Policy Conflict Management in Distributed SDN Environments

by

Sandeep Pisharody

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved May 2017 by the
Graduate Supervisory Committee:

Dijiang Huang, Chair
Gail-Joon Ahn
Violet Syrotiuk
Adam Doupé

ARIZONA STATE UNIVERSITY

August 2017

ABSTRACT

The ease of programmability in Software-Defined Networking (SDN) makes it a great platform for implementation of various initiatives that involve application deployment, dynamic topology changes, and decentralized network management in a multi-tenant data center environment. However, implementing security solutions in such an environment is fraught with policy conflicts and consistency issues with the hardness of this problem being affected by the distribution scheme for the SDN controllers.

In this dissertation, a formalism for flow rule conflicts in SDN environments is introduced. This formalism is realized in Brew, a security policy analysis framework implemented on an OpenDaylight SDN controller. Brew has comprehensive conflict detection and resolution modules to ensure that no two flow rules in a distributed SDN-based cloud environment have conflicts at any layer; thereby assuring consistent conflict-free security policy implementation and preventing information leakage. Techniques for global prioritization of flow rules in a decentralized environment are presented, using which all SDN flow rule conflicts are recognized and classified. Strategies for unassisted resolution of these conflicts are also detailed. Alternately, if administrator input is desired to resolve conflicts, a novel visualization scheme is implemented to help the administrators view the conflicts in an aesthetic manner. The correctness, feasibility and scalability of the Brew proof-of-concept prototype is demonstrated. Flow rule conflict avoidance using a buddy address space management technique is studied as an alternate to conflict detection and resolution in highly dynamic cloud systems attempting to implement an SDN-based Moving Target Defense (MTD) countermeasures.

ॐ तत् सत्

To my daughter Gayathri, my pride and joy ...

for making it all worthwhile.

कर्मण्येवाधिकारस्ते मा फलेषु कदाचन।
मा कर्मफलहेतुर्भूर्मा ते सङ्गोऽस्त्वकर्मणि॥

You have a right to perform your prescribed duties, but you are not entitled to the fruits of your actions. You should never be motivated by the results of your actions, nor be attached to inaction.

— *Srimad Bhagavad Gita 2:47*

ACKNOWLEDGMENTS

First, I would like to express my sincere gratitude to my advisor Dr. Dijiang Huang for years of experienced guidance, constructive feedback, broad knowledge and unending help; without which this dissertation would not have been possible. To merit his approval means a great deal to me. I am extremely grateful to my committee members, Dr. Ahn, Dr. Syrotiuk and Dr. Doupé for their insightful comments, and critical introspection of my work, which helped me better analyze and improve my work.

I want to thank my collaborators Abdullah Alshalan, Ankur Chowdhary, Adel Alshamrani, Janakarajan Natarajan and Iman El Mir for broadening my research horizons. Thank you to my colleagues in the Secure Networking & Computing (SNAC) lab at ASU - Dr. Chun-Jen (James) Chung, Dr. Qiuxiang Dong, Duo Lu, Dr. Bing Li, Yuli Deng, Oussama Mjihil, Bhakti Bohara, Fanjie Lin, Zhen Zeng, Dr. Huijun Wu, and Dr. Zhijie Wang - for substantive discussions and for providing an environment that was enjoyable to be a part.

Most of all, my profound gratitude goes to my family for being a pillar of support through this work, and my life in general. They have been an enduring source of inspiration for me to pursue a doctorate, making incredible sacrifices over many years so I might someday have this privilege. My parents showered me with love, put me through the best education possible and were the perfect role models for me. My brother showed me the value of hard work, which was something that came in handy these past few years. My daughter, who unfailingly brought a smile on my face, and helped me put things into perspective. My lovely wife and muse Shuchi provided me with crucial inspiration, encouragement when I was down and admonishment when I was slacking. She shouldered far more than her fair share of the parenting and household responsibilities, while I immersed myself in pursuit of this degree. Thank

you for being so understanding. The loss of our precious time together was the most painful thing of all.

Thank you to all the scientists that came before, who learned not to accept the status quo and question everything. Thank you to the ones in uniform, for putting it all on the line, so I can sit in the peace and comfort of my home and ponder. Thank you to my buddy Herbie (who's a good boy, Herbie?) who kept me constant company, sleeping at my feet as I worked. Thank you for coffee, football (Nebraska Cornhuskers and Green Bay Packers), Matt Drudge, coffee, RCP, Reddit, coffee, Netflix, Stephen Colbert, and coffee; for helping me maintain some sort of semblance of sanity during the last few years.

Finally, thank you to the NSF CyberCorps®: Scholarship For Service (SFS) program (NSF-SFS-1129561) for funding my studies and providing me with a stipend for the last three years making my education possible. Disclaimer - The views and conclusions contained in this document are my own, and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

I want to acknowledge how blessed I have been with the grace of God. Anything that I have desired in life, I have had an opportunity to attain. I hope to keep striving to prove myself worthy.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

Pervasiveness of Internet has resulted in nearly 40% of the global population using this technology to revolutionize the way we do business, socialize, gain knowledge, and entertain ourselves [1]. This widespread proliferation of the Internet has driven the need for mechanisms to secure our sensitive data and communication. It has been obvious for a while that security has no single solution, and defense in depth is a strategy that has long been in use [2]. Strong security controls coupled with audit, administrative reviews, and an effective security response plan is the only way anyone can achieve a holistic defense. An oft ignored but essential component of a security infrastructure is ensuring that devices are operating the way the administrators expect them to. For example, consider a firewall. These are foremost amongst the multitude of security devices that have earned their place in our networks. For firewalls to be an effective component of the security mechanism, active management of the firewalls, prevention of conflicts and achieving consistency between the corporate security policy and the implemented firewall rules are all crucial.

Information security often associates itself with three primary objectives [3]: *a*) confidentiality or secrecy relates to measures undertaken to ensure information is not accessible to the unauthorized entities; *b*) integrity, which is concerned with maintaining the consistency, accuracy, and trustworthiness of information over its life cycle; and *c*) availability, which involves ensuring that authorized entities have access to the information as designed. While the underlying network and technologies used to deliver the information changes, the objectives of information security remain intact. This has been true even after the advent of Software-Defined Networking (SDN) in the mid-2000s.

SDN is a transformative approach to network design and implementation, based on the premise of separating the control of network functions from the network devices themselves (switches, routers, firewalls, load balancers, etc.). By separating the control and the data planes, the goal was to unleash the true capability of software in managing networks and removing the constraints placed on them by hardware. Using the OpenFlow protocol, SDN switches can leverage the flexibility afforded by the ability to access header information from several layers of the Open Systems Interconnection (OSI) stack, allowing it to satisfy functionalities traditionally fulfilled by a multitude of physical devices. Along with the SDN support of programmable network interfaces, this flexibility makes SDN an ideal platform for multi-tenant data center deployments that require flexibility and dynamism in configuration and deployment. This is especially true in an Infrastructure-as-a-service (IaaS) cloud where Virtual Machines (VMs) are managed by tenants seeking technological and financial flexibility. Adoption of Bring Your Own Device (BYOD) architectures in modern enterprise environments adds further dynamicity and topology changes.

The decoupling of data and control planes in SDN brings about scalability concerns owing to potential bottlenecks at the controller. Studies suggest that although a centralized controller can scale for a respectable enterprise network, it would fail for a data center deployment [4, 5]. While researchers have explored architectures for decentralizing the SDN architecture [6, 7, 8, 9] they do not completely address flow rule management across this environment.

The flexibility and programmability of SDN allows for the ability to respond rapidly to changing user and security requirements and empowers users in a shared tenant environment to secure their own logical infrastructure in a perceivably private manner. Any security implementation by the tenant such as Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS), Deep Packet Inspection (DPI), Virtual

Private Networks (VPN), Moving Target Defense (MTD) etc., would be accomplished by installing new flow rules in the SDN-based environment. However, the shared control plane leaves open the potential for conflicts between flow rules from different tenants. Further, unlike traditional environments where new rules can get added only through an administrator, abstraction of the data plane from the control plane leads to applications being able to introduce new flow rules into the controller through an API. When done in an adversarial manner or without understanding existing flow rules and the desired security policy, this could result in potential flow rule conflicts. In a decentralized SDN-based environment with multiple controllers, the policy conflict issue is amplified since conflicts could arise due to different controllers not being in sync, and not having the same view of the environment especially in multi-path scenarios. To complicate matters further, a dynamically changing network topology adds its own wrinkles.

Just as firewall conflicts in a traditional network limits effectiveness of a security infrastructure [10], conflicts between flow rules on the controller limits the effectiveness and impact of a security implementation in an SDN-based environment[1]. Amongst issues that are heightened in an SDN-based cloud environment are issues caused by flow rule chaining, partial matches and by `set-field` actions.

Substantial research has attempted to address the problems brought forth above, significant amongst which are FortNOX [11] and the FlowGuard [12] framework. While they deal effectively with direct flow violations, they do not tackle conflicts across addresses over multiple layers. For instance, consider a multi-tenant SDN-based environment. Often, tenants use flat layer-2 topologies due to latency concerns, and the ability to conduct inline promiscuous monitoring using layer-2 devices [13]. A

[1]SDN-based environments are typically prevalent in multi-tenant data center environments or public cloud deployments. Since architectures of both of these have several common features, cloud environments and multi-tenant data centers are used interchangeably in this dissertation.

natural extension would be to implement layer-2 flow rule policies. The data center itself might operate with flow rules based on layer-3 addresses. If different policy enforcement points enforce policies based on addresses in different layers, inconsistent actions could result. Conflicts across multiple OSI layer addresses or *cross-layer conflicts* become severe in an SDN setup where each SDN switch, both physical and virtual, can be considered to be a distributed firewall instance, each with a different local view of the environment and policy.

I. Research Objectives & Contributions

In this dissertation, I first classify all potential conflicts in an SDN-based environment (including the hitherto unstudied cross-layer conflicts [14]). A methodology and implementation of a controller-based algorithm for extracting flow rules in a distributed controller environment, and detecting intra- and inter-table flow rule conflicts utilizing cross-layer conflict checking is detailed. Further, automatic and assisted conflict resolution mechanisms are discussed, and a novel visualization scheme for conflict representation is described. This work is implemented in a security policy analysis framework named Brew, built on an OpenDaylight (ODL) based SDN controller, that effectively scrubs the flow table in a distributed SDN-based cloud environment, highlights and resolves potential conflicts. To summarize, I accomplish the following:

- Extend firewall rule conflict classification in a traditional environment to SDN flow rule conflicts by identifying cross-layer conflicts (which tend to be transient in nature).
- Includes techniques for global prioritization of flow rules in a decentralized environment depending on the decentralization strategy.
- Implement a flow rule conflict detection system in a multiple, decentralized

controller based SDN-based cloud environment, amongst flow rules implementing Quality of Service (QoS) requirements.

- Provide strategies for unassisted resolution of flow rule conflicts, with the recognition that some conflicts may not be resolved without loss of information.
- Present a visualization scheme, implemented to help the administrators view flow rule conflicts graphically.

Finally, to generate conflict free countermeasures in an address hopping based MTD solution, knowledge from the conflict classification information and an address space management module was used. This module was part of a framework called CaCTuS that generates address hopping MTD countermeasures that are provably conflict-free.

II. Dissertation Organization

This dissertation is organized as follows. Chapter 2 discusses background information about security policy management and SDN that is fundamental to understanding the rest of this dissertation. It also discusses related work. Chapter 3 provides details about flow rules, implementing security policies using flow rules, and flow rule management challenges. It then produces a formalism that defines all flow rule conflicts. Cross-layer conflicts and traffic management flow rules are also considered. Next, Chapter 4 discusses distributed SDN controller considerations. Chapter 5 discusses major conflict resolution techniques. These works are consolidated and presented as part of a framework called Brew. Brew is described, analyzed and evaluated in Chapter 6. In Chapter 7 a prototype for generation of conflict-free MTD countermeasure is discussed. Finally, Chapter 8 details conclusions from this dissertation, and lays out a plan for ongoing and future research.

Chapter 2

BACKGROUND AND STATE OF THE ART

In this chapter, background information required to follow concepts covered in this dissertation, specifically security policy management and SDN are described. A brief overview of related works is also presented.

I. Evolution of Security Infrastructures

For over two centuries, the term firewall has been used to describe barriers that prevented or slowed the spread of an undesirable event, with initial use of the word in its literal sense. A lexicographical study of the term will show that the literal meaning of this term has faded from our vocabulary in the current information age. This dissertation uses the term firewall in its new data security connotation. Undoubtedly, firewalls are the most well understood and widely deployed security device in networks with near unanimous adoption.

Over the last few decades, firewalls have evolved from mere routers that separated networks from one another [15] to isolate undesirable events into devices that can conduct a multitude of actions on incoming traffic based on a security policy. Based on general functionality and complexity firewalls can be classified as belonging to five generations: *a*) first generation devices which began with routers that separated out different networks, and evolved into devices with basic packet filtering functions; *b*) second generation of firewalls which started with devices having capability to conduct stateful packet inspections, and moved into devices which perform DPI and application proxy functions; *c*) third generation screened network firewalls which had integrated perimeter security components; *d*) fourth generation firewalls which were essentially distributed firewall implementations; and *e*) fifth generation devices, which

are programmable middleboxes that can serve in a variety of roles, many of which include roles satisfied by traditional firewalls. Figure 2.1 shows this evolution and progression visually.

