are variable and cannot be known in advance (e.g. source port, destination IP address resolved by a DNS server). In such scenarios, the trust should be refined to precisely match the flow as soon as the flow appears (see Section 5.5.2 for ways to achieve late binding of flows to exceptions).

How to specify exceptions? Specifying exceptions is where the needs and requirements of network users can be described. The system should provide an interface for trusted users to input information about their required flows. In order to foster the automation of the system, exceptions should be *structured* using a predefined syntax or markup. In this way, control software can be developed to parse the contents of each exception and automatically transform the exception into network configurations. While there is no exception specification standard, at a minimum, the defined markup should be equivalent for all units in the campus network.

What information should be included in an exception? The mechanism used to request exceptions must include information related to the context in which the exceptions will be deployed. The exception request should include information such as application IDs, grant/project numbers, laboratory/group names, type of traffic being sent, etc. Unlike unstructured and paragraph justifications found in current forms to specify policy exceptions, context information should be precise and verifiable in order to quickly determine whether an exception is allowable or not.

What are the processing times for exceptions? Perhaps, the major problem of the current approach towards policy exceptions is its inability to cope with dynamic workflows that require near real-time deployments. Consequently, the exception system should deploy exceptions dynamically and on-demand. Users should be able to create (and tear down) exceptions in short timescales, avoiding the delays caused by tasks such as discussions in committees, collecting signatures for approval, scheduling meetings with IT staff, etc.

Backwards Compatibility. As noted earlier, varying degrees of security, yields varying levels of user (in)convenience. An exception system should not require a change in well-established legacy applications. Traditional tools such as SSH, RDP, iperf, bwctl, rclone, etc. should be unaware of whether an exception is being used or not. Likewise, traditional protocols used for route exchange and network management (e.g. SNMP, OSPF) should operate as they usually do in spite of the presence of policy exceptions. This will ensure that a campus network with no exceptions will safely default to traditional security enforcement mechanisms and no service is disrupted.

5.4 Implementing An Exception System

Taking into account the design considerations outlined above, we describe how those requirements were handled in our implementation. Overall, the exception-based security model is divided in two main parts, first, an interface for network providers to define trust (e.g. establish (on long time scales) which users are trusted to define exceptions for what portion of the flow space); and second, an automated request/response system that applications or users can contact to ask that exceptions be enabled in the network (on short time scales).

5.4.1 Defining Trust

A trusted flow is defined as a flow matching a portion of the flow-space controlled by a trusted user. The first part of the on-demand exception system involves formulating high-level policies that define trusted flows–that is, the conditions that must be satisfied for any connection to be associated with an exception. These decisions are made on human timescales and often involve human validation of the policy. The key to our approach is that trusted flows are defined before they are used. Campuses that do support (static) exceptions do not manually validate every individual exception as flows come in. Instead, they validate broad classes of traffic manually, and then reconfigure the network to allow those flows to be created dynamically and automatically by-pass middleboxes. Another difference of our approach from the traditional enforcement of policies in middleboxes, is that the definition of trust policies associates users/roles with specific flows—essentially to provide a "responsible party" for each flow granted an exception. To allow delegation of responsibility, the mechanism utilizes an *authorization tree* that arranges the set of all possible network flows, conditions, and possible actions, into a hierarchy where each (child) node in the hierarchy represents a subset of the flows, conditions and allowable actions, in the parent node. One or more users (typically represented by a group) are then associated with each node in the tree, giving them authority to define allowable exceptions for that portion of the flow space. This way of handling trust, removes the burden from campus IT staff (who are generally the root of the tree and have access to the whole network flowspace) and allows users to delegate responsibility for certain flows to other users, creating hierarchical authorization schemes consistent with the organization of administrative responsibility in the Internet and within the institution.

Exception Specification

The *authorization tree* is the place where information about the network users and their network traffic is stored. This data structure validates exception requests against pre-defined policies by users with varying levels of responsibility. Specifically, each node in the tree identifies a portion of the possible flow space and has an associated *Exception Specification (ESpec)* that contains high-level information that must be satisfied by any exception request (e.g. who, what, when, where) as well as information about the current status of the network (e.g., link congestion, overlapping or conflicting exceptions currently installed, available resources, etc). While one could envision ESpecs being written in a programming language as "plugins" to nodes in the authorization tree, such a design would make it more difficult for network providers to verify that a plugin is enforcing the policy correctly. As a result, ESpecs should be specified in a high-level policy definition language (or specialized markup) that could be easily compared against the intended policy exceptions without exposing the complexity of low-level implementation details to deploy an exception. For example, exceptions could be specified using human-readable sentences in PoLanCO (described in Chapter 3) and be subsequently translated into the defined markup in order to reconfigure the network. To provide an example of what an ESpec language might look like, consider the markup syntax shown in Fig. 5.2.

```
Exception Request Type: Add | Remove | Update
Auth Credentials: User ID | App ID | Project ID
Match: <FlowSpec>
Action: Max Bandwidth Path |  Min Latency Path | Min Hop Count Path | Block
Network Condition: Path Load < p | Time in [HH:MM, HH:MM]
Exception Duration: Flow Lifetime | n {days, hrs, mins, secs}
```

Figure 5.2: Example markup language for an ESpec

It is important to emphasize that the ESpec syntax presented in Fig. 5.2 may be changed to include (or remove) other information that network providers might consider important to know from the users in order to deploy an exception. However, it is critical that whatever information is included in an ESpec, it should not pose a bottleneck in the deployment of an exception (e.g. triggering an e-mail sent to the supervisor and waiting for her approval).

With the exception of the exception request type, which is the mechanism this syntax uses to know the type of processing the automated system must follow, all the ESpec information will be found in the nodes of the example authorization tree described below and displayed in Fig. 5.3.

Navigating the Authorization Tree

The authorization tree of the exception system specifies the trust relationships among multiple groups of network providers. Recall that the objective is to divide up the flowspace in a hierarchical manner, delegating the task of defining allowable flow exceptions to the (human) users responsible for those flows. In the context of a campus network, the root of the authorization tree would be defined by the Campus IT staff (Group: Campus IT) and would encompass all flows, actions, and network conditions on campus (represented by the * character in each property of the ESpec).

In the example shown in Fig. 5.3, Campus IT delegates the definition of exceptions for secure copy and secure web (SCP and HTTPS) flows originating from the College of Science (traffic: src=128.123.0.0/20,dstport=22,443]) to the IT staff in the College of Science (Group: CoS IT). Likewise, Campus IT delegates to the IT staff of the College of Communications possibility to deploy exceptions for all incoming traffic whose destination is in the college, although only for short periods of time (less than 10 minutes) between 9 and 10 am. Navigating down the tree, CoS IT staff members might further delegate the definitions of exceptions for SCP and HTTPS flows originating in the Biology Department (traffic: [src=128.123.0.0/24, dstport=22,443]) to the IT staff in Biology (Group Biology IT). In this case, the exceptions may ask for maximum bandwidth or low latency paths but should only be deployed in the network for the duration of the flow. For example, if the procedure involves transferring a data set, the exception should be torn down once the data set is successfully copied to the remote system and a TCP_FYN packet is received at the source. Moving further down, Biology IT staff might delegate the definition of scp and https separately to different labs. For example, exceptions for traffic originating in the Genomics Lab (traffic: [src=128.123.0.32/27, dstport=443]) are now limited to bandwidth paths whose destination is Google Drive, Amazon Web Services S3 storage, or the local object store. The above indicates that exceptions for the Genomics lab are for data transfers for cloud storage systems, whereas the exceptions of the Micro Biology lab are restricted on the deployment time rather than the final destination.

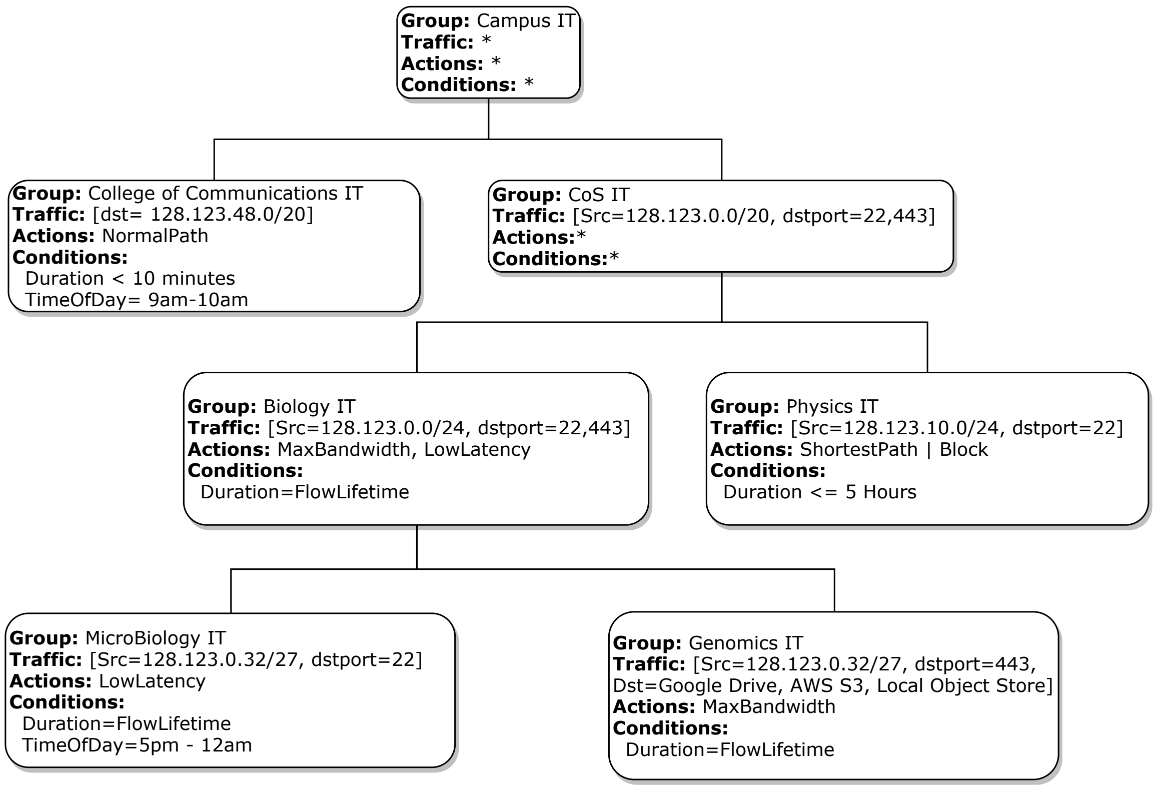


Figure 5.3: An Exception system authorization tree

5.4.2 Prototype System Architecture

With an authorization tree in place, where policies are defined by dividing the flowspace, the second part of the exception system addresses the need to automatically change the network state (e.g. using Network Security Caps described in Chapter 4) to deploy (or reject) the requested exception.

The automated component of the exception system accepts exception requests from trusted users that include all the information contained in an ESpec. Such requests are made via a REST API call (recommended) or through a web interface form that requires the user to enter the details manually.

The information in the request is evaluated against the authorization tree to determine if an exception is allowed (e.g., checking validity of credentials, ESpec format, valid request type, matches the flowspec, etc). If the user requesting the exception is trusted for the portion of the flow space to which the ESpec belongs, the system creates the exception by invoking SDN network management actions (e.g., computing OpenFlow rules, resolving domain names, learning location of the affected end-systems, etc) to deploy the exception to the appropriate network elements.

To test and evaluate our approach, an on-demand security system as described above was developed and deployed in the University of Kentucky. The system, called *VIP Lanes*, leverages SDN equipment found at multiple locations of the network and deploys exceptions by interacting with the SDN controller and installing (or removing) OpenFlow rules in the SDN-capable devices to configure the network to handle exception requests.

VIP Lanes is a secure and distributed system composed of modular and specialized components that perform deployment of security exceptions for high-throughput paths. The overall architecture and the interactions among components are shown in Fig. 5.4 and described below.