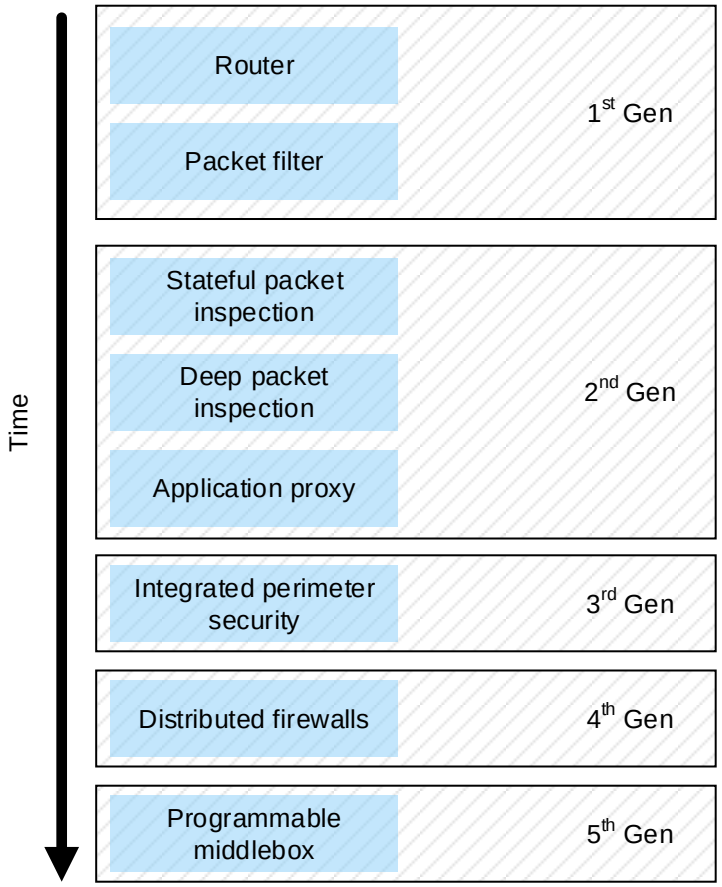

Figure 2.1: Firewall Evolution Timeline.

By design, a firewall is meant to act as a gate of sorts to traffic between the public Internet and private networks, thereby preventing network mishaps. Conventionally, they are thought to: *a*) be at the boundary between two different networks; and *b*) be examining every packet traversing this border, irrespective of whether the traffic inbound, or outbound. Using a set of security policies that have been defined by a

network administrator, this firewall determines which packets to allow, and which to block. These security policies are converted into firewall rules, with each rule consisting of the network 4-tuple (IP Source Address, IP Destination Address, IP Source Port, IP Destination Port), network protocol and associated action.

As data networks grow in size and complexity, their risk tolerance decreases. At the same time, this increasing scale has resulted in specification and management of firewall policies to become complex and an error-prone task. For instance, a centralized firewall located at the ingress into the network may have very strict default policies. If users want to use a service which requires opening new ports on the central firewall, the administrator may refuse to accommodate him/her due to the massive impact of one change. The potential of having to add individual rules for running home grown applications adds complexity to the rule-set, and hence increases the chance of introducing errors. Moreover, the information threat model has evolved to where organizations are aware that many of the vulnerabilities and threats are internal in nature [16]. To counter this new reality, organizations use layered firewalls where internal environments are separated from one another by using firewalls or related security devices to reduce malicious activities originating internally. Such a layered implementation of firewalls requires a separation of security policy controls into trusted, semi-trusted and untrusted networks.

To try and resolve these problems while affording us all the advantages of firewalls, Bellovin [17] proposes the use of distributed firewalls. In his solution, the security policy is still centrally defined. However, enforcing this security policy is the prerogative of distributed nodes (or endpoints, per Bellovin [17]). Such a distributed firewall model requires three components: *a*) a policy language, that states what connections are permitted and prohibited; *b*) a system management tool that can change security policy and enforce it; and *c*) a secure distribution mechanism that can safely distribute

the security policy to all the nodes in the distributed firewall. In his model, a compiler would translate the security policy from the policy language into some internal format. The system management tool, which works in a client-server framework, would then distribute this policy to all the distributed firewall nodes.

In the modern enterprises, it is very common to have multiple firewalls which decentralize the implementation of the desired security policy [10]. And, as suggested by Bellovin [17], the security policy is defined and enforced at different locations. Such a setup allows for the system administrators to remain in control of the security policy, while freeing them from the pain of consistent implementation of the policy across multiple devices. Moreover, with the rise of the cloud computing model, multi-tenant data center environments have witnessed explosive growth. In such environments, the data center provider would have a central security policy, with each tenant potentially having their own additional security measures. Managing and implementing these policies in a consistent manner is challenging.

II. Security Policy Management

Security policy management can be thought of as the framework for specification of an authentication and authorization policy, and the translation of this policy into information that can be used by devices to control access, management of key distribution, audit of security activities and information leakage [18]. This authorization usually pertains to permitting or denying access to resources or information [19]. Security management almost always also includes actions to be taken if any violations are detected.

Given the rapid growth in the scale of networks being deployed, traditional methods which rely on trained personnel to implement and manage information security has become more time consuming, and error-prone [20]. Maintaining a mandated network

security scheme for large scale data center networks and distributed environments is a formidable challenge. It is to this end that several policy-driven management techniques have been suggested [21]. By separating out security policies from their low level implementation and enforcement in the network, such methodologies simplify network management while paving the way for seamless growth of the network [22, 23].

The definition of policy itself is rather ambiguous and is often something of debate. Policies could be thought of as a specific way to dynamically implement static requirements [24]. Separating out the policy from the requirement enables them to be altered and adjusted to the environment or modified to improve performance, all the while ensuring that they still adhere to the requirement. In fact, the requirement can be used as a gauge to verify the functionality of the policy. A policy hierarchy that represents the relationships between different levels of policy abstractions, as shown in Figure 2.2, is generally accepted to be [3, 25, 26, 27]:

- Requirements, high-level abstract policies or management goals: These are generally natural language statements such as Service Level Agreements (SLA) or business goals. They are usually not enforceable at a device level, and are implemented using a lower abstraction level.

- Specification-level or network-level policies: These are specified by a human administrator in a precise format to provide abstractions for device-level implementations. These policies must be specific enough to drive automation.

- Low-level policies or device configurations: These are implemented on the devices themselves. These are often the bottleneck to both scalability, performance and interoperability.

The level of policy most relevant to our study is specification-level policy, or network policy. We adopt the definition of a network policy from the work of Damianou [3].

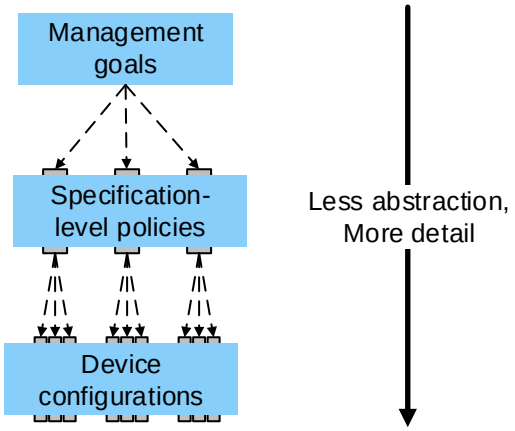

Figure 2.2: Policy Hierarchy.

Definition 2.1. A network policy consists of rules which define relationships between network resources and the network elements that provide those resources. Network policies manage and control the accessibility, reliability and the QoS experienced by networked applications and users.

The Internet Engineering Task Force (IETF) policy model [28] specifies that network policies be considered as rules that specify actions to be taken when certain conditions are met, described by the syntax in Listing 2.1.

Listing 2.1: IETF Network Policy

```
IF <condition(s)> THEN <action(s)>
```

While the syntax described follows the Condition-Action paradigm of most Policy Core Information Model (PCIM) [19] rules, flow rules written in this syntax follow similar semantics to an obligation [3] in the form of an Event-Condition-Action (ECA) paradigm [29] from event-driven architectures, with an implicit event trigger in case of a match. To help determine the action set when multiple conditions are met, most policies are associated with a priority value. Alternately, instead of specifying an

explicit priority, a role-based priority may be assigned to the policies depending on the origination point of the policy.

A typical policy-based management architecture as per the PCIM is shown in Figure 2.3. A Policy Manager (PM), serves to facilitate policy formulation, analysis and verification. Once verified, the policies are stored in a Policy Repository (PR). The Policy Decision Point (PDP) actively monitors the system for specific events. When triggered by certain conditions, the Policy Enforcement Point (PEP) comes into play - to enforce the policy actions.

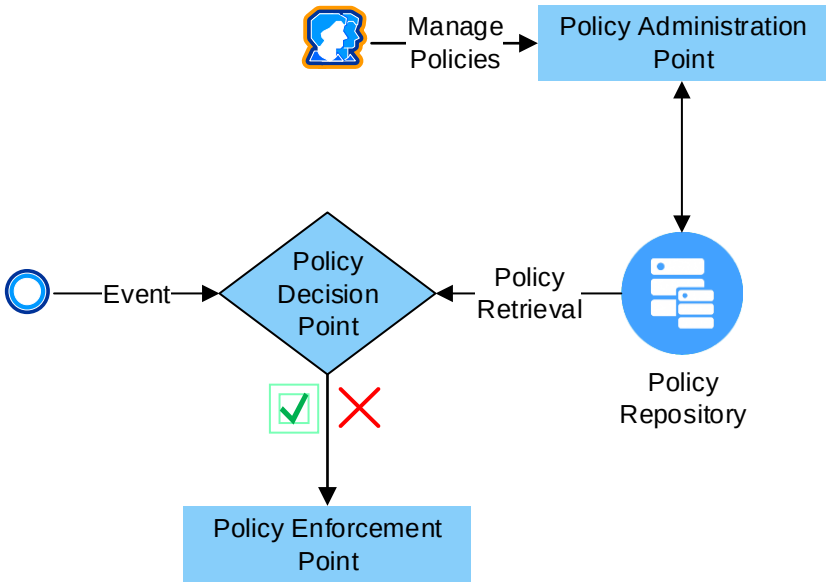

Figure 2.3: Policy Management Framework.

Several issues with regards to policy definition and implementation, such as policy storage and enforcement in a distributed environment are addressed natively in SDN, wherein the SDN controller can act as a PR and PDP, while the SDN switches can act as the PEP. A brief overview on SDN follows.

III. Software-Defined Networks (SDN)

The SDN paradigm is based on the premise that separating control of network functions from the network devices themselves (switches, routers, firewalls, etc.) can address several limitations associated with today's vertically integrated, closed and proprietary networking infrastructure. The adoption of virtualization technologies in computing, and the convergence of voice, video and data communication to IP networks fueled the need for such a shift in networking standards [30]. Figure 2.4 shows a typical network implemented using SDN in a data center environment. Four users have VMs running on the same physical host, with each VM connected to the same Open Virtual Switch (OVS) (described in Section III.B). Data frames that come from the VMs are tagged with a VLAN ID or some other ID based on the tunneling protocol in use, logically separating each of the four users. The OVS then uses flow rules it gets from the SDN controller to determine how to handle the traffic.

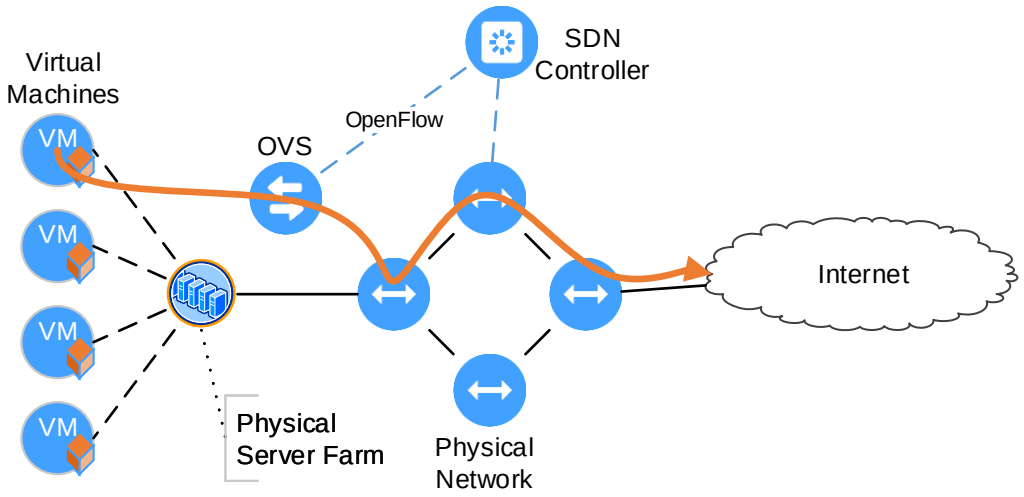

Figure 2.4: Typical Network Implemented Using SDN.

The separation of the control and data planes result in network switches becoming dumb forwarding devices, with control logic being implemented in a centralized[1] controller [31]. This not only allows the network administrators a much finer granularity of control over traffic flow, but also empowers them to respond to changing network requirements in a dynamic environment [32] in a much more effective manner. Figure 2.5 shows a simplified view of the SDN architecture.

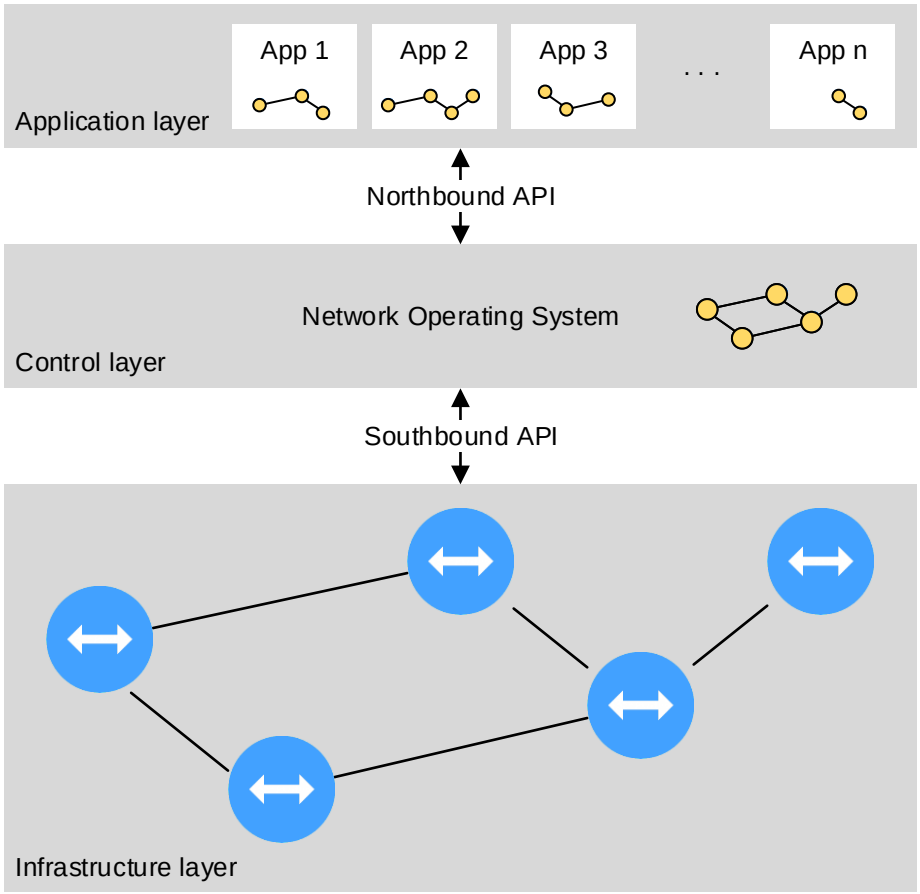

Figure 2.5: Abstraction in SDN.

[1]The controller only needs to be *logically* centralized. This may be implemented in a physically centralized or distributed system [6].

Use of SDN has picked up steam due to the following benefits:

- The traffic patterns culminating from the adoption of cloud systems and big-data computing do not adhere to the traditional notion of a north-south network.

- Separating network control from the hardware devices eliminates the need to configure each device individually. Having a central network policy that can be dispatched to the SDN devices reduces the time-to-deploy thereby enhancing profits for the data center or service providers.

- Since control is separated from the network devices, administrators can modify the behavior of the device by pushing software updates to the device, instead of conducting fork-lift upgrades - once again enhancing profits for data center providers.

- A singular device can handle the functionalities managed by multiple traditional network devices. For example, a single device could do switching, routing, load balancing and security functions. Further, SDN is vendor agnostic, thereby allowing providers more flexibility.

- SDN can organically provide traffic shaping and administer QoS. In current networks, provisioning different QoS levels for different applications is a highly manual process, and can not dynamically adapt to changing network conditions [33].

- SDN provides a layer of abstraction that allows application managers and administrators to dissociate from managing the physical hardware. In addition to having access to virtual disk and memory, SDN virtualizes a network operating system, abstracting the physical topology of the network from the applications.

As shown in Figure 2.5, several applications running on the same physical hardware could have different views of the network.

There are several definitions for a flow in literature. IP Flow Information eXport (IPFIX) [34] describes a flow to be a set of IP packets with a set of common properties passing an observation point in the network during a certain time interval. The typical properties of a flow include the network 4-tuple <source IP, source port, destination IP, destination port>, and the layer-4 protocol. A flow table is a listing of rules managing these flows. Thus, a flow table can, quite simply be thought of as a set of packet filtering rules. The description of a flow, however, needs to be extended to include not just the network 4-tuple, but a network 6-tuple which includes the OSI layer-2 source and destination addresses. Our formal definition of a flow is detailed in Chapter 3, Section II.

The SDN switches communicate with the controller over a secure TCP connection on a dedicated control network, separate from the data transfer network. The switches maintain at least one flow table, consisting of match conditions and associated actions. An ingress packet is matched against the flow table entries to select the entry that first matches (or a different rule selection type as discussed in Chapter 3, Section II) the ingress packet, and the associated instruction is executed. Such an instruction may explicitly direct the packet to another flow table, where the same process is repeated. When processing stops, the packet is processed with its associated action set. Figure 2.6 shows a visual representation of this process.

In addition to being deployed for a variety of traditional functionalities like routing, security and load balancing, SDN can be used for traffic engineering, end-to-end QoS enforcement, mobility management, data center implementation and reducing power consumption. Kreutz et al. [35] groups all these applications into five categories: *a*) traffic engineering; *b*) mobility and wireless; *c*) measurement and monitoring;

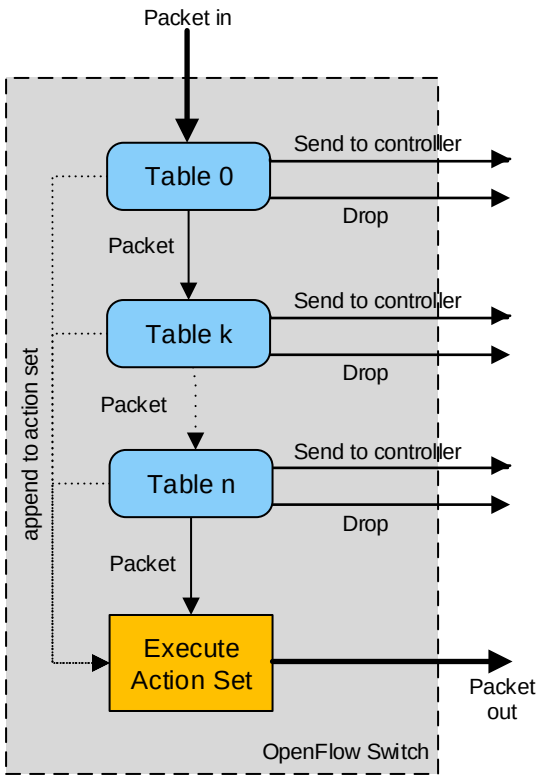

Figure 2.6: OpenFlow Pipeline Processing.

d) security; and *e*) data center networking. This dissertation emphasizes the application of SDN for security, and how shoddy flow rule management could cause security issues.

The programmability and flexibility offered by the SDN paradigm brings about a great potential upside for security processing, primarily because it offers an end-to-end, service-oriented connectivity model that is not bound by traditional routing constraints. Centralized, holistic knowledge of the environment means security policies can consolidate information from a diverse set of devices to deal with various threats. Policy management in SDN can be based on application, service, organization, and geographical criteria rather than physical configuration, thereby insulating security administration from enforcement.

A. *OpenFlow*

OpenFlow, defined by the Open Network Foundation (ONF) [36], is a protocol between the control and forwarding layers of an SDN architecture, and is by far the most widespread implementation of SDN. A basic OpenFlow architecture consists of end hosts, a controller and OpenFlow enabled switches. Note that contrary to the traditional network nomenclature, an OpenFlow *switch* is not limited to being a layer-2 device. The controller communicates with the switches using an OpenFlow API.

When a packet arrives at an OpenFlow switch, packets are processed as follows:

1. A flow table lookup attempting to match the header fields of the packet in question to the local flow table is done. If no matching entry is present, then the packet is sent to the controller for processing. When multiple entries that match the incoming packet are present in the flow table, the packet with the highest priority is picked. Details about the match fields and actions are provided in Chapter 3.

2. Byte and packet counters are updated.

3. Action(s) corresponding to the matching flow rule is(are) appended to the action set. If a different flow table is part of the execution chain, then processing continues.

4. Once all flow tables have been processed, execute the action set.

B. *Open Virtual Switch (OVS)*

Open Virtual Switch (OVS) [37] is open-source implementation of a distributed programmable virtual multilayer switch. OVS implementations generally consist of

flow tables, with each flow entry having match conditions and associated actions. OVS communicates with the controller using a secure channel, and generally uses the OpenFlow protocol. OVS has been widely integrated into major cloud orchestration systems such as OpenStack [38], CloudStack [39] etc., in lieu of the traditional Linux bridge.

Figure 2.7 represents the main components of OVS. The kernel module receives the packets from a NIC (physical or virtual). If the kernel module knows how to handle the packet, it simply follows the instructions. If not, the packet is sent to the `ovs-vswitchd` in userspace using NetLink. This determines how the packet should be handled using the OpenFlow protocol. The `ovs-vswitchd` communicates with a `ovsdb-server` via a socket. The `ovsdb-server` stores OVS configuration and switch management data in JSON format. All functions in the userspace can be accomplished using CLI commands.

OVS and vSwitch are used interchangeably in the remainder of this document.

IV. Related Work

A. Firewall Rule Conflicts

There have been several attempts to classify policy conflicts. Lupu and Morris [40, 41]. Lupu et al. describe conflict analysis for management policies, using a tool to conduct offline detection of conflicts in a large-scale distributed system. Eppstein and Muthukrishnan [42] use a deal with the packet classification and filter conflict detection problem. They use KD tree [43] to verify if two rules apply different actions on the same packet. This misses out on some conflict classification types, that involve sub-optimal rules. Fu et al. [24] manage policies as they apply to IPSec tunnels in both inter- and intra-domain environments. Hari et al. [44] present a conflict detection and resolution algorithm using a k-tuple filter that grows linearly. However,

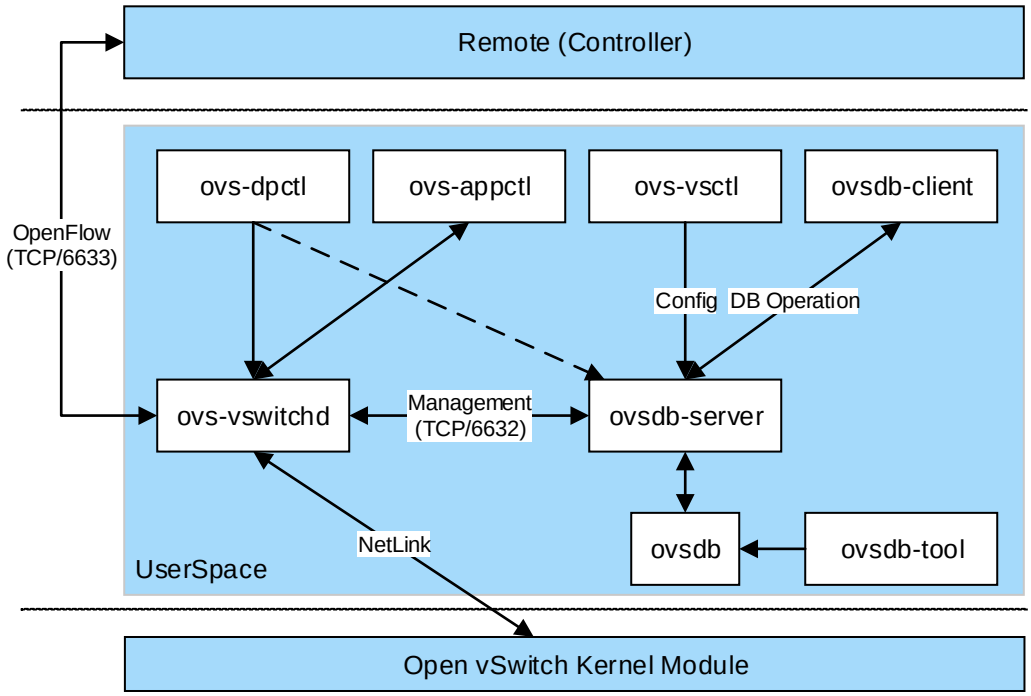

Figure 2.7: Open vSwitch Architecture.

the seminal work by Al-Shaer and Hamed [45] is often used to classify firewall conflicts. In that work, the authors also introduce the Firewall Policy Editor to provide a simple representation allowing for easy human recognition of firewall rule conflicts. The authors extend this work into a distributed environment in Firewall Policy Advisor (FPA) [10], which identifies where to insert new firewall rules to not incur any new policy conflicts. In FPA, the authors also introduce basic visualization by displaying simplified versions of complex firewall rules, and show firewall rule conflicts in tabular format.