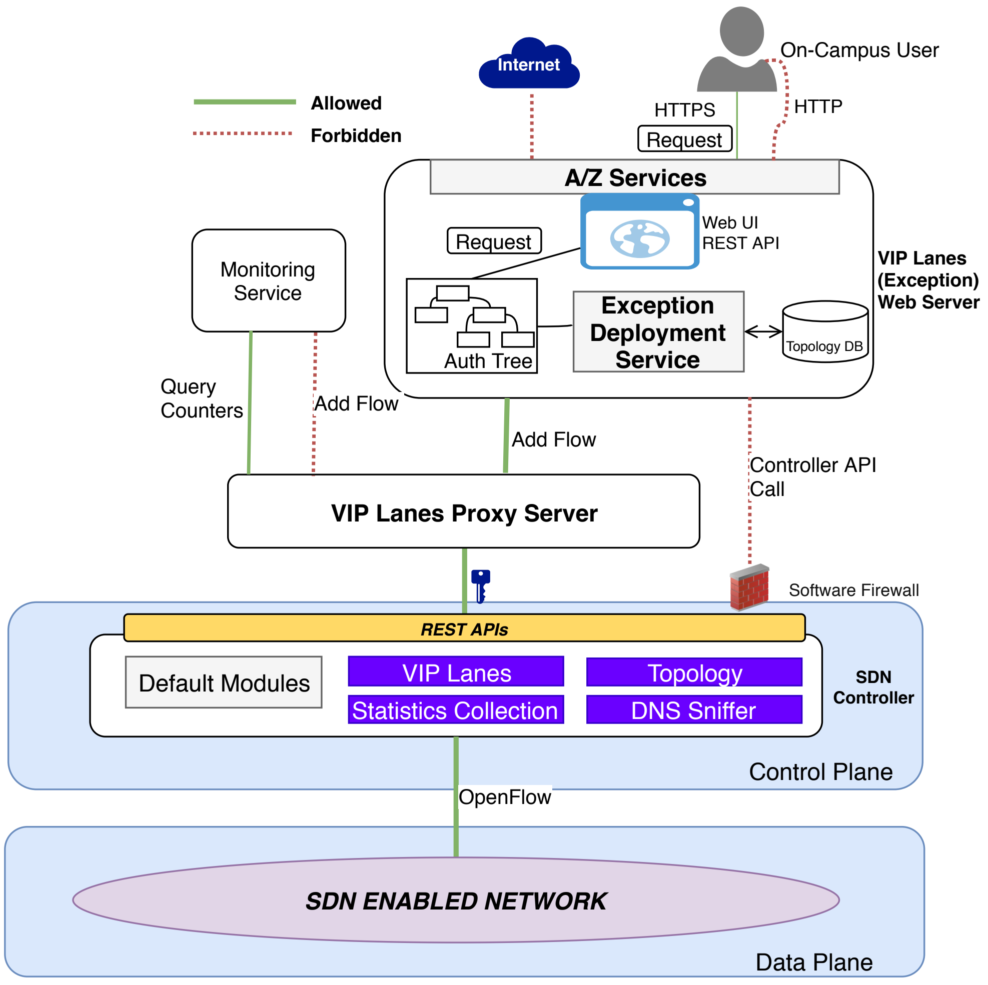


Figure 5.4: The VIP Lanes exception system architecture

VIP Lanes Server: The only point of contact for users and their applications to request an on-demand exception is through the server which can be considered the heart of the whole architecture. In order to protect the exception system from external attackers or snooping within the network, the server only accepts requests from campus internal IP addresses using encrypted HTTP requests. Furthermore, as the point of entrance to the system, the VIP Lanes server uses the campus' authentication services (LDAP) to ensure that only university members can request exceptions. Once a user has been authenticated, her exception request, which can be specified either via a web interface or by calling the server's REST API with a personal *private key*, is evaluated against the authorization tree created by the network administrators. Upon success, a deployment service is triggered to handle the request and translate the exception request into the appropriate configurations to the underlying network (see Section 5.4.3). As described shortly, the translation process of the VIP Lanes system requires querying detailed information about the network topology, information stored in a graph database. Because both the authorization tree and the topology database store sensitive information about the network and the policies, only local processes in the VIP Lanes server can reach them.

Topology Database: The deployment of exceptions for dynamic workflows require the exception system to maintain an accurate view of the network. VIP Lanes leverages current advances in NoSQL databases, particularly graph databases, and uses the Neo4j graph database to store detailed up-to-date information about the network based on information discovered by the SDN controller (e.g. hosts, SDN switches), SNMP (e.g. routing capabilities, VLAN information, MAC tables), and external data sources provided by the network administrator (e.g. location of certain middleboxes). The collected information is significantly broader than what SDN controllers provide. Graph databases are beneficial for VIP Lanes due to the following reasons:

- the *Cypher* graph database language – a declarative query language for Neo4j that simplifies the maintenance of topology data and provides an intuitive syntax to construct constrained queries, including path computations;
- a direct mapping from a network topology (devices, links) into the same representation in the database using nodes and edges. Further, the ability to manipulate sets of labels assigned to the stored elements allows the

representation of more complex network abstractions like active flows, IP addresses, topology snapshots, and virtual network functions (e.g., NAT);

- the ability to store heterogeneous collections of data as properties of elements in the network such as DPIDs for switches, MAC addresses for hosts, and bandwidth capacity for links; and
- an intuitive GUI (Fig. 5.5) that allows network operators to view current (and past) topology information, and to ask simple questions that are otherwise tedious to implement in imperative programming languages (e.g., *"what active flows go through switch X and avoid middleboxes of type T?"*). The prototype implementation is limited to the type of paths – e.g. widest, fastest, etc. However, as presented in Chapter 6, it could be extended to support exceptions and policies based on Neo4j queries.

Proxy Server The VIP Lanes proxy server acts as a gateway to access the SDN controller. The security exception system is a novel approach towards collaborative policy enforcement between users and providers. For example, as confirmed by experimental results, security exceptions for the transmission of big data transfers yielded a significant performance boost for campus researchers. However, the system could also open up potentially dangerous new avenues of attack—including attacks where an attacker could gain complete control of the underlying programmable network by breaking into the VIP Lanes server or the SDN controller. The fact that the exception system consists of several components that interact with the SDN controller creates a reasonably broad attack surface. Consequently, it is critical to secure the VIP Lanes exception system itself by analyzing and tightly constraining each of the requests that are sent to the controller. This is necessary because usually the NBI of existing SDN controllers enable a wide range of network management operations that are not needed by the components of the exception system. Moreover, the access control mechanisms present in the controllers are very limited, to the point that some of them (e.g. Floodlight, RYU, or POX) do not provide access control for REST-based APIs whatsoever. The Aruba VAN [55] controller used in the prototype supports very limited Role Based Access Control (RBAC) that currently provides a single role with access to all controller features (i.e. sdn-admin), giving far more control than is needed by the VIP Lanes exception server.

If attackers were to gain access to the *sdn-admin* role, they could bring ports up/down, capture any packet, inject traffic, overload the switches with control messages—all being capabilities not needed by the VIP Lanes exception system. To reduce the risk of attack but yet work with existing controllers, the VIP Lanes Proxy is the only entity authorized to access the SDN controller's APIs. All requests to the controller must go through the VIP Lanes Proxy that inspects the API calls and blocks any requests that invoke controller capabilities not needed by the exception server. In addition, the VIP Lanes Proxy serves as a certificate authority, signing client certificates (i.e. one for each component in the VIP Lanes system) so that clients can be identified and associated with a list of APIs they are authorized to invoke (i.e. a whitelist). (Note that if the SDN controller has no access control, a firewall – either standalone or on the controller, say via `iptables` – is needed to ensure packets cannot bypass the VIP Lanes Proxy to reach the controller.)

The data structure used to implement the VIP Lanes Proxy whitelist functionality is a map of clients (identified by the Common Name (CN) field of their signed certificates) to URLs (REST endpoints) that components are permitted to use (including the HTTP commands they are allowed to use per endpoint). Table 5.1 shows example whitelist entries, where the URLs are specified as extended regular expressions to narrow down the action field. For example, the first entry enforces all VIP Lanes management calls to use vlanes-specific structured cookie identifiers, isolating on-demand exceptions from the default general policies controlling campus traffic, and obscuring the meaning of an identifier from would-be attackers.

On a similar note, if an attacker compromises a component such as the monitoring system and then asks the SDN controller (via the VIP Lanes Proxy) to make a change to the network, its connection to the VIP Lanes Proxy would be ignored by the VIP Lanes Proxy, reducing the risk of an attack on the monitoring system. Moreover, if the attacker requests a resource that it does not have permission to (e.g. install a flow), its connection will be dropped, and therefore the risk of corrupting the network operation is minimized. Note that in this case, the attacker would still be able to use the customized statistics API. However, this is a very limited read-only operation that would cause no harm to the campus network.

The VIP Lanes Proxy default action for unauthorized requests is to ignore/drop the connection. However, the VIP Lanes Proxy could take more complex actions like reporting the incident to the network operator, or forwarding the traffic of the compromised component to a honeypot to learn more about the *modus operandi* of the attacker.

Table 5.1: Example whitelist entries in a VIP Lanes Proxy

Cert CN Field	*Authorized SDN Controller APIs*	*HTTP Commands Allowed*
vip-site.uky.edu	`^/sdn/viplanes/ab01[a-f0-9]{12}$`	`GET, POST, DELETE`
vip-site.uky.edu	`^/sdn/v2\.0/of/datapaths/[^/]+/ports/[^/]+$`	`GET`
vip-stats-db.uky.edu	`^/sdn/stcl/stats/counters$`	`GET`

SDN Controller Campus OpenFlow-enabled devices were paired with the Aruba VAN SDN controller [55]. Besides the built-in modules that allow the SDN controller to provide basic services like a web-GUI, REST APIs for simple management and individual rule installation, discovery of OpenFlow-devices and end-hosts via DHCP or their own neighbor discovery protocol (BDDP), the VIP Lanes exception system implementation consists of four management modules:

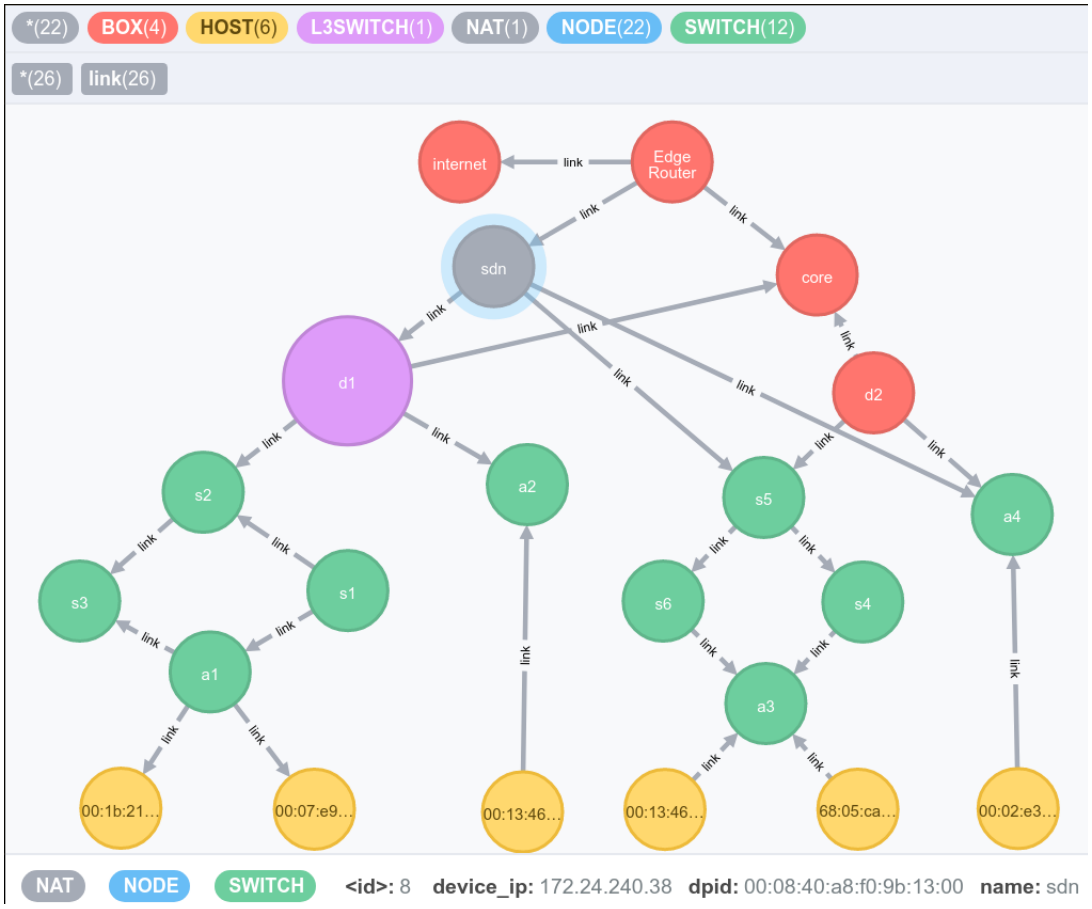
- **Management Module**: Central module whose purpose is to process automated requests coming from the path computation service and add, update or remove VIP Lanes from the network.
- **Global Topology Module**: Provides mechanisms to maintain versioned snapshots of the existing connections in the network as well as a cache of hosts learned via ARP messages.
- **Statistics Collector Module**: A multi-threaded application that periodically queries/polls switches for byte/packet counters of the installed VIP Lanes.
- **DNS Sniffer Module**: A helper module for cloud storage providers that do not provide fixed IP addresses for data transfers (e.g. Google Drive, Amazon S3). The module extracts resolved IP addresses from DNS response packets issued by the storage provider before the transmission is initiated.

SDN-Enabled Network The network where exceptions are deployed comprising several NSCs (described in Chapter 4) as well as the end systems attached to them. In order to join the SDN network (and instantiate VIP Lanes), buildings (and departments) have to deploy at least one NSC-capable device—typically a distribution and/or an access switch/router—and attach it to the SDN core making sure that all general-purpose traffic is routed to the campus "normal core" by default thus, preventing disruption of regular service to network users (Fig. 5.8). Today, there are more than 14 buildings in our campus network that have joined the SDN network and are eligible to deploy on-demand exceptions, including departments of disciplines that historically have not involved advanced IT tasks as part of their workflow but now benefit from the exceptions that they are allowed to deploy. While the VIP Lanes system is currently stable, complex tested networks needed to be created during the development phase of the system [128]. The testbeds resembled (most of) the behaviors found in the production network (e.g. inter-VLAN routing, IP address assignment, middlebox bottlenecks) and ensure that security concerns were mitigated. There were three iterations of the testbeds with increasing degrees of reality (and complexity). The first deployment was in a controlled environment in GENI [42], made entirely of software switches and therefore isolated from actual campus traffic. Then, a laboratory testbed comprising hardware switches (Aruba 3500/3800) that, in spite of running multiple OpenFlow (virtual) instances, gave us a closer to reality scenario and allowed the control software to be tested before rolling it out to production. Lastly, the system was tested in a limited portion of the campus network that included systems located in the Computer Science department.

5.4.3 Deploying Path Exceptions

Fig. 5.4 showed that after an exception is validated against the main authorization tree an exception deployment service is triggered in order to deploy the user request in the SDN network. In the VIP Lanes system, the path computation service deploys exceptions. The service attempts to find a middlebox-free path with the information provided by the user in the ESpec.

While the path computation could be written in a conventional imperative programming language by calling existing graph libraries [129, 130], computing the custom paths needed by VIP Lane would require tailoring the path computation algorithms included in these libraries to handle networks made up of heterogeneous elements, which is an error-prone and time-consuming task for a network programmer/operator. Instead, we opted to leverage the built-in capabilities of the Neo4j graph database to perform the path computation and topology data maintenance within the database.



<id>: 8 **device_ip:** 172.24.240.38 **dpid:** 00:08:40:a8:f0:9b:13:00 **name:** sdn

Figure 5.5: Neo4j GUI displaying the current topology, list of existing labels in the database, and detailed information assigned to the highlighted "*sdn*" node

Currently, VIP Lanes is capable of calculating three types of paths: the *fastest* (for low-latency requests), the *widest*(for high-bandwidth), and the *shortest* (for default routing). The *fastest* path query chooses the route based on the sum of latencies of all links on the path; the *widest* path query chooses the route with the maximum (greatest) bandwidth capability of the minimum-bandwidth link in a path; the *shortest* simply chooses the path with fewest hop counts. While Neo4j provides a built-in function for the shortest path, two declarative queries to compute the fastest and widest paths were constructed using Cypher. The queries are shown in Fig. 5.6.

Minimum Latency (fastest) Path

```
MATCH     (src {ip: srcip})-[:version]-(current:CURRENT),
          (dst {ip: dstip})-[:version]-(current)
WITH      src, dst
MATCH     p=(src)-[r:link*..{}]-(dst)
WITH      p, reduce(Latency=0, r in relationships(p)) |
          Latency + (r.latency)) as TotalLat
ORDER BY  TotalLat
RETURN    EXTRACT (n in nodes(p) | n.name ) AS names,
          EXTRACT (r in rels(p) | r.vlan ) AS vlans,
          ...,
          EXTRACT (n in nodes(p) | labels(n) as labels,
LIMIT     1
```

Maximum Bandwidth (Widest) Path

```
MATCH     (src {ip: srcip})-[:version]-(current:CURRENT),
          (dst {ip: dstip})-[:version]-(current)
WITH      src, dst
MATCH     p=(src)-[r:link*..{}]-(dst)
WHERE     ALL (n in NODES(p)
          WHERE SINGLE(m IN NODES(p) WHERE n.name=m.name))
WITH      p, EXTRACT (c in RELATIONSHIPS(p) | c.bw_cap)
          AS bwidths
UNWIND    bwidths AS b
WITH      p, MIN(b) AS Bandwidth '
WITH      p, length(p) AS Hops '
ORDER BY  Bandwidth DESC, Hops ASC
RETURN    EXTRACT (n in nodes(p) | n.name ) AS names,
          EXTRACT (r in rels(p) | r.vlan ) AS vlans,
          ...,
          EXTRACT (n in nodes(p) | labels(n) as labels,
LIMIT     1
```

Figure 5.6: Cypher queries to compute the fastest and widest paths

All three types of paths are middlebox-free. Relevant middleboxes and non-SDN devices present in the network (red nodes in Fig. 5.5) are identified in the database primarily through manually entered JSON-encoded configuration files that contain descriptions of the interfaces present at every middlebox (e.g., MAC and IP addresses, or neighbors). In some cases, these middleboxes may be discovered by the controller as hosts, and consequently, the path computation service uses the information stored in the configuration files to override the type of node that needs to be stored in the database.

When a path query is run in Neo4j, it returns not only the nodes and edges along the computed path, but also a selection of label and property data for each node and edge. The topology information is vital to the successful construction of custom exception paths because it describes what each OpenFlow switch must do in the selected path; thus, enabling NFV in the SDN network. The control software parses the topology information obtained from the database query and maps it into OpenFlow rules that the SDN controller installs at every NSC along the path. The generated rules ultimately dictate the behavior of every individual NSC for every approved exception. Consequently, it is common to have "multi-function" switches (like the *sdn* node in Fig. 5.5) that operate differently based on the location of the end hosts in the computed path. For example, for on-campus transmissions (e.g., *"a transfer from the Computer Science department to the Physics department"*) the *sdn* node behaves as an L2/L3 switch that rewrites MAC addresses or VLAN tags for every packet header in a flow. Additionally, that **same** switch functions (simultaneously) as a stateless Network Address Translation device that hides IP addresses of the LAN for flows going off-campus (e.g., *"sending data to a national lab"*). The flexibility of graph databases helps store not only the *de facto* NAT table, but also the set of public IP addresses to appropriately assign and produce OpenFlow rules that rewrite the source and destination IP addresses of packet headers for outbound (i.e., from the campus) and inbound (i.e., to the campus) exceptions going through the SDN network.

All path calculations require an accurate representation of current network topology in the graph database to prevent packet drops or forwarding traffic to restricted parts of the network. Since the network topology changes over time, the changes must be recorded in a timely manner to ensure that the database topology accurately reflects the current state of the actual topology. The up-to-date topology information is maintained through the versioning algorithm described in the next subsection.

5.4.4 Topology Versioning

Deploying exceptions requires an accurate view of the network. The challenge lies in determining the frequency of topology data updates without compromising efficiency. Ideally, the topology stored in the database T_{db} should always match the actual topology T_c (known by the controller) at any given time. However, proactively maintaining $T_c == T_{db}$ at all times is expensive and adds unnecessary overhead: if no exceptions are requested for a period of time, it is wasteful to continuously update T_{db}. An alternative approach is to check if $T_{db} == T_c$ before *each* path query is executed and update T_{db} if the condition is not met. The latter approach eliminates unnecessary topology updates. However, it adds a user-noticeable delay to the path calculation process as the topology grows. To tackle this problem, we implemented a topology versioning algorithm. Fig. 5.7 illustrates how the mechanism operates when components of the architecture trigger relevant events.

When the controller boots up or a new version of the topology module is deployed (light-gray box), T_c represents the topology learned by the controller. A random 64-bit number v_c is the version of the topology stored in T_c. We define a flag T_{c_req} that indicates whether the topology has been requested by the path computation service or not. Later, when the controller detects a *topology event* (yellow elements), the version number v_c is increased by 1 iff (1) the event is different from a host joining or leaving the topology—a frequent event based on a controller cache timeout that is updated whenever the controller sees a packet from a host—and (2) the path computation service has requested a newer version of the topology. Both conditions ensure the algorithm avoids the situation where T_{db} is constantly being updated even though the path computation service does not currently need the latest version.

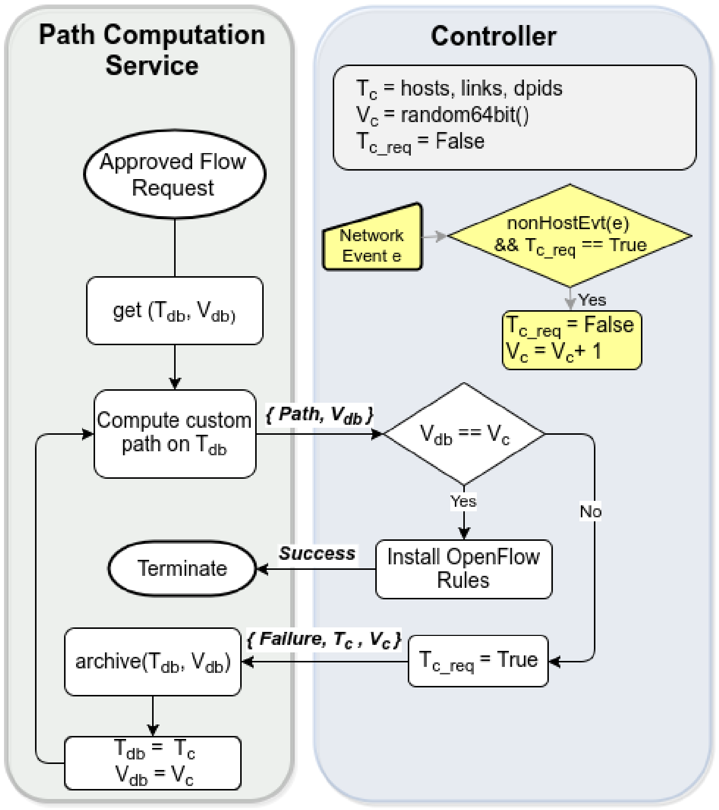


Figure 5.7: Topology versioning mechanism flow chart

When the database is initialized, T_{db} is initialized to the current value of T_c. Likewise, the database version number is initialized to the controller's version number (i.e., $v_{db} = v_c$). Later, when VIP Lane request comes in via the path computation service (white elements), the path computation service calculates the VIP Lane path using T_{db} and sends the computed path along with v_{db} to the controller to actually install the SDN path. When the controller receives the request, it first checks if v_{db} is equal to the v_c (i.e., the topologies are in sync). If so, OpenFlow rules are generated and the algorithm terminates. Otherwise, the controller realizes the path computation service needs a newer version of the topology (by setting T_{c_req} = True), and rejects the current path installation request.

As a result, a response message is built including the most recent values of v_c and T_c. Once the response gets back to the path computation service, current data of T_{db} and v_{db} is archived as an old version in the Neo4j database, and new snapshots are added the T_c and v_c values provided in the response. After this update, the process starts over again and the path calculation is done on a more recent topology snapshot.

5.5 Evaluation

This section presents performance measurements collected when deploying on-demand exceptions and sending/receiving data to common sites used by researchers and scientists in the campus of the University of Kentucky. The factors affecting big data transfers when the network is not the bottleneck are analyzed. The analysis shows that the tool used to perform transfers has a significant impact on the final performance of a transfer.

5.5.1 Throughput Measurements to ESNet Sites

In the first set of experiments, VIP Lanes exceptions were deployed to measure the throughput at various locations on the campus network (Fig. 5.8) to reach sites that are known to be used for research activities and therefore, can be trusted (i.e., are allowed to by-pass campus network policy compliance checks).

Specifically, the `bwctl` [131] program measured the throughput from one of the University of Kentucky campus libraries (KSL), the Computer Science department (JFH), a newly built science building (JSB), and the department of agriculture (AG) to ESnet sites located in various geographic regions of the United States (San Diego, Washington D.C., and Chicago) and to the Data Transfer Node (DTN) at the University of Kentucky which is located in the campus Science DMZ.