Firmato [13] is a firewall management toolkit that helps users separate out security policy and the underlying network topology. It uses a Model Definition Language (MDL) to translate the security model into configuration files for Lucent managed firewalls. Firmato ensures there are no policy conflicts in the system and that the rules in the firewall are up to date with the security policy. Fireman [46] detects

misconfiguration stemming from: *a*) violations of user-specified security policies; and *b*) inconsistencies among firewall rules. It parses firewall configurations and converts them into an operational semantics representation, which is then sent to the administrator for decision making. Unlike FPA, Fireman also recognizes inter-firewall rule conflicts.

FAME [47] is a conflict management environment to detect and resolve conflicts by using rule based segmentation. In FAME, the authors used a matrix to represent conflicting and non-conflicting address segments; but fails while trying to represent larger rule sets.

V. Capretta et al. [48] proposed a formalization of conflict detection for firewalls, but constrained themselves to only look at rules where the actions are different; thereby missing out on some conflict classes. Rei [49] is a language based on deontic logic [50] that defines security policies as possible actions on a resource. All policies in Rei are free of conflicts due to the presence of meta-policies defined by an administrator, which are used to resolve conflicts. If a meta-policy that covers the conflict does not exist, by default the deny action is prioritized. FLIP [51] is a high-level firewall configuration policy language in which security policies are translated into lower level configuration to be loaded onto the devices. Since FLIP is a centralized configuration generation point, the rules generated will be conflict-free due to FLIP preventing overlap of any kind [52].

Fang [53] is a tool that reads in the vendor specific configuration files and converts them into an internal representation, which is then be presented to the administrator in a tabular form in simple text. While it is one of the earliest work in visualization of rule conflicts, it is devoid of any graphics. While it is a step to making security configurations vendor agnostic, it does not display any relation between conflicting rules. The onus is on an experienced administrator to submit the right query that

would present the conflict. PolicyVis [54] used overlapping bars to represent conflict types, and colors to represent the action. However, the conflicts are visible only when a certain scope is defined. A sunburst visualization is used by Mansmann et al. [55] to visualize the rule set, but does not provide any visualization for flow rule conflicts. None of the above works, provide scalable rule conflict visualization that provides high-level conflict categorization, with granular information provided upon need to the administrator.

B. SDN Security & SDN Policy Management

While advances in SDN have made it central to deployment of a cloud environment, security mechanisms in SDN trail its applications. A basic SDN firewall was introduced as part of Floodlight [56], wherein the first packet in a new flow is sent to the controller to be matched against a set of flow rules. The resulting action set is then sent to the OpenFlow switch. The action set is applied for the current flow, and cached for enforcement on all future flows matching the same conditions. In case of a dynamic security policy update to the controller, the OpenFlow switches are oblivious to this situation and could implemented a dated action set. Javid et al. [57] built a layer-2 firewall for an SDN-based cloud environment using a tree topology for a small network using a POX controller and restricted traffic flow as desired. Suh et al. [58] illustrated a proof-of-concept version of a traditional layer-3 firewall over an SDN controller. An application layer firewall using SDN was demonstrated by Shieha [59].

FRESCO [70] allows for the implementation of security services in an OpenFlow environment by providing reusable modules accessible through a Python API. To address conflicts that might arise in an OpenFlow environment, FRESCO introduces a Security Enforcement Kernel (SEK) that prioritizes rules to assist in conflict resolution, but does not tackle complex flow rule inconsistencies. FortNOX [11] is an extension

to the NOX controller that implements role-based and signature based enforcement to ensure applications running on the controller do not circumvent the existing security policy, thereby enforcing policy compliance. In FortNOX, reusable modules are used to protect the flow installation mechanism against adversaries, but conflict analysis is conducted only between a new flow rule and existing rules, without considering dependencies within flow tables. Thus, implementing FortNOX in a distributed environment would be challenging. Decision making in FortNOX seems to follow a least permissive strategy instead of making a decision keeping the holistic nature of the environment in mind. Moreover, it uses only layer-3 and layer-4 information for conflict detection, which we believe is incomplete since SDN flow rules could use purely layer-2 addresses for decision making. In addition, FortNOX would not be able to handle partial flow rule conflicts.

In their work, Cholvy and Cuppens [62] conduct consistency analysis of security policies, focusing on access control policy. Much of their study focuses on defining and ensuring security policy consistency based on deontic logic, like Rei. But paradoxes do exist in deontic logic [63] and hence such work would be beneficial only in a complementary manner to any security implementation involving network devices such as firewalls.

Natarajan et al. [71] study two different conflict detection techniques for flow rules. The first approach internally represents flow rules using a combination radix trie and a hash, which are then used to identify conflicts. The second approach is an ontology based detection system. However, the authors do not discuss conflicts in distributed environments, or how to resolve any conflicts.

FlowChecker [64] identifies intra-switch conflicts within a single flow table using Binary Decision Diagrams (BDD) [65]. Certain conflicts in SDN networks can also be determined by expanding the work of Gu et al. [66] in detecting anomalies in network

traffic tailored to a SDN environment. However, these would be limited to detecting network invariants that can be detected by comparing current network traffic with a baseline distribution.

Pyretic [67], a high-level language written in Python courtesy the Frenetic project [68], allows users to write modular applications. Modularization ensures that rules installed to perform one task do not override other rules. Using a mathematical modeling approach to packet processing, Pyretic compares the list of rules as functions that use a packet as an input, and have a set of zero or more packets as output. Given its mathematical base, Pyretic deals effectively with direct policy conflicts, by placing them in a prioritized rule set much like the OpenFlow flow table. However, indirect security violations or inconsistencies in a distributed SDN environment cannot be handled by Pyretic without a flow tracking mechanism such as the one discussed by Fayazbakhsh et al. [69].

VeriFlow [72] is a proposed layer between the controller and switches which conducts real time verification of rules being inserted. It uses search rules based on Equivalence Classes (ECs) to maintain relationships and determine which policies would be affected in case of a change. Thus, it can verify that flow rules being implemented have no errors due to their dependence on faulty switch firmware, control plane communication, reachability issues, configuration updates on the network, routing loops, etc. Like VeriFlow, NetPlumber [73] sits between the controller and switches. Using header space analysis [74], it ensures that any update to a policy is compared to all dependent policies to prevent and report violations.

FlowGuard [12] is a security tool specifically designed to resolve security policy violations in an OpenFlow network. FlowGuard examines incoming policy updates and determines flow violations in addition to performing stateful monitoring. It uses several strategies to refine policies, most of which include rejecting a violating flow.

Research in SDN security enforcement such as AVANT-GUARD [60] allow for development of security enforcement kernels, and threat detection to applications. In Sphinx [61], the authors extend attack detection in SDN to include a broader class of attacks including untrusted switches and hosts. However, these security solutions implicitly assume the presence of a conflict-free security policy for implementation, and do not address the problem of conflicting flow rules.

Since a flow can be defined using addresses in multiple layers, policy checking approaches in SDN should differ from traditional approaches by being able to consider indirect security violations, partial violations or cross-layer conflicts. However, none of the works discussed above tackle these problems. Moreover, they appear not to fully leverage the SDN paradigm that lets flow rules do traffic shaping in addition to implementing accept/deny security policy. To that end, we propose Brew, a framework that considers cross-layer dependencies while ensuring conflict-free policies in a distributed SDN-based environment. Additionally, Brew analyzes traffic shaping policies including rate limiting policies along with security policies to detect and resolve direct, indirect and partial conflicts.

C. Distributed SDN Environments

Distributed controller environments in SDN are widely studied. Onix [6] facilitates distributed control in SDN by providing each instance of the distributed controller access to holistic network state information through an API. HyperFlow [8] synchronizes the network state among the distributed controller instances while making them believe that they have control over the entire network. Kandoo [75] is a framework tailored for a hierarchical controller setup. It separates out local applications that can operate using the local state of a switch, and lets the root controller handle applications that require network-wide state. DISCO [76] is a distributed control

plane that relies on a per domain organization, and contains an east-west interface that manages communication with other DISCO controllers. It is highly suitable for a hierarchically decentralized SDN controller environment. ONOS [77] is an OS that runs on multiple servers, each of which acts as the exclusive controller for a subset of switches and is responsible for propagating state changes between the switches it controls.

Dixit et al. [9] presented an approach to dynamically assign switches to the controllers in a multiple controller environment in in real-time. The balanced placement of controllers can reduce the cost and the overhead for dynamic assignment of controllers. Bari et al. [78] also presented a technique to dynamically place controllers depending on the changes of number of flows in the network. Controller placement problems have been studied extensively from a performance perspective [4, 79, 78, 80], and based on resilience [81, 82, 83, 84, 85]. Several of these works serve as groundwork for the controller decentralization strategies that are employed during the implementation of the Brew framework.

Chapter 3

FLOW RULE CONFLICTS

I. Flow Rules

OpenFlow v1.3.1 specifications [86] describe a flow rule to consist of the following fields:

- Priority which describes the precedence of the rule, and is defined in the range $[1, 65535]$. Higher priority values are preferred over lower values. If left unspecified, the priority field defaults to 32768.

- Match fields which consist of protocol specific header information, hardware addresses, and metadata that is used to match incoming flows. In all, the basic class in OpenFlow 1.3.1 can match thirty-nine different values, amongst which thirteen values are required to be handled by all switches. These are: *a*) ingress port; *b*) Ethernet source address; *c*) Ethernet destination address; *d*) Ethernet type; *e*) IPv4 source address; *f*) IPv4 destination address; *g*) IPv6 source address; *h*) IPv6 destination address; *i*) IPv4 or IPv6 protocol number; *j*) TCP source port; *k*) TCP destination port; *l*) UDP source port; and *m*) UDP destination port. The complete list of match fields is shown in Table A.1 in Appendix A.

- Packet counters which keep track of the number of packets that utilize the flow rule, and are updated each time a packet match is detected. About forty different counters are specified in the OpenFlow 1.3.1 specification.

- An action set that contains instructions on what to do when a matching flow is detected. Associated actions can: *a*) forward packets through a specified port; *b*) flood the packet on all ports; *c*) change QoS; *d*) encapsulate; *e*) encrypt;

f) rate limit; *g*) drop the packet; and *h*) be customized using various `set-field` actions. The action sets are carried between flow tables, in cases where pipeline processing of flow tables is in effect. The complete list of match fields is shown in Table B.1 in Appendix B.

- Timeouts which specify the maximum amount of time or idle time before a switch would consider the flow rule expired.

- A cookie value chosen by the controller. This value is not visible to the switches, and therefore not used when processing packets. It may however, be used by the controller to filter flow statistics, flow modification and flow deletion.

Since packet counters, timeouts and cookie values are not central to handling flow rule conflicts, in the remainder of this dissertation, we limit discussion of flow rules to priority, match fields and actions set fields. Table 3.1 shows a sample flow table rules with the selected fields present. The data in the table has been written to be human readable. The mapping of the columns is as follows: *a*) Rule #, present only to refer to the rules in this dissertation and not present in OpenFlow; *b*) Priority; *c*) Source MAC, which is specified using the Ethernet source address field; *d*) Destination MAC, which is specified using the Ethernet destination address field; *e*) Source IP, which is specified using the IPv4 source address field; *f*) Destination IP, which is specified using the IPv4 destination address field; *g*) Protocol, which is specified using the IPv4 protocol field; *h*) Source Port, which is specified using either the TCP source field or the UDP source field; *i*) Destination Port, which is specified using either the TCP destination field or the UDP destination field; and *j*) Action, which is specified in the action set but simplified here to just forward and drop. All required fields, ignored in Table 3.1 can be assumed to be wildcarded.

Rule #	Priority	Source MAC	Dest MAC	Source IP	Dest IP	Protocol	Source Port	Dest Port	Action
1	51	*	*	10.5.50.0/24	10.211.1.63	tcp	*	*	forward
2	50	*	*	10.5.50.5	10.211.1.63	tcp	*	80	forward
3	52	*	*	10.5.50.5	10.211.1.0/24	tcp	*	*	forward
4	53	*	*	10.5.50.0/24	10.211.1.63	tcp	*	*	drop
5	54	*	*	10.5.50.5	10.211.1.63	tcp	*	*	drop
6	51	*	*	10.5.50.0/16	10.211.1.63	tcp	*	*	drop
7	55	*	*	10.5.50.5	10.211.1.0/24	tcp	*	1000-1007	drop
8	57	11:11:11:11:11:ab	11:11:aa:aa:11:21	*	*	*	*	*	forward
9	58	*	*	*	*	tcp	*	80	drop

Table 3.1: Flow Table Example.

II. Flow Rule Model

In order to formally create a model that describes flow rules in an SDN-based cloud environment, an address n is defined in Definition 3.4.

Definition 3.1. A frame space of a rule r is the subset of all possible 6-byte hexadecimal numbers representing OSI layer-2 (MAC) addresses, and is expressed as a 2-tuple (ϵ_s, ϵ_d) with subscript s denoting source and d denoting destination addresses.

Definition 3.2. A packet space of a rule r is the subset of all possible 32-bit numbers representing OSI layer-3 (IPv4) addresses, and is expressed as a 2-tuple (ζ_s, ζ_d) with subscript s denoting source and d denoting destination addresses.

Definition 3.3. A segment space of a rule r is the subset of all possible 16-bit numbers representing OSI layer-4 (TCP/UDP) addresses, and is expressed as a 2-tuple (η_s, η_d) with subscript s denoting source and d denoting destination addresses.