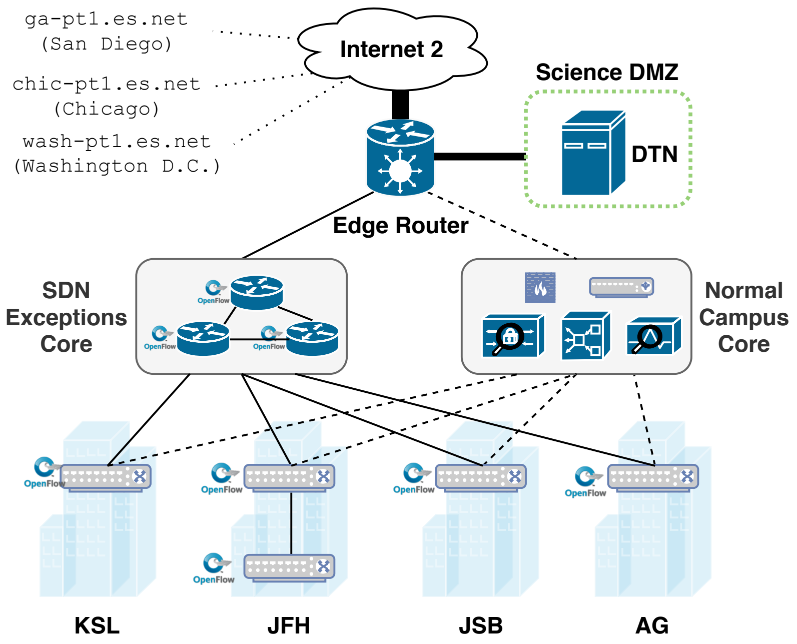


Figure 5.8: SDN-enabled campus topology used to deploy exceptions

In order to eliminate the influence of variations in the client machine specifications used in each of the campus buildings, all the tests were run on a Macbook Pro with an Intel Core i5 processor 2.4 GHz, 16 GB RAM, and an external Thunderbolt2 10G adapter attached to it. Additionally, jumbo frames (i.e. MTU 9000) were enabled in the client machine as well as in all the VLANs used to perform the transfers. Lastly, some variables of the system's TCP/IP stack (e.g. TCP window scale factor or receive buffer) were tuned following the recommendations published by ESnet [132] to maximize the performance during each test.

For each site and building we measured two throughputs. First, the performance obtained by letting the *Normal* campus network security appliances inspect packets to enforce policies. Then, short-lived on-demand security *Exceptions* were deployed to send data from the laptop to the trusted sites and the performance over the middlebox-free exception path was measured. ***Note on rule installation times***: We observed that on average, it takes 314 ms to deploy each exception in the data plane. Since deploying an exception happens as part of a big data transfer, it has almost no impact on the performance.

Table 5.2 shows the data collected after running the above experiments. At a first glance, it is clear that using the VIP Lanes exception mechanism researchers in most cases could benefit from a performance boost from tens of megabits per second (under normal conditions) to multiple gigabits per second (using exceptions). Unsurprisingly, the improved throughput was affected by the geographic location of the trusted site, e.g., speeds to the DTN reached close to 7.2 Gbps whereas measurements at San Diego (on the opposite coast of our campus) were below the 700 Mbps mark. Nonetheless, as can be seen in Fig. 5.9, the speedup factor, i.e., how much faster the throughput is by using exceptions, was not necessarily bound to geographic location. For example, the improvement from AG to the DTN was only of 11x the normal throughput, whereas from that same location to Chicago ESnet site the factor jumped to 50x. In most of the cases, the speedup factor was higher than 20x with only two data points below.

Note that for certain workflows (e.g. Big Data transfers) the specs of the client machine are expected to be more powerful than those of the laptop used to run the tests. Therefore, these results could serve as a baseline or reference point of the potential improvements that can be obtained using more capable systems.

5.5.2 Exceptions for Transmissions to an External Cloud Provider

Transferring big data to various cloud storage providers is becoming increasingly important in recent years. There are multiple factors affecting the performance of big

Table 5.2: Throughput from four campus buildings to trusted sites

Site	*KSL*	*JFH*	*JSB*	*AG*
San Diego, CA	31.3 (**669**)	28.8 (**671**)	31 (**669**)	19 (**663**)
Chicago, IL	182 (**3959**)	36.4 (**3129**)	95.4 (**3974**)	74.1 (**3707**)
Washington, D.C.	70 (**1289**)	29.4 (**1400**)	69.4 (**1570**)	56.7 (**1532**)
DTN	300 (**7120**)	67.7 (**7140**)	320 (**7200**)	644 (**7123**)

*The numbers are shown as *Normal (**Exception**)* throughput in Mbps

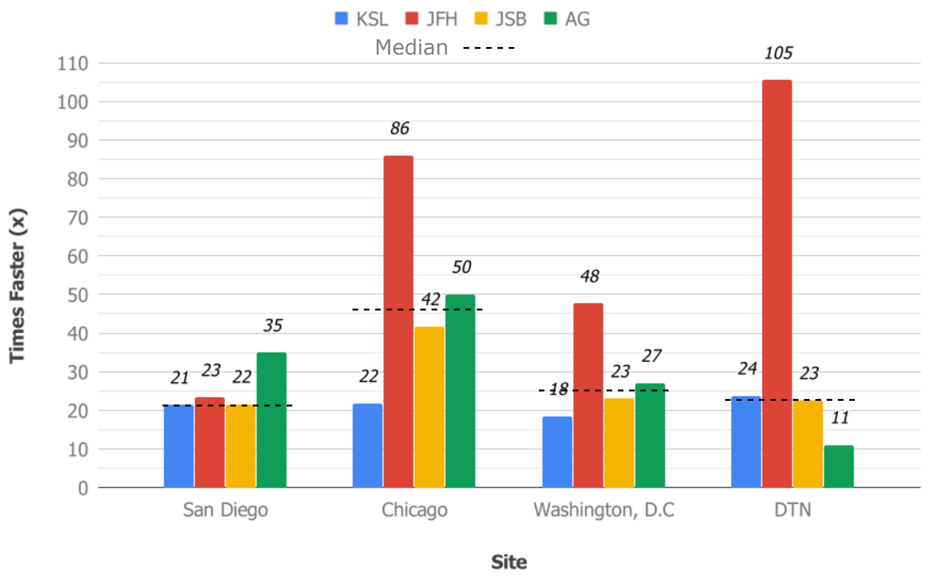


Figure 5.9: Speedup factors at different sites using exceptions

data transfers that range from the capabilities of the end-system, the cloud storage provider policies, the network infrastructure, the size of the data, the geographic location, the data transfer tool, and many more. Analyzing the optimal combination of parameters is out of the scope of this dissertation but we presented results in [133] where we showed that Amazon S3 and Google Drive provided the best performance overall and the `rclone` data transfer tool provided the best performance to its configurability and back-off mechanism to efficiently and reliably move data to external stores. We also identified that `rclone` outperforms all other cloud transfer applications. Consequently, we have recommended `rclone` to researchers on campus as the tool of choice. Among cloud storage providers, Google Drive has become very popular among researchers both because of its good performance and cost (free unlimited storage). Therefore, the analysis in this dissertation will focus on using `rclone` to transfer data from campus file servers and desktops to Google Drive.

This section explores performance as it relates to the location of the researcher's machine in the campus network and the tool parameters used in the transfer. Two types of *end-nodes* where researchers "live and work" were analyzed: (file) *server nodes* that have substantial computing power, and *desktop nodes* that are more resource constrained. In both cases, nodes are located deep inside the campus network. Similar to the previous experiments, the major problem with data transfers transfers is that the path to the cloud has to go through various security middleboxes.

The Problem of Moving Targets

While in the previous measurements the ESnet sites and the campus Data Transfer Node (DTN) had fixed IP addresses, transferring data to both Amazon S3 and Google Drive is challenging because they implement *moving target defense* practices [134] that dynamically change the destination IP address of the cloud storage system. This situation gets more complicated when the tool used to transfer data can issue multiple parallel connections (e.g. `rclone`) for an individual workflow because it is necessary to obtain the destination IP for every individual connection, deploy an exception for that connection, and finally start the transfer. In addition, for high-end machines that have a large number of CPUs, the likelihood of some connections resolving to the same IP address significantly increases. In such cases, only the source port number selected by the data transfer tool would differentiate the individual connections. Although with Amazon S3 is possible to circumvent this problem by preemptively querying the list of subnets assigned to a particular region (e.g. east-1) [135] and use the returned ranges as part of the OpenFlow rules, other storage systems such as Google Drive do not officially advertise the ranges of public IP addresses used to store files in the cloud.

In order to develop a solution to this problem, we inspected the packet captures from a host pushing data to Google Drive using `rclone` with multiple parallel connections. We observed that, as part of the initialization process, multiple DNS request/reply packets were generated before the actual TCP sockets were created. In fact, the tests showed that in general *the number of DNS calls was equivalent to the number of parallel connections*. By further inspecting contents of the DNS packets, we realized that the `A RECORD` field in the DNS reply packets contained a list of IP addresses offered by the remote storage system to start the transmission; in 99% of the cases, the first IP was always picked by the application.

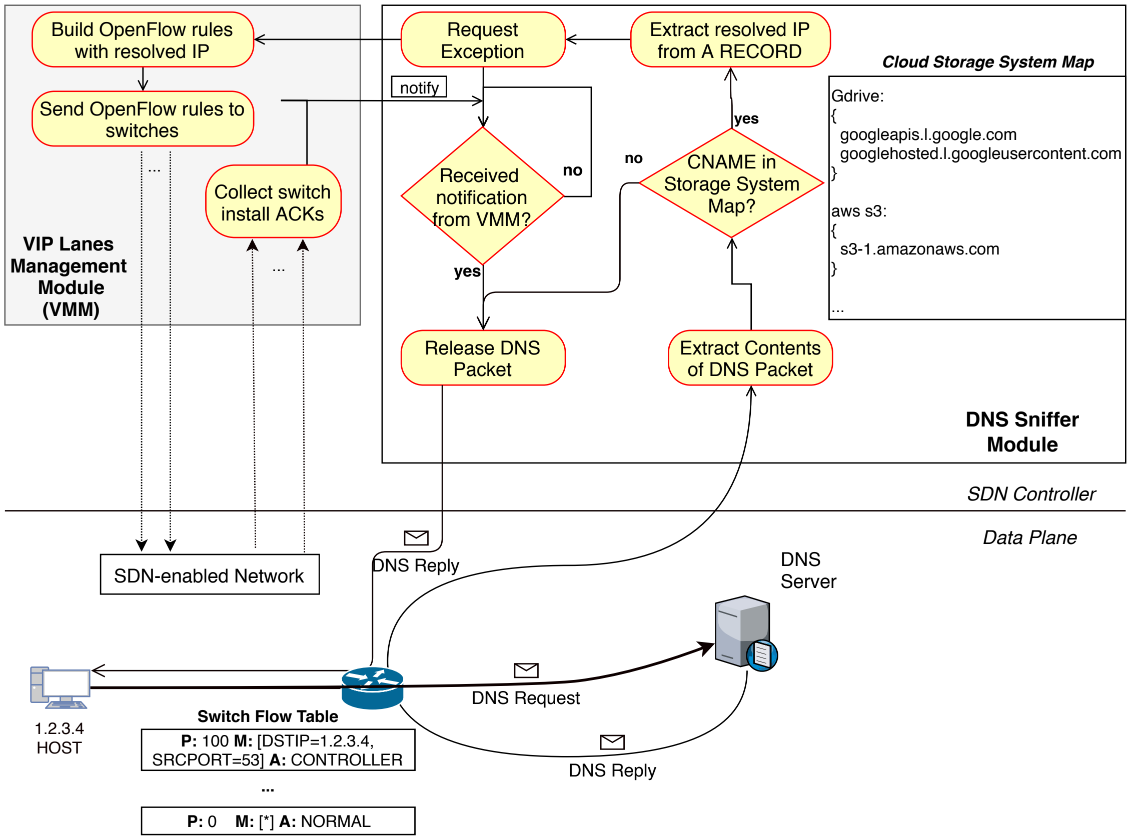

Build OpenFlow rules with resolved IP
Send OpenFlow rules to switches
Collect switch install ACKs
VIP Lanes Management Module (VMM)
Request Exception
Extract resolved IP from A RECORD
notify
Received notification from VMM?
no
yes
CNAME in Storage System Map?
no
yes
Cloud Storage System Map
Gdrive:
{
googleapis.l.google.com
googlehosted.l.googleusercontent.com
}
aws s3:
{
s3-1.amazonaws.com
}
...
Release DNS Packet
Extract Contents of DNS Packet
DNS Sniffer Module
SDN Controller
Data Plane
SDN-enabled Network
DNS Reply
DNS Server
DNS Request
DNS Reply
1.2.3.4 HOST
Switch Flow Table
P: 100 M: [DSTIP=1.2.3.4, SRCPORT=53] A: CONTROLLER
...
P: 0 M: [*] A: NORMAL

Figure 5.10: DNS Sniffer module packet processing to deploy exceptions per connection

A *DNS sniffer module* for the SDN controller was developed to leverage the above findings, consequently, allowing researchers to dynamically deploy exceptions for data transfers to storage systems with no fixed IP addresses. Fig. 5.10 shows all the events and processes required to deploy such "on-the-fly" exceptions.