Definition 3.4. An address space n of a rule r is the 6-tuple representing the frame space, packet space and segment space, and is expressed as $(\epsilon_s, \epsilon_d, \zeta_s, \zeta_d, \eta_s, \eta_d)$, with subscript s denoting source and d denoting destination addresses. An address space is interchangeably called an address in this dissertation.

If N is the universal set of address spaces, we have:

Definition 3.5. A flow rule r is a function $f : N \to N$ that transforms n to n', where n' is $(\epsilon'_s, \epsilon'_d, \zeta'_s, \zeta'_d, \eta'_s, \eta'_d)$ together with an associated action set a, that can have any of the values from Appendix B. Thus,

$$r := f(n) \rightsquigarrow a$$

The `set-field` capabilities in the action fields of the rules ensures that any, all or none of the fields in n may be modified as a result of the transform function f.

Considering cases where the action set a is a pointer to a different flow table, we can apply the transform function on the result of the original transform function n'. Formally, if $r := f(n) \rightsquigarrow a$; $f(n) = n'$ and $a := g(n') \rightsquigarrow a'$ then,

$$r := g(f(n)) \rightsquigarrow a'$$

Thus, multiple rules applied in succession to the same input address space can simply be modeled as a composite function. It must be noted that the complexity of the flow rule composition function would be exponential in nature, since each flow rule could have multiple actions, each of which themselves could recursively lead to multiple actions.

III. Security Policies using Flow Rules

Due to the ability to alter headers from multiple layers of the OSI stack, flow rules in the OpenFlow protocol can inherently be used for traffic forwarding, routing and traffic shaping. Research has shown that, in addition to traffic manipulation functionalities, most security policies can be transferred into flow entries and deployed on OpenFlow devices [87].

While several security mechanisms implemented in traditional environments depend on routing traffic through middleboxes [88, 89, 90], it has been demonstrated that integrating processing into the network is just as effective [91]. The centralized control in the SDN paradigm makes can make this integration simple and elegant. Models to implement traditional security functions such as firewall rules, an IDS and Network Address Translation (NAT) rules in software have been demonstrated to be successful [92, 93, 94]. SIMPLE, a framework that achieves OpenFlow based enforcement of middlebox policies has been demonstrated in [95]. Contextual meaning to assist in the implementation of middlebox policies using FlowTags was demonstrated in [69].

Further, an OpenFlow based multi-level security system that implements desired security policies using flow rules to accomplish network traffic monitoring as well as verification of packet contents has been successfully implemented [96]. François et al. survey these and several other security implementations using OpenFlow [97] .

Four of the most generic security related policies are firewall, IPS/IDS, load balancing and NAT rules, each of which can be expressed using the flow rule tuple. A typical firewall rule, that blocks all Telnet traffic can be specified in OpenFlow as follows. Note that `nw_proto=6` signifies TCP.

Listing 3.1: Firewall Rule Using OpenFlow.

```
priority=51, nw_proto=6, tp_dst=23, actions=drop
```

Similarly, a load balancer policy, IPS/IDS policy or a NAT policy could be implemented by modifying the layer-3 source or destination address to send the flow to a specific device as follows:

Listing 3.2: Load Balancer Rule Using OpenFlow.

```
priority=51, nw_src=10.5.50.5, nw_dst=10.211.1.1, actions=mod_nw_dst
   =10.211.1.63,output:3
```

IV. Flow Rule Management Challenges

Unlike traditional firewall rules, flow rules can match more than just OSI layer-3 and layer-4 headers making them inherently more complex by virtue of having additional variables to consider. Since wildcard rules are allowed in OpenFlow, a partial conflict[1] of a flow policy could occur, thereby adding complexity to the resolution of conflicting flow rules.

As discussed in Section I, actions that can be applied on a match include forwarding to specific ports on the switch, flooding the packet, changing its QoS levels, dropping

[1]caused when there is a partial overlap in the address spaces of the rules, as described in Section VI

the packet, encapsulating, encrypting, rate limiting or even customizable actions using various `set-field` actions. The `set-field` functionality is a double-edged sword. One the one hand, it provides flexibility and allows the OpenFlow protocol to define complex virtual paths for traffic, and helps assert granular control. Cross-layer interaction is bolstered by virtue of having flow rules using `set-field` actions to change packet headers at several layers dynamically. But it also introduces significant management challenges, such as the origin binding problem [69, 98].

OpenFlow specifications on how a flow match is determined are ambiguous, with the specifications stating that an incoming packet is compared against the match fields in priority order, and the first complete match is selected [86]. However, when the `OFPFF_CHECK_OVERLAP` flag is not set in the controller, multiple flow entries with the same priority can be set, in which case, the selected flow entry is explicitly undefined [86]. This is often the case in multi-tenant data centers, since setting the `OFPFF_CHECK_OVERLAP` flag would result in capping the size of each flow table to $65,535$ entries. When multiple matching rules with the same priority are encountered, directions on how to deal with the issue are unclear, and not standard across different implementations. While some implementations install *sensible* behavior such as more specific flows taking precedence over less specific flows, this is not specified in the OpenFlow specification [86], and not implemented in OVS. For instance, the only constraint OVS places requires flow descriptions to be in normal form, i.e., a flow can specify details for a particular layer header only if the protocol field in its lower layers are populated. That is, if the layer-2 protocol type `dl_type` is wildcarded, indicating use of any layer-3 protocol, then the flow rule can not specify layer-3 IPv4 source and destination addresses. But, this requirement only does not prevent conflict causing scenarios. Furthermore, research has shown that despite there being clear prioritization rules in OpenFlow, certain hardware OpenFlow switches ignore priorities

and treat rules installed later as more important [99]. Needless to say, ambiguity is highly undesirable in any security implementation, and preventing conflicts in flow rules is key.

Security implementations using SDN leverage the ability to make dynamic changes to the network and system configurations to have a lean, agile and secure environment. Since this usually results in environments that are constantly in flux, ensuring synchronization of the flow rules on all the distributed controllers is challenging. As and when the logical topology changes, the flow rules in place must be modified in accordance to ensure policy compliance. Additionally, ensuring that the changing flow rules are always in line with the security policy of the organization is not trivial [100].

Finally, flow rules in an SDN environment can be generated by any number of applications rather than just from an administrator. While this can reduce the workload on the administrator and help with chronic complexity management, there exists a potential for misplaced priorities between some of the flow rule generation points. Besides, an application acting maliciously can wreak havoc across the environment if not detected early enough [101]. Having multiple applications with the ability to concurrently update flow rules can lead to unexpected conditions if the holistic nature of the environment is not considered. For example, consider a load balancing and a DPI application running on the controller. If the DPI detects an intrusion on a node, it would attempt to migrate traffic off it. However, if the load balancer was responsible for allocating new incoming connections to the device with the fewest number of active connections, it might effectively sabotage the attempts of the DPI program.

To summarize, flow rule management is more complex than rule management in a traditional environment because:

- Match conditions cover more fields than in traditional environments.

- The `set-field` actions lead to cross-layer interaction in SDN flows.
- Flow rule priority field is not unique, and there is no standard on how to handle flow rules with the same priority.
- Ensuring synchronicity of rules in a multiple controller environment is not trivial when the topology is constantly changing.
- There are multiple generation points for flow rules, and there exist potential for some of the generation points to not have the same priorities as the administrator.

V. Motivating Scenarios

One of the major benefits of using SDN to implement a cloud environment is the ability to have multiple applications run on the SDN controller, each of which has complete knowledge of the cloud environment. This can be leveraged by the cloud provider to provide Security-as-a-Service (SaaS). A few potential examples of services in a SaaS suite are Firewalls, VPN, IDS, IPS, MTD, etc. Implementing a management system that only specifies security policies without tackling topological interaction amongst constituent members has always been a recipe for conflicts [24].

With the SDN controller having visibility into the entire system topology along with the policies being implemented, several of the conflict causing scenarios in traditional networks were handled. However, there are several instances where conflicts can creep into the flow table such as policy inconsistencies caused by: *a*) service chain processing where multiple flow tables that handle the same flow might have conflicting actions; *b*) VPN implementations that modify header content could result in flow rules being inadvertently being applied to a certain flow; *c*) flow rule injection by different modules (using the northbound API provided by the controller) could have conflicting actions for the same flow; *d*) matching on different OSI layer addresses resulting in

different actions; and *e*) administrator error. This list, while incomplete, goes to show how prevalent policy conflicts in SDN-based cloud environments could be.

Three distinct case studies in an SDN-based cloud environment where the security of the environment is put at risk due to flow rule conflicts are discussed next. The first scenario serves as an example where rules from different applications conflict with each other, and the second scenario serves as an example where rules from a single module might cause conflicts due to the dynamism in the environment. The last scenario once again discusses how inconsistent view of the network state results in different applications inserting flow rules with incomplete information.

A. Case Study 1: Moving Target Defense (MTD)

Traditional approaches to addressing security issues in a dynamic, distributed environment concerned themselves with implementing security through individual components, and not considering security holistically. This leads to two critical weaknesses: *a*) defense against insider attack is minimal; and thus *b*) when perimeter defenses fail, internal systems are ripe for the picking. As a counter, security applications that implement MTD is a topic that is hotly researched.

MTD techniques have been devised as a tactic wherein security of a cloud environment is enhanced by having a rapidly evolving system with a variable attack surface, giving defenders an information advantage [102]. An effective countermeasure used in MTD is network address switching [103], which can be accomplished in SDN with great ease. Since an MTD application could dynamically and rapidly inject new flow rules into an environment, it could lead to conflicts between the new and old flow rules.

In the data center network shown in Figure 3.1, we have Tenant A hosting a web farm. Being security conscious, only traffic on TCP port 443 is allowed into the IP

addresses that belong to the web servers. When an attack directed against host $A2$ has been detected, the MTD application responds with countermeasures and takes two actions: *a*) a new web server (host $A3$) is spawned to handle the load of host $A2$; and *b*) the IP for host $A2$ is migrated to the Honeypot network and assigned to host $Z1$.

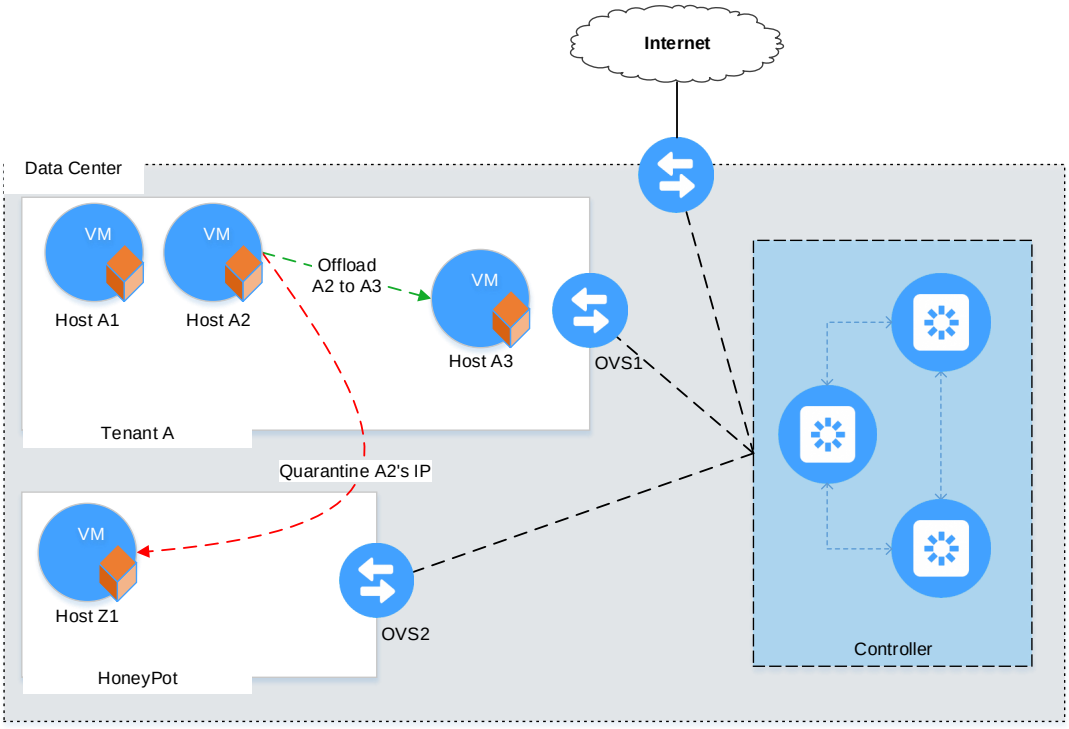

Figure 3.1: Policy Conflicts in SDN-based Cloud Caused by MTD.

To run forensics, isolate and incapacitate the attacker, the Honeypot network permits all inbound traffic, but restricts egress traffic to other sections of the data center. These actions result in new flow rules being injected into the flow table that: *a*) permits *all* traffic inbound to the IP that originally belonged to host $A2$, but now belongs to host $Z1$; *b*) modifies an incoming packet's destination address from host $A2$ to host $A3$ if the source is considered to be a non-adversarial source; *c*) stops all

outbound traffic from the IP that originally belonged to host $A2$, but now belongs to host $Z1$ to the rest of the data center; and d) permits traffic on port 443 to host $A3$. The original policy allowing only port 443 to the IP of host $A2$, and the new policy allowing all traffic to the IP address of host $Z1$ are now in conflict.

B. Case Study 2: VPN Services

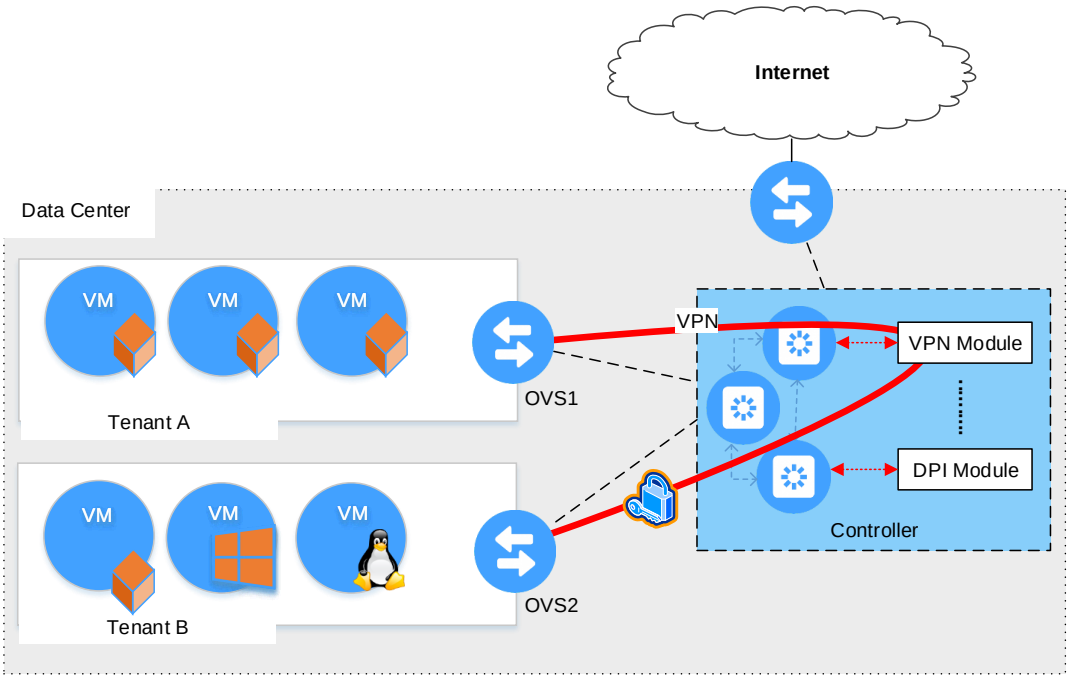

Figure 3.2: Policy Conflicts Caused by Different Applications in an SDN-based Cloud.

In a multi-tenant hosted data center, the provider could have layer-3 rules in place to prevent certain tenants from sending traffic to one another for monetization, compliance or regulatory reasons or even due to technical reasons. Hosts in two different tenant environments, Tenant A and Tenant B, can establish a layer-2 tunnel (either as a host-to-host tunnel or a site-to-site tunnel) between themselves to do single

hop communication or to encrypt communication between them as shown in Figure 3.2. If another application running on the controller inserts policies to implement DPI, all traffic originating from Tenant A destined to Tenant B will be dropped, since they are encrypted and fail the DPI standards. Clearly, there is an inherent conflict between flow rules inserted by different applications running on the SDN controller, leading to a shoddy user experience.

C. Case Study 3: Load Balancing & IDS

As introduced in Section IV and similar to the scenario in Case Study #2, consider an SDN-based data center environment where a load balancing application as well as an IDS application run on the SDN controller. Upon detecting intrusions, the IDS could implement a countermeasure that offloads traffic from the compromised node. However, the load balancing application which routes new connections based on their active load might start redirecting new traffic to the compromised node, since the system would infer that the compromised node has the least amount of load.

VI. Flow Rule Conflicts

A. Problem Setup

When a packet arrives at an OVS, its match fields are compared to the match fields of the rules in the flow table. There are multiple ways a rule could be selected, namely: *a*) First match, where the first rule that matches the specified fields of the packet is selected; *b*) Best match, where the entire firewall rule set is examined to determine the rule that provides the tightest bounds to the specified fields; *c*) Deny take precedence, where any rule with a deny action is automatically preferred over other actions; and *d*) Most/Least specific take precedence, where the rule with the most/least specific match for the match fields [104]. The first match selection is by far

the most prevalent way to select a matching flow rule. In this dissertation, we assume all selections to be based on first match selection, with rules ordered by priority. When multiple rules with the same priority exist, the newest rule has precedence.

B. Conflict Classes

Consider a flow table F containing rule set $\{r_1, r_2, ..., r_n\}$. We can represent a flow rule r_i using the tuple $(p_i, \epsilon_i, \zeta_i, \eta_i, \rho_i, a_i)$, where *a)* p_i is the priority. *b)* ϵ_i is the frame space of the rule. *c)* ζ_i is the packet space of the rule. *d)* η_i is the segment space of the rule. *e)* ρ_i is the OSI layer-4 protocol. *f)* a_i is the action set for the rule.

For all devices, including SDN devices or traditional firewalls, we deal with two main problems:

- Packet Classification Problem: In a firewall with rule set R, for an incoming packet Π_{in} with address 6-tuple n_{in} and protocol ρ_{in}, the packet classification problem [42], seeks to find out the set $R_m \subseteq R$ where $R_m = \{r_i \mid (\, r_i \in R) \wedge (n_i = n_{in}) \ \wedge (\rho_i = \rho_{in})\}$. The problem can be further extended to determine rule $r_x = (p_x, n_x, \rho_x, a_x) \in R_m$ such that $p_x > p_y \ \forall \ r_y \in R_m$.

- Conflict Detection Problem: The conflict detection problem [42] seeks to find rules r_i, r_j such that $r_i, r_j \in R$ and $(n_i = n_j) \wedge (\rho_i = \rho_j) \wedge (a_i \neq a_j \vee p_i \neq p_j)$.

We formally define the set operations on addresses at each OSI layer. Let $\xi \in \{\epsilon, \zeta, \eta\}$ be a 2-tuple (ξ_s, ξ_d) denoting an address at OSI layer-2, layer-3 or layer-4, with subscript s denoting the source address and d denoting the destination address. Then the following definitions apply.

Definition 3.6. $\xi_i \subseteq \xi_j$ if and only if they refer to the same OSI layer, and $\xi_{si} \subseteq \xi_{sj} \wedge \xi_{di} \subseteq \xi_{dj}$.

Definition 3.7. $\xi_i \not\subseteq \xi_j$ if and only if they refer to the same OSI layer, and $\xi_{si} \not\subseteq \xi_{sj} \vee \xi_{di} \not\subseteq \xi_{dj}$.

Definition 3.8. $\xi_i \subset \xi_j$ if and only if they refer to the same OSI layer, and $(\xi_{si} \subset \xi_{sj} \wedge \xi_{di} \subseteq \xi_{dj}) \vee (\xi_{si} \subseteq \xi_{sj} \wedge \xi_{di} \subset \xi_{dj})$.

Definition 3.9. Address Intersection $\xi_i \cap \xi_j$ produces a tuple $(\xi_{si} \cap \xi_{sj}, \xi_{di} \cap \xi_{dj})$ if and only if ξ_i and ξ_j refer to the same OSI layer.

Definition 3.10. Conflict detection problem [42] seeks to find rules r_i, r_j such that $r_i, r_j \in R$ and $(n_i \cap n_j \neq \emptyset) \wedge (\rho_i = \rho_j) \wedge (a_i \neq a_j \vee p_i \neq p_j)$.

Definition 3.11. Flow rule address space $n_i \subseteq n_j$ iff $\epsilon_i \subseteq \epsilon_j \wedge \zeta_i \subseteq \zeta_j \wedge \eta_i \subseteq \eta_j$;

Since flow rules in an SDN-based cloud environment are clearly a super-set of rules in a traditional firewall environment, work on flow rule conflicts are an extension of the work on firewall rule conflicts. While several works have classified firewall rule conflicts [40, 44, 42, 46]; the seminal work by Al-Shaer and Hamed [45] is often used to classify firewall rule conflicts in a single firewall environment. The classifications used in the work of Al-Shaer and Hamed [45] are extended to formally classify flow rule conflicts, and further adapted to suit a distributed environment.

Knowing that OpenFlow specifications clarify that if a packet matches two flow rules, only the flow rule with the highest priority is invoked, the classification of different conflicts in SDN environments are detailed in the remainder of this Section. The conflict classification is visually represented in Figure 3.3 and Figure 3.4. Figure 3.3 shows the address space overlap and flow rule conflicts for rules with different priorities, and Figure 3.4 shows the address space overlap for flow rules with the same priority.

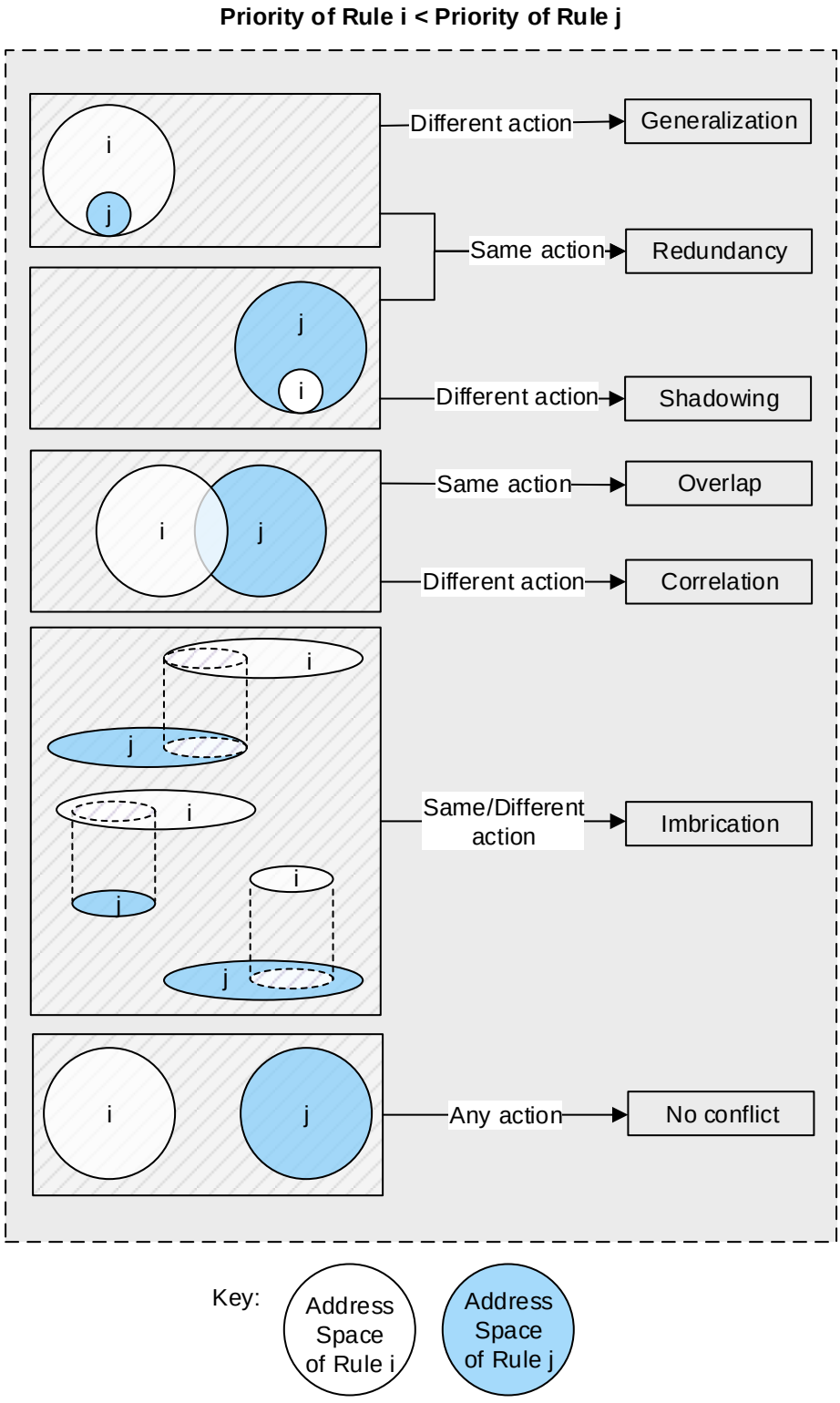

Figure 3.3: Address Space Overlap and Flow Rule Conflicts for Rules with Different Priorities.