To start off, whenever there is a request for an exception to a cloud storage system (say "Google Drive"), an "intercept rule" R is added to the flow table of the closest OpenFlow switch connected to the host initiating the transfer. The rule has a short timeout (less than two minutes), a higher priority than any other rule, and instructions to send to the controller all DNS reply packets (source port 53) whose destination address is the one of the host (1.2.3.4 in the figure). Then, the user may initiate the transmission with `rclone` (or any other tool) possibly issuing p parallel connections. As mentioned earlier, the tool generates p DNS request packets that are forwarded to a valid DNS server using the "normal" campus network. Note that it is irrelevant whether the DNS server is local or external (e.g. using Google's DNS 8.8.8.8). Once the DNS server issues a reply, R instructs the switch to send the packet to the controller where its contents are processed by the DNS sniffer module. Upon receipt of the packet, the controller extracts information from the packet such as the Canonical Name (also referred to as `CNAME`) of the destination host and the `A RECORD` containing the list of IP addresses that can be used to reach the remote site. Once the information is stored in memory at runtime, the module must check whether the extracted `CNAME` is in an authorized list of storage system mappings. For example, for an exception to Google Drive the only two accepted `CNAME` values are `googleapis.l.google.com` and `googlehosted.l.googleusercontent.com`. The validation step is necessary since R intercepts *all* DNS responses issued to the host during a two minute window. The DNS responses could possibly include messages sent to services other than the data transfer tool (e.g. a web browser, an SSH connection). For non-data-transfer cases, the DNS reply is simply sent back to the switch and then forwarded to the host for normal processing.

However, if the `CNAME` is included in the authorized list of storage system mappings, then the first IP address in the `A RECORD` is included in the request issued to install an exception to the VIP Lanes Management Module (VMM). The VMM prepares the corresponding OpenFlow rules, stores in a local cache the requested VIP Lane, and submits the rules to the NSCs that will carry the traffic of the exception. In order to prevent a race condition between the installation of an exception and the start of a connection under the “normal” path, the DNS sniffer module does not send the packet back to the host until the exception is successfully deployed in the SDN network (or the VMM confirms there is already an exception with the resolved IP). The rules are considered installed when the DNS sniffer module receives a notification from the VMM. Lastly, the controller releases the DNS response packet back to the host, and the transmission starts making use of the deployed exception path. Note that the extra DNS processing adds an additional delay to the delivery of DNS response packets. However, we observed that processing each DNS packet takes less than 500 milliseconds in the worst case, which is significantly less than the default timeout of 5 seconds applications use to reissue another DNS request to the DNS server.

By pairing `rclone` with an SDN-enabled VIP Lanes campus network and tuning the `rclone` parameters (e.g. number of parallel transfers, chunk size), researchers are able to obtain significantly better throughput from end system nodes, sometimes comparable to the performance from the DTN, even from dedicated server nodes and desktop nodes deep in the campus network to remote storage systems that dynamically change their IP addresses using moving target practices.

5.5.3 Experiment Setup

For this set of experiments two more source nodes were added and are shown in Fig. 5.11 as *Aztec Desktop* and *Flint Server*. The characteristics of the nodes are as follows:

- **Flint Server:** A server machine with a high-speed path (and fewer hops) to the Internet. It has a high-end processor (an Intel(R) Xeon(R) CPU E5-2650L v3) with 48 cores running at 1.80GHz, 180 GB of RAM, and a 10 Gbps network interface with jumbo frames (i.e. MTU 9000) enabled.

- **Aztec Desktop:** A desktop workstation in a computer laboratory running Ubuntu 16.04. The node has an Intel(R) Core(TM) i5-4570S processor with 4 cores running at 2.90GHz, 8 GB of RAM, a 1 Gbps network interface with jumbo frames enabled.

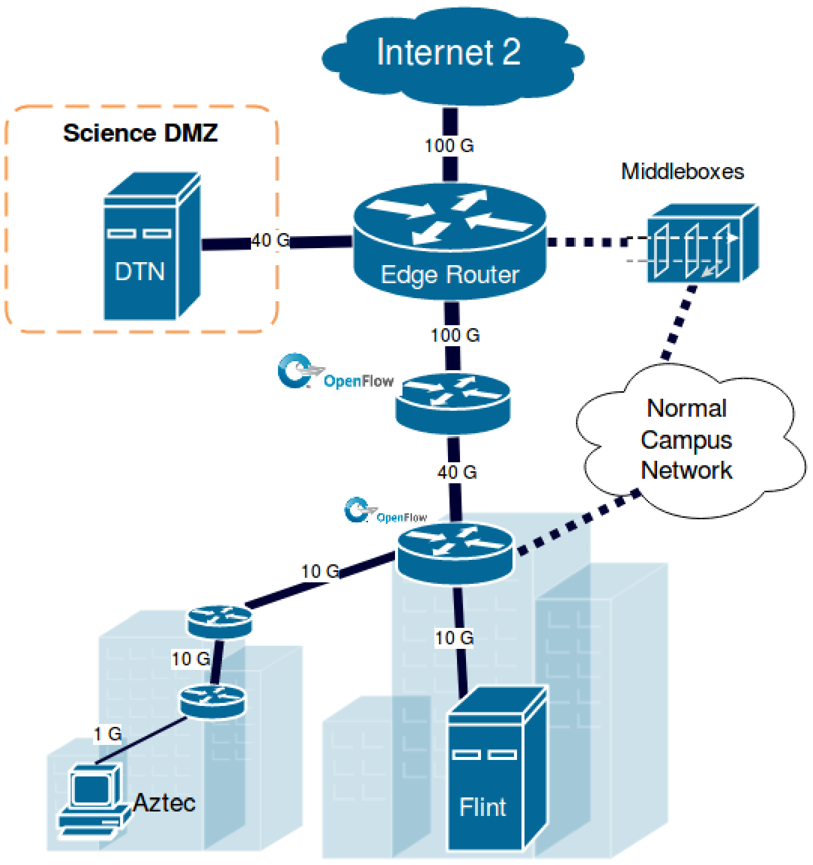


Figure 5.11: Location of source nodes in campus network

The datasets used for the experiment have varying file sizes and number of files. Specifically, three datasets, each consisting of one single file varying in size: 1GB, 10GB, and 100GB. In addition, each of the files was divided into 10 and 50 equally-sized files using the `split` command line utility to obtain six additional datasets, for a total of nine data sets.

The datasets were uploaded from all three locations (i.e. Aztec Desktop, Flint Server, and DTN) to Google Drive 4 times and downloaded from Google Drive to these locations also 4 times. The throughput was recorded once the transmission finished. For every push and pull operation, the numbers of parallel connections (4, 8, 16 or 32) and chunk size parameters (4 MB, 8 MB, 16 MB and 32MB) were changed as well. Due to the 750GB upload limit per account imposed by Google Drive, the tests for the 100 GB data sets were limited to only 16 and 32 transfers.

5.5.4 Results

Viplanes Boost: By enabling high-speed paths for big data science flows, servers can, at least in some cases, achieve speeds close to their maximum capacity and often similar to speeds obtained on the high-end DTN node which are sufficient to move big data to a cloud storage system very quickly. For example, as shown in Table 5.3, some of the measurements recorded speeds greater than 700 Mbps from the Aztec Desktop machine, which is approaching the theoretical maximum of 1 Gbps, and is about 5-7x faster than going through the normal campus network (~100-150 Mbps), and orders of magnitude faster than the speeds recorded by others [136] (~600 KB/s in the best case) when moving data to other cloud storage systems from a campus machine.

Table 5.3: Upload and download speeds by location (in Mbps)

Location-Dir	Mean	Std Dev	Maximum
Aztec Desktop-up	395	159	734
DTN-up	854	903	5664
Flint Server-up	437	381	2164
Aztec Desktop-down	385	176	768
DTN-down	1839	1226	5204
Flint Server-down	1420	799	3986

Chunk Size: When uploading large files, it is often useful to chunk the file into multiple smaller pieces as retransmissions incur in less overhead (in case of retransmission only a small piece is retransmitted). `rclone` allows the user to specify the size of each generated chunk to be loaded in memory by the thread in charge of transmitting the file. Fig. 5.12 shows six summary statistics (i.e. minimum, first quartile, median, mean, third quartile, and maximum) for throughput (in Mbps) for 4 different chunk sizes for the DTN, Flint Server, and Aztec Desktop. Based on the figure, we can observe that the mean values increase with chunk size. Increasing the chunk size, however, had less impact on throughput than increasing the number of parallel transfers. For example, for the DTN, when the the chunk size changes from 4, to 8, to 16 and then to 32 MB, the mean throughput changes from 556, to 727, to 1046 and then to 1062 Mbps, respectively, whereas when the number of parallel transfers changes from 4, to 8, to 16 and then to 32, the mean throughput changes from 425, to 651, to 1077, and then to 1187 Mbps, respectively.

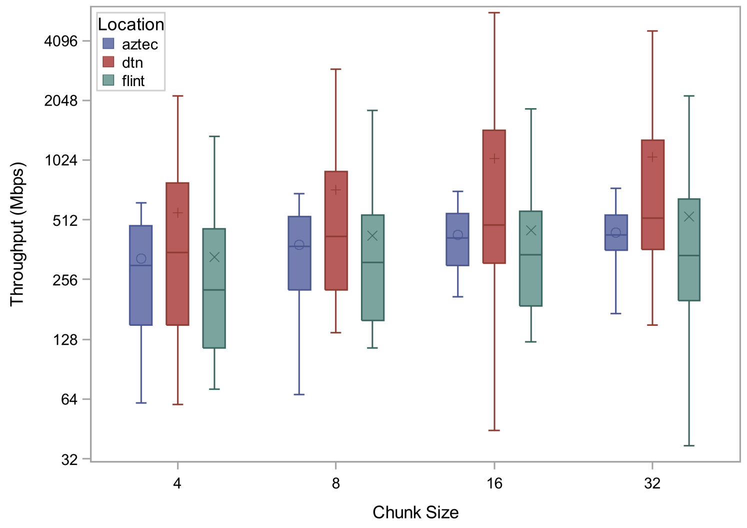
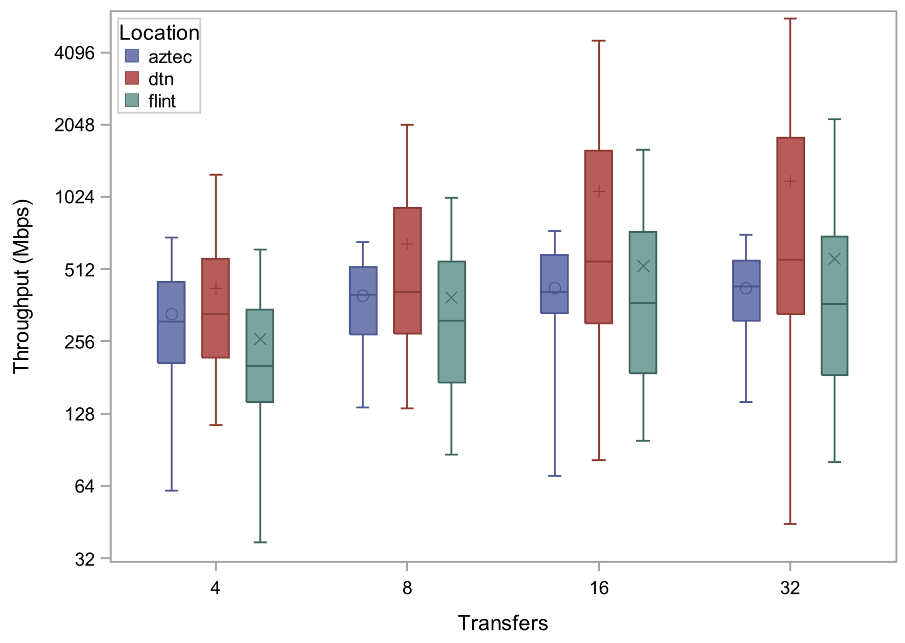
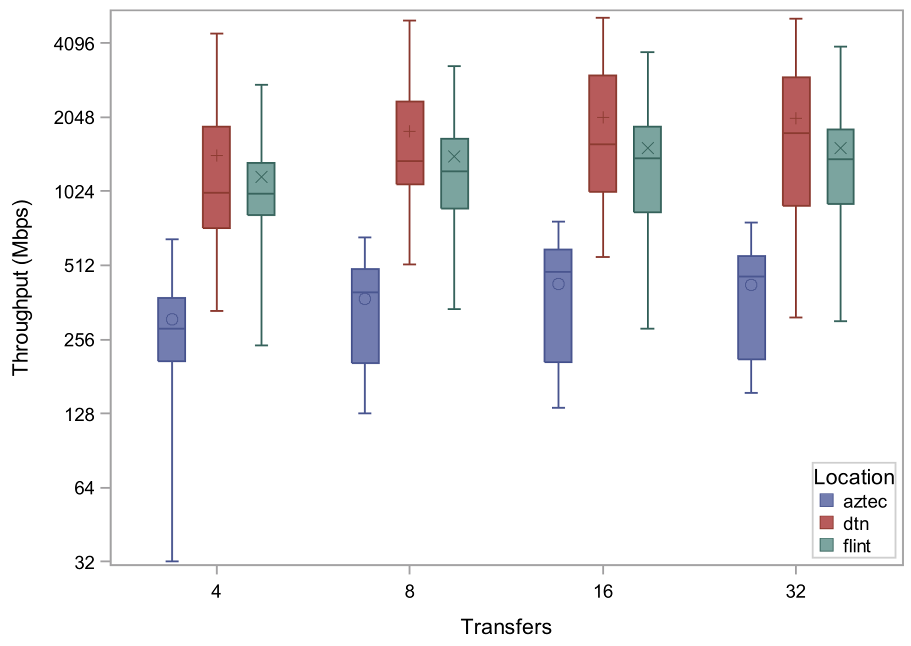