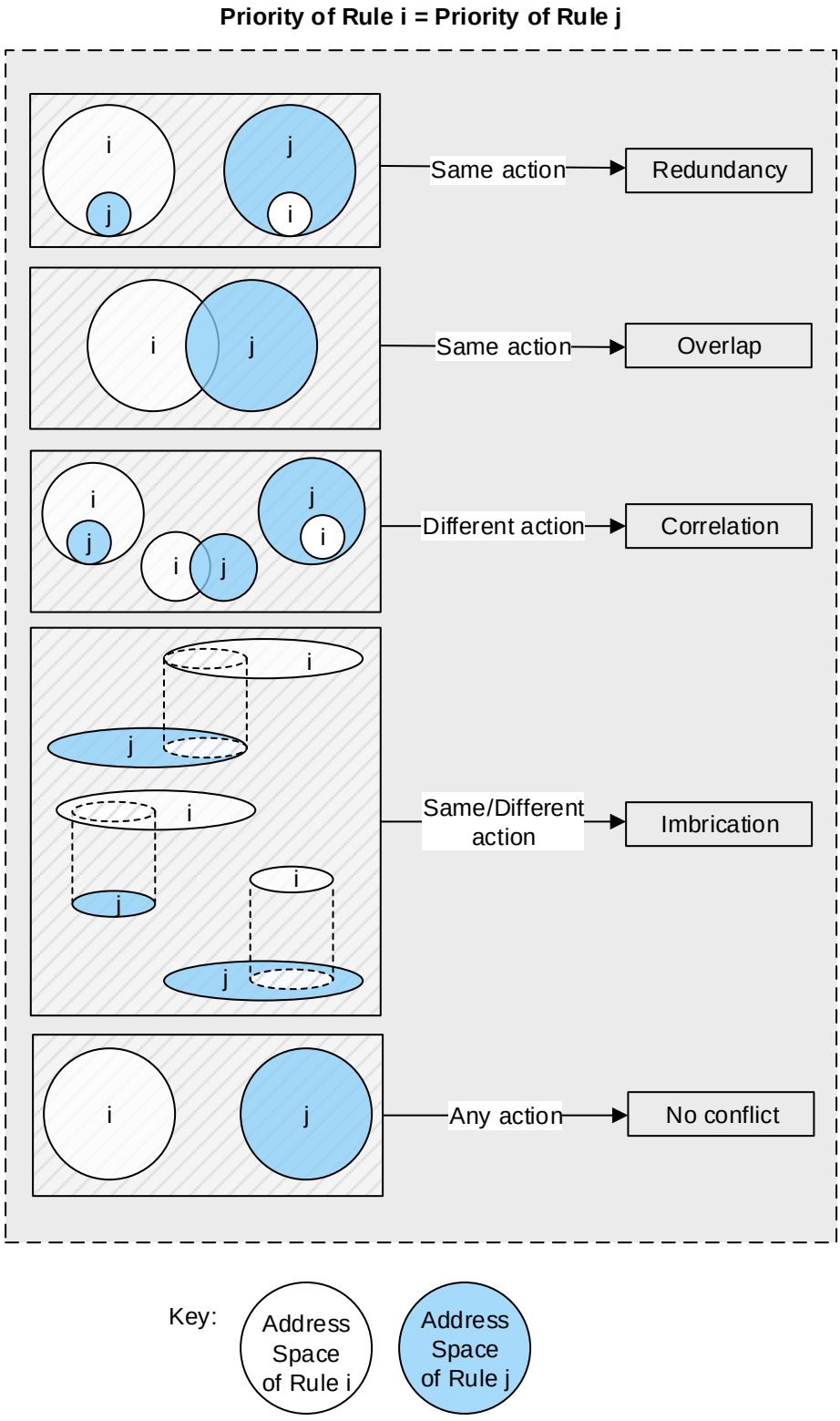

Figure 3.4: Address Space Overlap and Flow Rule Conflicts for Rules with Same Priority.

1) Redundancy: A rule r_i is redundant to rule r_j iff: *a*) address space $n_i \subseteq n_j$; *b*) protocol $\rho_i = \rho_j$; and *c*) action $a_i = a_j$. For example, consider rules 1 and 2 from Table 3.1, shown below for easy reference. Rule 2 has an address space that is a subset to the address space of rule 1, with matching protocol and actions. Hence, rule 2 is redundant to rule 1. Redundancy does not pose a serious issue, but instead, is more of an optimization and efficiency problem.

Listing 3.3: Flow Rules with Redundancy Conflict.

```
<flow_id=1> priority=51, nw_src=10.5.50.0/24, nw_dst=10.211.1.63,
   nw_proto=6, actions=output:3

<flow_id=2> priority=50, nw_src=10.5.50.5, nw_dst=10.211.1.63,
   nw_proto=6, tp_dst=80, actions=output:3
```

2) Shadowing: A rule r_i is shadowed by rule r_j iff: *a*) priority $p_i < p_j$; *b*) address space $n_i \subseteq n_j$; *c*) protocol $\rho_i = \rho_j$; and *d*) action $a_i \neq a_j$. In such a situation, rule r_i is never invoked since incoming packets always get processed using rule r_j, given its higher priority. Shadowing is a serious issue since it shows a conflict in a security policy implementation [45]. For example, rule 4 has the same address space as rule 1, with the same protocol, but conflicting actions. But, the priority of rule 4 is higher than that of rule 1, which results in rule 1 never being invoked. Hence, rule 1 is shadowed by rule 4.

Listing 3.4: Flow Rules with Shadowing Conflict.

```
<flow_id=1> priority=51, nw_src=10.5.50.0/24, nw_dst=10.211.1.63,
   nw_proto=6, actions=output:3

<flow_id=4> priority=53, nw_src=10.5.50.0/24, nw_dst=10.211.1.63,
   nw_proto=6, actions=drop
```

3) Generalization: A rule r_i is a generalization of rule r_j iff: *a*) priority $p_i < p_j$; *b*) address space $n_i \supseteq n_j$; and *c*) action $a_i \neq a_j$. In this case, the entire address space

of rule r_j is matched by rule r_i [45]. As shown below, rule 1 is a generalization of rule 5, since the address space of rule 5 is a subset of the address space of rule 1, with the same protocols, but different actions. Note that if the priorities of the rules are swapped, it will result in a shadowing conflict. In traditional firewall management practices, it was common practice to add such rules for administrators to isolate a smaller portion of the traffic managed separately from a larger set of traffic.

Listing 3.5: Flow Rules with Generalization Conflict.

```
<flow_id=1> priority=51, nw_src=10.5.50.0/24, nw_dst=10.211.1.63,
   nw_proto=6, actions=output:3

<flow_id=5> priority=54, nw_src=10.5.50.0, nw_dst=10.211.1.63,
   nw_proto=6, actions=drop
```

4) Correlation: Classically, a rule r_i is correlated to rule r_j iff: *a*) address space $n_i \not\subseteq n_j \wedge n_i \not\supseteq n_j \wedge n_i \cap n_j \neq \emptyset$; *b*) protocol $\rho_i = \rho_j$; and *c*) action $a_i \neq a_j$ [45]. As shown below, rule 3 is correlated to rule 4.

Listing 3.6: Different Priority Flow Rules with Correlation Conflict.

```
<flow_id=3> priority=52, nw_src=10.5.50.5, nw_dst=10.211.1.0/24,
   nw_proto=6, actions=output:2

<flow_id=4> priority=53, nw_src=10.5.50.0/24, nw_dst=10.211.1.63,
   nw_proto=6, actions=drop
```

Since multiple SDN flow rules can have the same priority, we make the following addition to the correlation conflict to satisfy requirements in an SDN environment: A rule r_i is correlated to rule r_j iff: *a*) priority $p_i = p_j$; *b*) address space $n_i \cap n_j \neq \emptyset$; *c*) protocol $\rho_i = \rho_j$; and *d*) action $a_i \neq a_j$. Thus, the correlation conflict now encompasses all policies that have the different actions, overlapping address spaces and the same priority. Scenarios where address spaces of two flow rules are subsets or supersets, which would have been categorized as generalization and shadowing in a

traditional environment are classified as a correlation if the priorities of the two flows are the same. For example, in Table 3.1, rule 6 is correlated to rule 1.

Listing 3.7: Same Priority Flow Rules with Correlation Conflict.

```
<flow_id=1> priority=51, nw_src=10.5.50.0/24, nw_dst=10.211.1.63,
   nw_proto=6, actions=output:3

<flow_id=6> priority=51, nw_src=10.5.50.0/16, nw_dst=10.211.1.63,
   nw_proto=6, actions=drop
```

5) Overlap: A rule r_i overlaps rule r_j iff: *a*) address space $n_i \not\subseteq n_j \wedge n_i \not\supseteq n_j \wedge n_i \cap n_j \neq \emptyset$; *b*) protocol $\rho_i = \rho_j$; and *c*) action $a_i = a_j$. An overlap rule is similar to a correlation; but with the same action set. Note that the overlap conflict holds irrespective of the priority of the rules in question. This overlap can be seen between rule 6 and rule 7 in Table 3.1, shown below.

Listing 3.8: Flow Rules with Overlap Conflict.

```
<flow_id=6> priority=51, nw_src=10.5.50.0/16, nw_dst=10.211.1.63,
   nw_proto=6, actions=drop

<flow_id=7> priority=55, nw_src=10.5.50.5, nw_dst=10.211.1.0/24,
   nw_proto=6, tcp_dst=0x03e8/0xfff8, actions=drop
```

6) Imbrication: The criteria discussed above does not cover all potential conflicts in SDN environments. Consider the case of flow rules where: *a*) only layer-3 header fields are used as a condition (rule 1-7 in Table 3.1); *b*) only layer-2 header fields are used as a condition for decision (rule 8); and *c*) only layer-4 header fields are used as a condition (rule 9). Even though using our definitions there is no overlap in address space, and hence there should be no conflict, a packet could match more than one of these rules. We classify such policy conflicts as imbricates, and address them by introducing the concept of *reconciliation* (described in Chapter 6) which maps all headers to the same layer. Currently, all cross-layer conflicts are classified as imbrication. They are examined in further detail in Section VI.C.

C. Cross-layer Policy Conflicts

As opposed to a traditional network, flow rules in SDN, could have matches on multiple header fields, thereby resulting in indirect dependencies. For example, consider traffic originating from Host A destined to Host B in Figure 3.5. This flow would clearly match both the flow rules shown in Listing 3.9. Rule with cookie value `0x2b0b` would match on the layer-2 source, and layer-3 destination address; while rule with cookie value `0x2b3a` would match on the layer-2 source, and layer-2 destination address. Since both the rules have the same priority, the action taken by the controller would be inconsistent. As mentioned earlier, since there is no direction in the specification on how to deal with such a scenario, different controllers may deal with these conflicts in a different manner. A flawed approach to tackle this problem would be to expand the address space from layer-3 and layer-4 to include layer-2 addresses, and determine rule conflicts as in a traditional environment. However, since there exists an indirect dependency between the layer-2 and layer-3 addresses, an apples-to-apples comparison impossible. Moreover, flow rules could exist that do not specify all the header fields adding another wrinkle.

Further, such conflicts between rules based on addresses over multiple OSI layers are more complex than the other conflict classifications, since they are transient in nature. For example, the mapping between a layer-2 MAC address and layer-3 IP addresses in Figure 3.5 might result in a conflict between two flow rules at time t_1 in the layer-3 address space. But if the IP-MAC address mapping changes, there may not be an address space overlap between the two rules at time t_2. This makes imbrication conflicts hard to find and even harder to resolve.

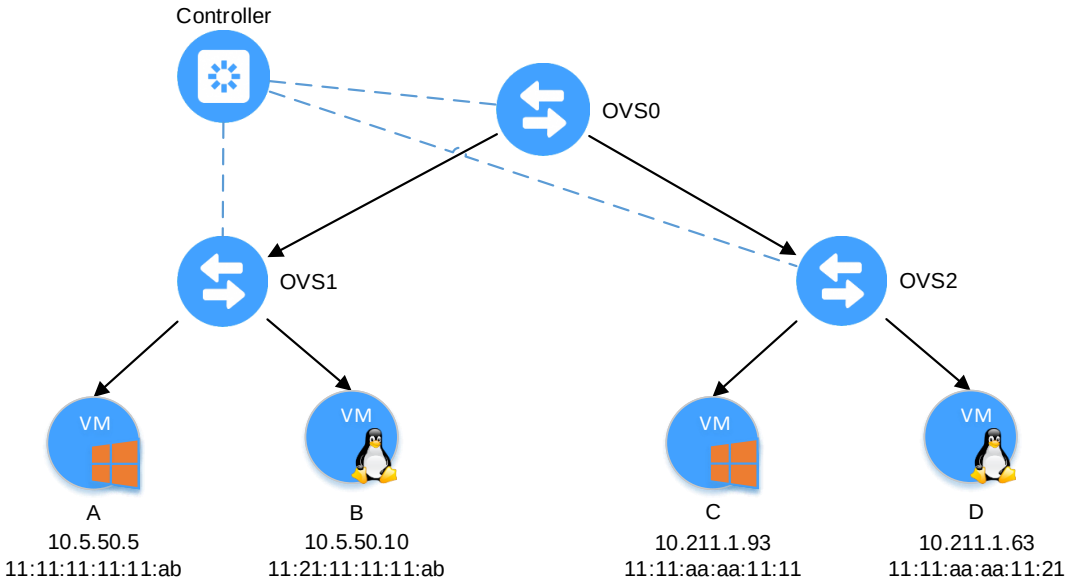

Figure 3.5: Cross-layer Flow Rule Conflict in SDN Environments.