Figure 5.12: Upload speed by chunk size (log scale)

Parallel Connections and Number of Cores: Tools that were able to create multiple parallel connections yielded better performance in all storage systems. The connections parameter is particularly important if one wants to take advantage of the core count found in high-end machines (e.g., the DTN). As seen in Fig. 5.13, using a lower number of threads than the number of cores produces slower speeds for the capabilities of the source node. The influence of this parameter is more noticeable at the DTN and Flint Server for both uploads and downloads. For example, the



(a) Upload



(b) Download

Figure 5.13: Throughput by number of connections (log scale)

maximum throughput from the Flint Server while pushing data to Google Drive measured 617 Mbps, 1013 Mbps, 1608 Mbps, and 2164 Mbps when increasing the number of parallel transfers from 4 to 8 to 16 to 32 respectively. A similar behavior can be seen while analyzing the DTN as both nodes have >= 32 cores.

5.6 Final Remarks

In this chapter we presented a new approach towards security enforcement based on the observation that policy exceptions provide the means for network users and network providers to collaboratively deploy legitimate network configurations tailored to specific workflows. Short-lived, on-demand, fine-grained exceptions provide an opportunity for trusted (and authenticated) users to provide context about their traffic using a structured markup language, and in return get special treatment from network providers. The approach significantly improves on the current manual and time-consuming procedure users have to go through to request a policy exception for their dynamic workflows on campus that allow static exceptions (noting that many do not provide exceptions of any sort). Further, we described how new advances in programmable networks enable the possibility to develop control software that can (1) create trust relationships via delegation of the network flowspace across localized network providers, and (2) automate the deployment of an exception by reconfiguring the network to satisfy the needs of the user. We described a prototype implementation of the security exception mechanism to improve the high-speed big-data transfer in our campus network, what measures were taken to secure the system, and how we were able to lock-down exceptions to individual flows following the principle of least privilege. Our experimental results demonstrate that the transfer rate of trusted users can be improved significantly when on-demand exceptions allow them to bypass middleboxes which can remain as the mechanism to enforce policies on general-purpose traffic.

Chapter 6. Conclusion and Future Work

6.1 Dissertation Summary

This dissertation described systems and concepts that address problems associated with the definition, deployment, and enforcement of network security policies and their exceptions on campus networks. We introduced *PoLanCO*, a human-readable language that allows network operators to write technically-precise statements that are further translated into device configurations without network operator intervention. We presented the building blocks of PoLanCO including the emerging Software-Defined Networking architecture, state-of-the-art decision systems used in business management (Business Rule Management System), and a *Drools translation function* $T(w)$ that transforms human-readable words into valid executable code. We presented a series of examples where operators can write PoLanCO statements derived from imprecise Acceptable Use Policy (AUP) documents. Moreover, we showed that the derived PoLanCO statements can enforce policies at various locations of a campus network. We also described the concept of *Network Security Caps (NSCs)*, a security policy enforcement layer embedded in network devices (e.g. switches, routers) that does policy compliance checks on incoming network traffic prior to regular device packet forwarding. In order to achieve a seamless deployment that does not disrupt ongoing communications, we realized NSCs via OpenFlow-hybrid devices that support the $NORMAL$ port. We showed that NSCs separate policy enforcement from end system and device functionality. Consequently, NSCs can help protect against server misconfigurations that might introduce policy violations and security exploits in the network. Lastly, we proposed a novel approach towards network security based on the notion of trusted *on-demand security exceptions*. We developed a system that brings network providers and users together in order to dynamically adapt the network to user-specific workflows. On one hand, providers get to know what users are doing in the network because users share information about their traffic. On the other, users get a preferred treatment for (some of) their flows when they justify their need for an exception to network providers. Our experiments indicated that short-lived fine-grained security exceptions improve the throughput of researcher workflows by more than an order of magnitude.

6.2 Future Work

The systems introduced in this work to improve the definition, enforcement, and management of network security policies can be extended in the following ways:

PoLanCO Policy Management Extensions: The Policy Language for Campus Operations (PoLanCO) presented in Chapter 3 leverages basic features of the Drools Business Rule Management System to write human-readable and technically-precise statements. At present, PoLanCO statements do not support more complex policy management features such as policy scheduling or distributed policy definition. The language/grammar of PoLanCO can be extended to include additional Drools-specific "rule attributes" that provide enhanced business rule management. For example, the attribute *date-effective* could allow a policy rule to be activated in the future (e.g. scheduling revised student-related policies effective at the start of the next semester); likewise, *date-expires* could be used to specify the times when a policy cannot be active (e.g. forbid all access to a server during maintenance windows); and lastly, the attribute *agenda-group* could be incorporated to control the order of execution of groups of PoLanCO statements (e.g. IT policies are enforced first, then departmental policies, and lastly authorized exceptions).

Reverse Engineering Policies from Configurations: Section 3.4 described the translation pipeline (i.e. set of steps) that transforms an AUP-like document (high-level) into executable Drools code that contacts SDN controllers to push OpenFlow rules that enforce the policy (low-level).

With the introduction of PoLanCO as a middle-layer between low-level configurations and high-level policies, a bottom-up translation becomes feasible and could be used for policy verification. For example, given the current state of all NSCs rules in the network, it could attempt to construct the PoLanCO statements that generated the rules. Then, the reverse engineered PoLanCO statements could be compared with the original statements to verify if there is a discrepancy that could pose a security risk to the network.

Enforcement Beyond OpenFlow: The described prototype systems leverage SDN-capable networks and use OpenFlow as the principal mechanism for policy enforcement. However, not all university campuses have (or allow the deployment of) SDN-enabled equipment in their network infrastructure. Instead, the concepts and systems introduced in this dissertation could be adapted to other types of centralized systems that allow IT groups to push configurations to several network devices. For example, Cisco's Digital Network Architecture [19] uses a controller to troubleshoot, configure, and manage Cisco devices in a network (many campus networks use Cisco equipment across their infrastructure). Likewise, access to Wireless Access Points controllers could extend the capabilities of PoLanCO and on-demand security exceptions to enforce policies on wireless equipment and connections.

Integration of Graph Database Queries: In the systems described in this work, the Neo4j graph database stores network information about the discovered topology. Besides traditional data storage, the prototype exception system executes Neo4j queries that compute three middlebox-free paths (i.e. the widest, the fastest and the shortest paths) for the installation of on-demand exceptions for big data transfers. However, we argue that more queries can be constructed to express various types of policies (or on-demand exceptions) due to the availability of a large amount of information about the network. For example, path queries that avoid specific portions of the network such as finding a path from the machines in a department X to the Internet such that traffic never goes through the medical campus; querying all the nodes that can be reachable on a particular VLAN; or enforcing paths that must go through a series of middleboxes (e.g. Firewalls, IDS/IPS, load balancers, etc.).

Traceability: The introduction of PoLanCO into the process that translates acceptable use policies into configurations allows for the use of natural language processing tools and techniques to trace if an Acceptable Use Policy is represented in the PoLanCO statements.

Appendix A Acronyms

ACL Access Control List

API Application Programming Interface

ARP Address Resolution Protocol

AS Autonomous System

AUP Acceptable Use Policy

BGP Border Gateway Protocol

BRMS Business Rule Management System

BYOD Bring Your Own Device

CIO Chief Information Officer

CLI Command-Line Interface

CTO Chief Technology Officer

CapEx Capital Expenditures

DDoS Distributed Denial of Service

DHCP Dynamic Host Configuration Protocol

DiffServ Differentiated Services

DMZ Demilitarized Zone

DNS Domain Name System

DoS Denial of Service

DPI Deep Packet Inspection

DSCP Differentiated Services Code Point

DSL Domain-Specific Language

DTN Data Transfer Node

ESpec Exception Specification

FTP File Transfer Protocol

GENI Global Environment for Network Innovations

GUI	Graphical User Interface
HPC	High-Performance Computing
HTTPS	HTTP over TLS
HTTP	HyperText Transfer Protocol
IBGP	Internal Border Gateway Protocol
IDPS	Intrusion Detection and Prevention Systems
IDS	Intrusion Detection System
IPSec	IP Security
IPS	Intrusion Prevention System
IS-IS	Intermediate System to Intermediate System
ISP	Internet Service Provider
IT	Information Technology
IT	Information Technology
IoT	Internet of Things
JSON	JavaScript Object Notation
L2TP	Layer 2 Tunneling Protocol
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
LLDP	Link-Layer Discovery Protocol
NAT	Network Address Translation
NFV	Network Function Virtualization
NOS	Network Operating System
NSC	Network Security Cap
NSF	National Science Foundation
OSD	Object Storage Daemon
OSM	Object Storage Monitor
OSPF	Open Shortest Path First

OS	Operating System
OVS	Open vSwitch
OpEx	Operational Expenditures
PBR	Policy-Based Routing
PPTP	Point-to-Point Tunneling Protocol
PWC	Policy Writing Committee
QoS	Quality-of-Service
RADIUS	Remote Authentication Dial-In User Service
RAS	Remote Access Server
RCP	Routing Control Platform
RDP	Remote Desktop Protocol
RIP	Routing Information Protocol
SDK	Software Development Kit
SDN	Software-Defined Networking
SNMP	Simple Network Management Protocol
SSH	Secure Shell
SSL	Secure Sockets Layer
STP	Spanning-Tree Protocol
TLS	Transport Layer Security
ToS	Type of Service
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VLAN	Virtual Local Area Network
VM	Virtual Machine
VPN	Virtual Private Network
WAP	Wireless Access Point

Bibliography

[1] B. Carpenter and S. Brim, “Middleboxes: Taxonoomy and Issues,” Internet Requests for Comments, RFC Editor, RFC 3234, February 2002. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3234.txt

[2] U. of Montana, “web servers policy,” https://www.umt.edu/it/policies/webservers.php, 2013.

[3] B. University, “Server security policy,” https://www.baylor.edu/its/index.php?id=43844, 2005.

[4] P. Grun, “Introduction to infiniband for end users,” *White paper, InfiniBand Trade Association*, 2010.

[5] C. M. University, “Wired/wireless network bandwidth usage guideline,” https://www.cmu.edu/computing/services/endpoint/network-access/guidelines/bandwidth.html, 2017.

[6] M. Yu, J. Rexford, X. Sun, S. Rao, and N. Feamster, “A survey of virtual lan usage in campus networks,” *IEEE Communications Magazine*, vol. 49, no. 7, pp. 98–103, July 2011.

[7] C. Rigney, S. Willens, A. Rubens, and W. Simpson, “Remote Authentication Dial In User Service (RADIUS),” Internet Requests for Comments, RFC Editor, RFC 2865, June 2000. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2865.txt

[8] “Policy governing access to and use of university information technology resources,” *University of Kentucky Regulations*, 2008. [Online]. Available: http://www.uky.edu/regs/sites/www.uky.edu.regs/files/files/ar/ar10-1.pdf

[9] U. of California at Berkeley, “Minimum security standards for networked devices,” https://security.berkeley.edu/minimum-security-standards-networked-devices-mssnd, 2017.

[10] U. of Utah, “Information security policy,” https://regulations.utah.edu/it/4-004.php, 2016.

[11] U. of Houston, “Connecting devices to university networks,” http://www.uh.edu/af/universityservices/policies/mapp/10/100304.pdf, 2018.

[12] L. U. Chicago, “ITS policies and guidelines,” https://www.luc.edu/its/aboutits/itspoliciesguidelines/index.shtml, 2018.

[13] Fail2Ban, “Fail2ban: ban hosts that cause multiple authentication errors,” https://github.com/fail2ban/fail2ban, 2016.

[14] P. Congdon, "Link layer discovery protocol and mib," *V1. 0 May*, vol. 20, no. 2002, pp. 1–20, 2002.

[15] "Ieee standard for local and metropolitan area networks: Media access control (mac) bridges," *IEEE Std 802.1D-2004 (Revision of IEEE Std 802.1D-1998)*, pp. 1–281, June 2004.

[16] C. Hedrick, "Routing information protocol," Internet Requests for Comments, RFC Editor, RFC 1058, June 1988. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1058.txt

[17] J. Moy, "Ospf version 2," Internet Requests for Comments, RFC Editor, RFC 2178, July 1997. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2178.txt

[18] Y. Rekhter, T. Li, and S. Hares, "A border gateway protocol 4 (bgp-4)," Internet Requests for Comments, RFC Editor, RFC 4271, January 2006. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4271.txt

[19] Cisco, "Cisco digital network architecture," https://www.cisco.com/c/en/us/solutions/enterprise-networks/index.html, 2019.

[20] A. Networks, "Aruba controllers: Advanced performance and security," https://www.arubanetworks.com/products/networking/controllers/, 2019.

[21] N. C. Team, "netfilter firewalling, nat, and packet mangling for linux," https://netfilter.org/, 2019.

[22] R. C. L. (Netgate), "pfsense - world's most trusted open source firewall," https://www.pfsense.org, 2019.

[23] T. O. Project, "Openbsd pf - user's guide," https://www.openbsd.org/faq/pf, 2019.

[24] Microsoft, "Windows defender firewall," windows.microsoft.com/it-IT/windows7/products/features/windows-firewall, 2019.

[25] Palo Alto Networks, "Palo alto networks next generation firewalls," https://https://www.paloalto-firewalls.com/, 2019.

[26] SonicWall, "Network security services platform high end firewall series," https://www.sonicwall.com/products/firewalls/high-end/, 2019.

[27] Fortinet, "Fortigate: Next generation firewall," https://www.fortinet.com/products/next-generation-firewall.html, 2019.

[28] R. Bace and P. Mell, "Nist special publication on intrusion detection systems," BOOZ-ALLEN AND HAMILTON INC MCLEAN VA, Tech. Rep., 2001.

[29] R. Sommer, "Bro: An Open Source Network Intrusion Detection System." in *DFN-Arbeitstagung über Kommunikationsnetze*, 2003, pp. 273–288.

[30] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks." in *Lisa*, vol. 99, no. 1, 1999, pp. 229–238.

[31] O. I. S. Foundation, "Suricata: Open source ids / ips / nsm engine," www.suricata-ids.org, 2019.

[32] R. Kent and K. Seo, "Security architecture for the internet protocol," Internet Requests for Comments, RFC Editor, RFC 4301, December 2005. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4301.txt

[33] K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, and G. Zorn, "Point-to-point tunneling protocol (pptp)," Internet Requests for Comments, RFC Editor, RFC 2637, July 1999. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2637.txt

[34] W. Townsley, A. Valencia, A. Rubens, G. Pall, G. Zorn, and B. Palter, "Layer two tunneling protocol," Internet Requests for Comments, RFC Editor, RFC 2661, August 1999. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2661.txt

[35] L. Spitzner, "Honeypots: catching the insider threat," in *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, Dec 2003, pp. 170–179.

[36] T. Benson, A. Akella, and D. Maltz, "Unraveling the Complexity of Network anagement," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 335–348.

[37] N. Feamster, J. Rexford, and E. Zegura, "The Road to SDN: An Intellectual History of Programmable Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014.

[38] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. van der Merwe, "The case for separating routing from routers," in *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, ser. FDNA '04. New York, NY, USA: ACM, 2004, pp. 5–12.

[39] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4d approach to network control and management," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 5, pp. 41–54, 2005.

[40] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 1–12, Aug. 2007.

[41] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed

software defined wan," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[42] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, "GENI: A federated testbed for innovative network experiments," *Computer Networks*, vol. 61, pp. 5–23, 2014.

[43] M. Birk, G. Choudhury, B. Cortez, A. Goddard, N. Padi, A. Raghuram, K. Tse, S. Tse, A. Wallace, and K. Xi, "Evolving to an sdn-enabled isp backbone: key technologies and applications," *IEEE Communications Magazine*, vol. 54, no. 10, pp. 129–135, October 2016.

[44] B. Schlinker, H. Kim, T. Cui, E. Katz-Bassett, H. V. Madhyastha, I. Cunha, J. Quinn, S. Hasan, P. Lapukhov, and H. Zeng, "Engineering egress with edge fabric: Steering oceans of content to the world," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017, pp. 418–431. [Online]. Available: http://doi.acm.org.ezproxy.uky.edu/10.1145/3098822.3098853

[45] P. Kalmbach, J. Zerwas, P. Babarczi, A. Blenk, W. Kellerer, and S. Schmid, "Empowering self-driving networks," in *Proceedings of the Afternoon Workshop on Self-Driving Networks*, ser. SelfDN 2018. New York, NY, USA: ACM, 2018, pp. 8–14. [Online]. Available: http://doi.acm.org/10.1145/3229584.3229587

[46] A. S. Jacobs, R. J. Pfitscher, R. A. Ferreira, and L. Z. Granville, "Refining network intents for self-driving networks," in *Proceedings of the Afternoon Workshop on Self-Driving Networks*, ser. SelfDN 2018. New York, NY, USA: ACM, 2018, pp. 15–21. [Online]. Available: http://doi.acm.org/10.1145/3229584.3229590

[47] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett, "Sdx: A software defined internet exchange," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 551–562. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626300

[48] M. Bruyere, G. Antichi, E. L. Fernandes, R. Lapeyrade, S. Uhlig, P. Owezarski, A. W. Moore, and I. Castro, "Rethinking ixps architecture in the age of sdn," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 12, pp. 2667–2674, Dec 2018.

[49] Y. Xu and Y. Liu, "Ddos attack detection under sdn context," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, April 2016, pp. 1–9.

[50] C. Buragohain and N. Medhi, "Flowtrapp: An sdn based architecture for ddos attack detection and mitigation in data centers," in *2016 3rd International Conference on Signal Processing and Integrated Networks (SPIN)*, Feb 2016, pp. 519–524.

[51] F. M. Facca, E. Salvadori, H. Karl, D. R. López, P. A. A. Gutiérrez, D. Kostic, and R. Riggio, “Netide: First steps towards an integrated development environment for portable network apps,” in *Proceedings of the 2013 Second European Workshop on Software Defined Networks*, ser. EWSDN ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 105–110. [Online]. Available: http://dx.doi.org/10.1109/EWSDN.2013.24

[52] S. Rivera, Z. Fei, and J. Griffioen, “Raptor: A rest api translator for openflow controllers,” in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, April 2016, pp. 328–333.

[53] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, “Pga: Using graphs to express and automatically reconcile network policies,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM ’15. New York, NY, USA: ACM, 2015, pp. 29–42.

[54] D. Comer and A. Rastegarnia, “OSDF: an intent-based software defined network programming framework,” *CoRR*, vol. abs/1807.02205, 2018.

[55] H. P. Enterprise, “Aruba van sdn controller,” https://community.arubanetworks.com/t5/Aruba-Applications/ct-p/ArubaApplications, 2019.

[56] B. Pfaff and B. Davie, “The Open vSwitch Database Management Protocol,” Internet Requests for Comments, RFC Editor, RFC 7047, December 2013. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7047.txt

[57] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, “Network Configuration Protocol (NETCONF),” Internet Requests for Comments, RFC Editor, RFC 6241, June 2011. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6241.txt

[58] “A Simple Network Management Protocol (SNMP),” Internet Requests for Comments, RFC Editor, RFC 1157, May 1990. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1157.txt

[59] Cisco, “Opflex: An open policy protocol white paper,” https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-731302.html, 2014.

[60] P. Shi, S. Rivera, L. Pike, Z. Fei, J. Griffioen, and K. Calvert, “Enabling shared control and trust in hybrid SDN/legacy network,” in *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, July 2019, pp. 1–9.

[61] R. T. Fielding, “Architectural styles and the design of network-based software architectures,” Ph.D. dissertation, 2000, aAI9980887.

[62] grpc, "A high-performance, open-source universal rpc framework," https://grpc.io/, 2019.

[63] O. N. Foundation, "ONF launches SDN northbound interface working group," https://www.opennetworking.org/news-and-events/latest-news/onf-launches-sdn-northbound-interface-working-group/, 2013.

[64] ——, "OpenFlow Switch Specification," https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf, 2012.

[65] H. P. Enterprise, "Hp network protector sdn application," https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c04626978, 2014.

[66] Y. Ben-Itzhak, K. Barabash, R. Cohen, A. Levin, and E. Raichstein, "Enforsdn: Network policies enforcement with sdn," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, May 2015, pp. 80–88.

[67] J. N. Bakker, I. Welch, and W. K. G. Seah, "Network-wide Virtual Firewall Using SDN/OpenFlow," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov 2016, pp. 62–68.

[68] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software Defined Networks," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 1–13.

[69] M. Rezvani, A. Ignjatovic, M. Pagnucco, and S. Jha, "Anomaly-free policy composition in software-defined networks," in *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, May 2016, pp. 28–36.

[70] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul, "Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 19–24. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491203

[71] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486022

[72] A. Lara and B. Ramamurthy, "Opensec: Policy-based security using software-defined networking," *IEEE Transactions on Network and Service Management*, vol. 13, no. 1, pp. 30–42, March 2016.

[73] S. Shin and G. Gu, "Cloudwatcher: Network security monitoring using openflow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?)," in *Proceedings of the 2012 20th IEEE International Conference on Network Protocols (ICNP)*, ser. ICNP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–6.

[74] S. W. Shin, P. Porras, V. Yegneswara, M. Fong, G. Gu, and M. Tyson, "Fresco: Modular composable security services for software-defined networks," in *20th Annual Network & Distributed System Security Symposium.* NDSS, 2013.

[75] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: Towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1384609.1384625

[76] S. A. Mehdi, J. Khalid, and S. A. Khayam, "Revisiting traffic anomaly detection using software defined networking," in *Recent Advances in Intrusion Detection*, R. Sommer, D. Balzarotti, and G. Maier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 161–180.

[77] S. E. Schechter, J. Jung, and A. W. Berger, "Fast detection of scanning worm infections," in *Recent Advances in Intrusion Detection*, E. Jonsson, A. Valdes, and M. Almgren, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 59–81.

[78] M. M. Williamson, "Throttling viruses: restricting propagation to defeat malicious mobile code," in *18th Annual Computer Security Applications Conference, 2002. Proceedings.*, Dec 2002, pp. 61–68.

[79] Y. Gu, A. McCallum, and D. Towsley, "Detecting anomalies in network traffic using maximum entropy estimation," in *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 32–32. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251086.1251118

[80] M. V. Mahoney, "Network traffic anomaly detection based on packet bytes," in *Proceedings of the 2003 ACM Symposium on Applied Computing*, ser. SAC '03. New York, NY, USA: ACM, 2003, pp. 346–350. [Online]. Available: http://doi.acm.org/10.1145/952532.952601

[81] A. Zaalouk, R. Khondoker, R. Marx, and K. Bayarou, "Orchsec: An orchestrator-based architecture for enhancing network-security using network monitoring and sdn control functions," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, May 2014, pp. 1–9.

[82] L. Dridi and M. F. Zhani, "Sdn-guard: Dos attacks mitigation in sdn networks," in *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*, Oct 2016, pp. 212–217.

[83] S. Shirali-Shahreza and Y. Ganjali, "Flexam: Flexible sampling extension for monitoring and security applications in openflow," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 167–168.

[84] C. V. Neu, C. G. Tatsch, R. C. Lunardi, R. A. Michelin, A. M. S. Orozco, and A. F. Zorzo, "Lightweight ips for port scan in openflow sdn networks," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, April 2018, pp. 1–6.

[85] W. Han, Z. Zhao, A. Doupé, and G.-J. Ahn, "Honeymix: Toward sdn-based intelligent honeynet," in *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks &#38; Network Function Virtualization*, ser. SDN-NFV Security '16. New York, NY, USA: ACM, 2016, pp. 1–6. [Online]. Available: http://doi.acm.org.ezproxy.uky.edu/10.1145/2876019.2876022

[86] S. Kyung, W. Han, N. Tiwari, V. H. Dixit, L. Srinivas, Z. Zhao, A. Doup, and G. Ahn, "Honeyproxy: Design and implementation of next-generation honeynet via sdn," in *2017 IEEE Conference on Communications and Network Security (CNS)*, Oct 2017, pp. 1–9.

[87] N. Krawetz, "Anti-honeypot technology," *IEEE Security and Privacy*, vol. 2, no. 1, pp. 76–79, Jan. 2004. [Online]. Available: http://dx.doi.org.ezproxy.uky.edu/10.1109/MSECP.2004.1264861

[88] T. Holz and F. Raynal, "Detecting honeypots and other suspicious environments," in *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, June 2005, pp. 29–36.

[89] Z. Zhao, F. Liu, and D. Gong, "An sdn-based fingerprint hopping method to prevent fingerprinting attacks," *Security and communication networks.*, vol. 2017, pp. 1–12, 2017.

[90] R. van der Pol, B. Gijsen, P. Zuraniewski, D. F. C. Romão, and M. Kaat, "Assessment of sdn technology for an easy-to-use vpn service," *Future Generation Computer Systems*, vol. 56, pp. 295–302, 2016.

[91] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Practical declarative network management," in *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, ser. WREN '09. New York, NY, USA: ACM, 2009, pp. 1–10.

[92] A. Voellmy, A. Agarwal, and P. Hudak, "Nettle: Functional reactive programming for openflow networks," Yale University, Tech. Rep. YALEU/DCS/RR-1431, July 2010.

[93] A. Voellmy, H. Kim, and N. Feamster, “Procera: A language for high-level reactive network control,” in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN ’12. New York, NY, USA: ACM, 2012, pp. 43–48.

[94] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, “Frenetic: A Network Programming Language,” *SIGPLAN Not.*, vol. 46, no. 9, pp. 279–291, Sep. 2011.

[95] N. P. Katta, J. Rexford, and D. Walker, “Logic programming for software-defined networks,” in *Workshop on Cross-Model Design and Validation (XLDI)*, vol. 412, 2012.

[96] M. Reitblatt, M. Canini, A. Guha, and N. Foster, “Fattire: Declarative fault tolerance for software-defined networks,” in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN ’13. New York, NY, USA: ACM, 2013, pp. 109–114.

[97] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, “Real Time Network Policy Checking Using Header Space Analysis,” in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13).* Lombard, IL: USENIX, 2013, pp. 99–111.

[98] C. Monsanto, N. Foster, R. Harrison, and D. Walker, “A compiler and run-time system for network programming languages,” *SIGPLAN Not.*, vol. 47, no. 1, pp. 217–230, Jan. 2012.

[99] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, “A balance of power: Expressive, analyzable controller programming,” in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN ’13. New York, NY, USA: ACM, 2013, pp. 79–84. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491201

[100] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, “Merlin: A language for provisioning network resources,” in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT ’14. New York, NY, USA: ACM, 2014, pp. 213–226.

[101] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, “Kinetic: Verifiable dynamic network control,” in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15).* Oakland, CA: USENIX Association, May 2015, pp. 59–72.

[102] Y. Xia, S. Jiang, T. Zhou, and S.Hares, “Nemo (network modeling) language,” https://tools.ietf.org/html/draft-xia-sdnrg-nemo-language-00, 2015.

[103] Y. Tang, G. Cheng, Z. Xu, F. Chen, K. Elmansor, and Y. Wu, “Automatic belief network modeling via policy inference for sdn fault localization,” *Journal of Internet Services and Applications*, vol. 7, no. 1, pp. 1–13, 2016.

[104] C. M. University, “Web server security guidelines,” https://www.cmu.edu/iso/governance/guidelines/web-server.html, 2014.

[105] R. T. Fielding and G. Kaiser, “The apache http server project,” *IEEE Internet Computing*, vol. 1, no. 4, pp. 88–90, July 1997.

[106] A. Ronacher, “Flask: web development, one drop at a time,” http://flask.pocoo.org/, 2019.

[107] I. Sysoev, “Nginx,” http://www.nginx.com/, 2019.

[108] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford, “The cutting edge of ip router configuration,” *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 1, pp. 21–26, Jan. 2004. [Online]. Available: http://doi.acm.org/10.1145/972374.972379

[109] D. Kreutz, F. M. V. Ramos, P. E. Verssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan 2015.

[110] G. Steinke and C. Nickolette, “Business rules as the basis of an organizations information systems,” *Industrial Management & Data Systems*, vol. 103, no. 1, pp. 52–63, 2003. [Online]. Available: https://doi.org/10.1108/02635570310456904

[111] H. Barringer, K. Havelund, D. Rydeheard, and A. Groce, “Rule systems for runtime verification: A short tutorial,” in *Runtime Verification*, S. Bensalem and D. A. Peled, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1–24.

[112] H. . . AGTIVE 2011 Budapest and A. S. C. Author, *Drools: A Rule Engine for Complex Event Processing*, ser. Applications of Graph Transformations with Industrial Relevance. Berlin, Heidelberg :: Springer Berlin Heidelberg :, 2012, vol. 7233.

[113] U. of Missouri-St. Louis, “Information technology network usage policy,” https://www.umsl.edu/technology/networking/networkpolicy.html, 2019.

[114] W. S. University, “Network security/firewall policy,” https://www.weber.edu/ppm/Policies/10-3_NetworkSecurity.html, 2007.

[115] O. Ben-Kiki, C. Evans, and B. Ingerson, “Yaml ain’t markup language (yaml) version 1.1,” *yaml. org, Tech. Rep*, p. 23, 2005.

[116] Neo4j, “Neo4j graph platform,” https://neo4j.com/, 2019.

[117] R. Daigneau, *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*, 1st ed. Addison-Wesley Professional, 2011.

[118] B. I. S. Office, "Network printer security best practices," https://security.berkeley.edu/education-awareness/best-practices-how-articles/system-application-security/network-printer-security, 2019.

[119] O. College and Conservatory, "Network policy," https://www.oberlin.edu/cit/policies/network-policy, 2019.

[120] OpenStack, "Open source software for creating private and public clouds," https://www.openstack.org/, 2019.

[121] Ceph, "The future of storage," https://ceph.com/, 2019.

[122] H. Zimmermann, "Osi reference model - the iso model of architecture for open systems interconnection," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 425–432, April 1980.

[123] Cisco, "Policy-based routing," https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_pi/configuration/15-mt/iri-15-mt-book/iri-pbr.pdf, 2019.

[124] E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski, "The Science DMZ: A Network Design Pattern for Data-intensive Science," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 85:1–85:10.

[125] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[126] U. of Louisville, "Policy exception process," https://louisville.edu/security/policies/policy-exception-process, 2019.

[127] R. I. of Technology, "Exception process and compliance," https://www.rit.edu/security/content/exception-process-and-compliance, 2014.

[128] S. Rivera, J. Chappell, M. Hayashida, A. Groenewold, P. Oostema, C. Voss, H. Nasir, C. Carpenter, Y. Song, Z. Fei, and J. Griffioen, "Creating complex testbed networks to explore sdn-based all-campus science dmzs," in *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, May 2017, pp. 259–264.

[129] A. Hagberg, P. Swart, and D. Schult, "Software for complex networks," http://networkx.github.io/, 2019.

[130] G. Csardi and T. Nepusz, “The network analysis package,” https://igraph.org, 2014.

[131] “Bandwidth test controller (bwctl),” http://e2epi.internet2.edu/bwctl/, 2019.

[132] U. D. of Energy, “Host tuning,” https://fasterdata.es.net/host-tuning/, 2019.

[133] S. Rivera, J. Griffioen, Z. Fei, M. Hayashida, P. Shi, B. Chitre, J. Chappell, Y. Song, L. Pike, C. Carpenter, and H. Nasir, “Navigating the unexpected realities of big data transfers in a cloud-based world,” in *Proceedings of the Practice and Experience on Advanced Research Computing*, ser. PEARC ’18. New York, NY, USA: ACM, 2018, pp. 22:1–22:8. [Online]. Available: http://doi.acm.org/10.1145/3219104.3229276

[134] R. Zhuang, S. A. DeLoach, and X. Ou, “Towards a theory of moving target defense,” in *Proceedings of the First ACM Workshop on Moving Target Defense*, ser. MTD ’14. New York, NY, USA: ACM, 2014, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/2663474.2663479

[135] A. W. Services, “Aws ip address ranges,” https://ip-ranges.amazonaws.com/ip-ranges.json, 2019.

[136] P. Shen, K. Guo, and M. Xiao, “Measuring the QoS of Personal Cloud Storage,” in *Fifth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, July 2014, pp. 1–6.

Vita

- **Personal Information**
  - **Name:** Sergio Andres Rivera Polanco
  - **Birth place:** Bogota, Colombia
- **Education**
  - University of Kentucky, Lexington, KY
    M.S., Computer Science (*en-passant*) awarded in May 2017
  - Pontifical Xaverian University, Bogota, Colombia
    B.S., Computing & Mathematics awarded in March 2012
- **Employment History**
  - Network Engineer Intern. Google LLC.
    Sunnyvale, CA, USA. May 2018–August 2018
  - Graduate Research Assistant. University of Kentucky.
    Lexington, KY, USA. August 2014–July 2019
  - IT Security Administrator. Banco Davivienda.
    Bogota, Colombia. April 2013–July 2013
- **Scholastic and Professional Awards**
  - Student Travel Awards
    - ACM PEARC, 2019 & 2017
    - ACM SOSR, 2018
    - ACM/IEEE SC, 2017
    - Large Scale Networking Workshop on Operationalizing SDN, 2017
    - CMD-IT Academic Careers Workshop, 2017
  - Best Graduate Poster Award
    3rd Annual Society of Postdoctoral Scholars Research Symposium, University of Kentucky, 2017
  - Duncan E. Clarke Memorial Innovation Award, Department of Computer Science, University of Kentucky, 2016.
  - Thaddeus B. Curtz Memorial Scholarship, Department of Computer Science, University of Kentucky, 2016
  - Best Demo Award
    IEEE International Workshop on Computer and Network Experimental Research Using Testbeds, 2016 & 2015

– Scholarship for Academic Excellence
  Pontifical Xaverian University, 2010 & 2011

- **Publications**

1. **S. Rivera** et. al., 2019. Expressing and Managing Network Policies for Emerging HPC Systems. In *Practice and Experience in Advanced Research Computing (PEARC 19)*, July 28-August 1, 2019, Chicago, IL, USA. ACM, New York, NY, USA, 7 pages.
2. Shi P., **S. Rivera**, et al.,"Enabling Shared Control and Trust in Hybrid SDN/Legacy Networks,". In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, Valencia, Spain,2019.
3. J. Griffioen, Z. Fei, **S. Rivera**, et al., "Leveraging SDN to Enable Short-Term On-Demand Security Exceptions,". In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, Arlington, VA, USA, 2019, pp. 13-18.
4. **S. Rivera**, et al. 2018. Navigating the Unexpected Realities of Big Data Transfers in a Cloud-based World. In *Proceedings of the Practice and Experience on Advanced Research Computing (PEARC '18).* ACM, New York, NY, USA, Article 22, 8 pages.
5. M. Hayashida, **S. Rivera**, J. Griffioen, Z. Fei, and Y. Song. 2018. Debugging SDN in HPC Environments. In *Proceedings of the Practice and Experience on Advanced Research Computing (PEARC '18).* ACM, New York, NY, USA, Article 7, 8 pages.
6. J. Griffioen, K. Calvert, Z. Fei, **S. Rivera**, et al., "VIP Lanes: High-Speed Custom Communication Paths for Authorized Flows,". In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, Vancouver, BC, 2017, pp. 1-9.
7. **S. Rivera**, et al. 2017. Dynamically Creating Custom SDN High-Speed Network Paths for Big Data Science Flows. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact(PEARC17).* ACM, New York, NY, USA, Article 59, 4 pages.
8. **S. Rivera** et al., "Demo abstract: Enabling high-throughput big data transfers across campus networks,". In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Atlanta, GA, 2017, pp. 954-955.
9. **S. Rivera** et al., "Creating complex testbed networks to explore SDN-based all-campus science DMZs,". In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Atlanta, GA, 2017, pp. 259-264.
10. **S. Rivera**, Z. Fei and J. Griffioen, "RAPTOR: A REST API translaTOR for OpenFlow controllers,". In *2016 IEEE Conference on Computer*

Communications Workshops (INFOCOM WKSHPS), San Francisco, CA, 2016, pp. 328-333.

11. Griffioen J. et al. (2016) The GENI Desktop. In: McGeer R., Berman M., Elliott C., Ricci R. (eds) The GENI Book. Springer, Cham **(Co-author)**

12. **S. Rivera**, Z. Fei and J. Griffioen, "Providing a High Level Abstraction for SDN Networks in GENI,". In *2015 IEEE 35th International Conference on Distributed Computing Systems Workshops*, Columbus, OH, 2015, pp. 64-71.