Listing 3.9: Flow Rule Conflict Based on Addresses in Different OSI Layers.

```
cookie=0x2b0b, duration=926.421s, table=0, n_packets=1378, n_bytes
   =271308, idle_age=77, priority=100 dl_type=0x800 dl_src
   =11:11:11:11:11:ab nw_dst=10.211.1.63 actions=NORMAL

cookie=0x2b3a, duration=949.733s, table=0, n_packets=622, n_bytes
   =957, idle_age=144, priority=100, dl_type=0x800 dl_src
   =11:11:11:11:11:ab dl_dst=11:11:aa:aa:11:21 actions=drop
```

D. Traffic Engineering Flow Rules

Traffic engineering (TE) generally includes analysis of network traffic to enhance performance at operational and resource levels [105]. In data center and cloud service provider environments, QoS and resilience schemes are also considered as major TE functions, especially since several applications not only have bandwidth requirements, but also require other QoS guarantees [106]. Given the holistic network view that the SDN controller possesses, TE mechanisms in SDN can be much more

efficient and intelligent, when compared to traditional IP-based mechanisms. Research on SDN based TE has tackled the tradeoffs between latency and load balancing, focusing on: *a*) controller load balancing[2] [6, 8, 75, 107, 108]; *b*) switch load balancing [109, 110, 111, 112, 113]; and *c*) the use of multiple flow tables. Our focus is determining how any implementation of TE functions in SDN environments using flow rules might conflict with security policies. We steer clear of considering controller and switch load balancing issues dealing with TE, and look at how implementing TE policies that direct traffic along certain paths, and implementing QoS for certain flows might interfere with security policy concerning the same flows.

OpenFlow specifications enable packets belonging to certain flows to be directed to a queue of an egress port. However, using queues to implement QoS requires that some configuration be done on the switches themselves, in addition to the controller. The snippet below shows creation of a QoS queue on an OVS that rate limits the maximum rate of this QoS policy to 1 Mbps, while setting the maximum rate to 5 Mbps.

Listing 3.10: QoS Using Rate Limiting.

```
$ ovs-vsctl set port eth0 qos=@newqos -- --id=@newqos create qos
    type=linux-htb other-config:max-rate=1000000 other-config:max-
    rate=5000000
```

Further QoS related additions in OpenFlow enable the setting of rate limiting functions. These utilize meter table entries, which define per-flow meters that measure rate of packets assigned, and enable controlling that rate. Meters are associated directly with flow entries, as opposed to different queues of egress ports, and contain: *a*) an identifier; *b*) the specified way to process the packet; and *c*) counters. Use of the meter table, as shown in Figure 3.6, to establish a game theory based security

[2]Multiple controller scenarios are discussed in Chapter 4

framework was demonstrated by Chowdhary et al. [114], wherein the `rate` sub-field of `band` field in meter table was used to establish rate limiting for non-cooperating actors. Their work shed light on novel ways to use TE to implement security.

Our approach to tackling conflicts while using either of these two QoS scenarios is abecedarian, wherein only the forward/deny aspect of the rule is considered in the detection and resolution of conflicts.

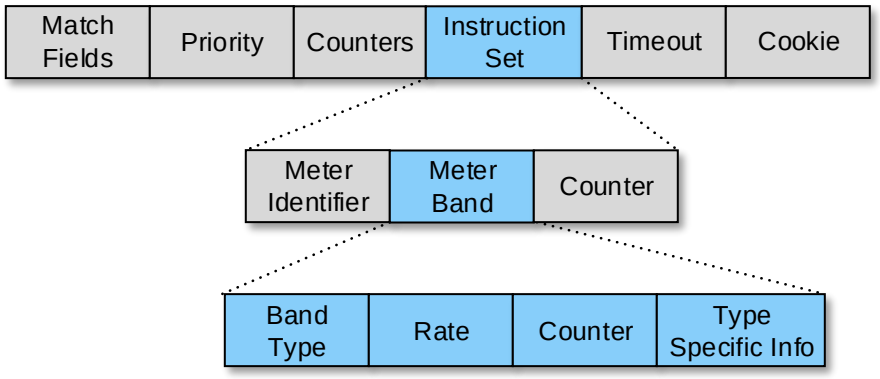

Figure 3.6: Meter Band in OpenFlow Specification.

Chapter 4

DISTRIBUTED SDN CONTROLLER CONSIDERATIONS

SDN was designed with a centralized control plane in mind. This empowers the controller with a complete network-wide view and allows for the development of control applications and for easier policy enforcement. Centralizing the control plane in SDN is fraught with scalability challenges associated with the SDN controller being a bottleneck [4]. Benchmarking tests on an SDN have shown rapid increase in the performance of a single controller, from about $30,000$ responses per second using NOX [115] in 2009 to over $1,350,000$ responses per second for Beacon [108] in 2013. But with data center architectures dealing with 100 GB network traffic (equal to about 130 Million Packets Per Second (MPPS) [116]), a single controller would still not scale well enough to be deployed in a cloud environment [5]. Further, large production environments still demand performance and availability [77]. Distributing the controller responsibilities to multiple devices/applications, while maintaining logical centralization is an obvious solution. Figure 4.1 shows a representation of major different distributed controller categories; namely clustered and hierarchical.

While the OpenFlow protocol supports multiple controller environments, the controllers themselves need to be able to: *a*) allow a switch to establish communication with them; and *b*) have mechanisms in place to process handover, fail overs, etc. OpenFlow shields itself from the complexities of a multiple controller environment, and just requires the controller to have one of three roles - `OFPCR_ROLE_MASTER`, `OFPCR_ROLE_EQUAL` or `OFPCR_ROLE_SLAVE`[1].

[1]The *Master* and *Slave* roles are self-explanatory. *Equal* role and *Master* role are exactly the same, with the difference that only one controller can be *Master* at a time, while multiple controllers can share the *Equal* role.

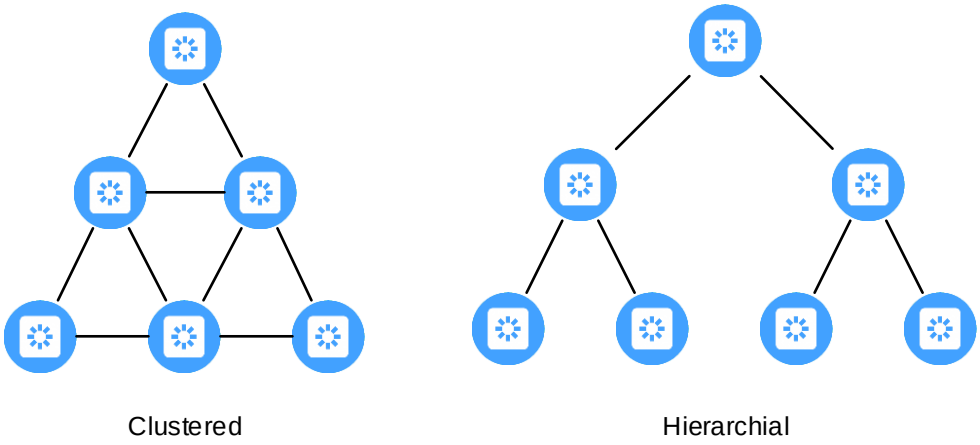

Figure 4.1: Distributed Controller Classes.

Moving to a distributed controller environment essentially splits the roles of the SDN controller between multiple devices that communicate with the switches, and a data store that retains complete environment knowledge.

I. Challenges in Multiple-Controller Domain

Several studies have attempted to study distributed SDN controllers (see Chapter 2, Section IV). However, despite their attempt at distributing the control plane, they require a *consistent* network wide view in all the controllers. Maintaining synchronization and concurrency in a highly dynamic cloud environment is problematic. Since the SDN switches look to the controllers for answers while handling unfamiliar flows, knowing which controller to ask is important. Moreover, the controllers themselves need to have methodologies to decide who controls which switch, and who reacts to which events. And most of all, consistency in the security policies present on the controller is paramount - the absence of which might result in attackers using application hopping across multiple partitions of the SDN environment without permissions.

Since one of the primary motivations behind SDN was a centralized control plane that has complete knowledge of the environment, maintaining a complete picture after dividing the network into multiple sub-networks requires information aggregation. This could be challenging, especially when the environments are dynamic.

While designing a distributed controller architecture, the implications of controller placement needs to be considered carefully. For example, security policies in a mesh controller architecture would have to ensure minimal address space overlap; while in a hierarchical architecture, it may be acceptable for the lower level (leaf) controllers to share address spaces. In environments where latency is a concern, the distance between controllers and the switches needs to be minimized.

Finally, studies also suggest that distributed control planes are not adaptable to heterogeneous and constrained network deployments [76]. This removes a certain amount of desired design flexibility from the SDN setup.

II. Controller Decentralization Model

Choosing a decentralized control architecture is not trivial. There are several controller placement solutions, and factors such as the number of controllers, their location, and topology impact network performance [117]. Three major issues need to be elucidated [118] while determining the decentralization architecture:

- Efficient east and westbound APIs need to be developed for communication between SDN controllers.
- The latency increase introduced due to network information exchange between the controllers need to be kept to a minimum.
- The size and operation of the controller back-end database needs to be evaluated.

Since the key piece of information required for accurate flow rule conflict detection (and resolution, as will be described in Chapter 5) is the priority value of the flow rule p, the key challenge in extending flow rule conflict resolution from a single controller to a distributed SDN-based cloud environment lies in associating *global priority* values to flow rules. Definition 4.1 defines global priority.

Definition 4.1. A global priority p' of a rule r_i is value in the range $[1, 65535]$, as determined by weighing the priority value p by the rule origination point's position in the global distribution scheme. Alternately, p' could be obtained using a static mapping scheme from p.

To illustrate Definition 4.1, consider flow rule as shown with a priority value of 51. If this flow rule originated in a controller or application with a weight of 2, its global priority would be 102.

Listing 4.1: Global Priority Determination.

```
priority=51, nw_src=10.5.50.5, nw_dst=10.211.1.1, actions=output:3

global_priority=102, priority=51, nw_src=10.5.50.5, nw_dst
   =10.211.1.1, actions=output:3
```

The strategies to associate these global priority numbers to flow rules in different decentralization scenarios differ drastically. We classify five different multiple controller scenarios, and the global priority assignment logic followed by our framework for each of them.

A. Clustered Controllers

The clustered SDN controller is the simplest of the multiple controller environments. It is ideal for smaller networks, where one controller can process all events and run all applications. Clustering adds a layer of defense against the controller being a single point of failure by having one or more controllers in an active/standby scenario. Since

all the controllers run the same applications and communicate with all the data plane devices, the global priority assigned to the rules would be equal to the priority of the rule.

Flow rule conflict classification and resolution in clustered controllers is handled exactly like in a single controller environment, owing to a lack of partitioning of data plane devices or applications. Hence, discussion of clustered controllers as a decentralization strategy is limited in the rest of this dissertation.

B. Host-based Partitioning

Host-based partitioning is most like a traditional layered network architecture, where an SDN controller handles the functionalities of an access layer switch, combined with the intelligence of a router and access control server. The SDN-based cloud environment is separated into domains, where each domain is controlled by a single controller. As shown in Figure 4.2, the tenant infrastructure in multi-tenant data center environment could be considered a domain that is handled by one controller. All the controllers present in the environment would maintain global knowledge of the environment by communicating with each other using east-west communication APIs.

Running on the assumption that the controller knows best about the main it is responsible for, flow rules which contain match conditions with addresses *local* to the controller are preferred. For example, the rule with cookie value `0xa` added onto Controller 1 permits DNS traffic into host 10.211.1.5, which we assume, is an address assigned to Tenant A. If the rule with cookie value `0xb` is added on Controller 2, the two conflicting flow rules will be known to all the controllers owing to the controllers sharing their information.

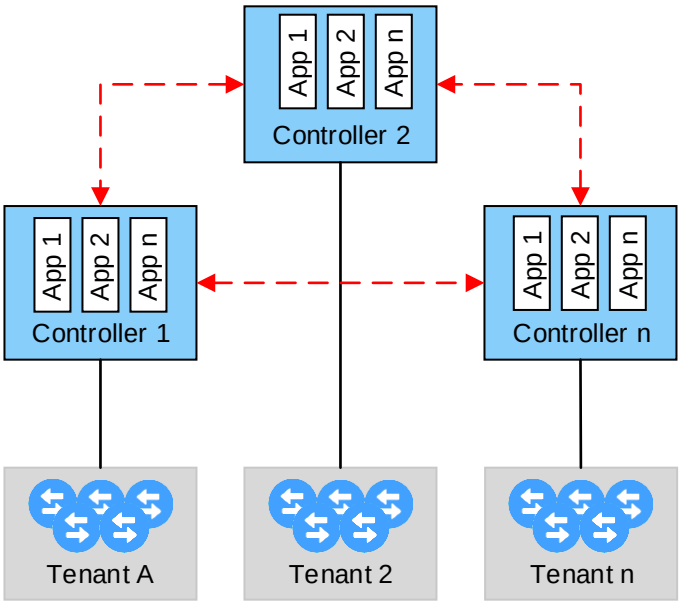

Figure 4.2: Host-based Partitioning.

Listing 4.2: Conflicting Flow Rules in Host-based Partitioning.

```
cookie=0xa, priority=100, nw_dst=10.211.1.5, nw_proto=17, udp_dst
   =53, actions=output:1
cookie=0xb, priority=100, nw_dst=10.0.0.0/8, nw_proto=17, udp_dst
   =53, actions=drop
```

To help select the rule most applicable to the tenant, we assign weights to the flow rules such that the ones originating from the controller assigned to the specific domain (Controller 1, in our example) is considerably higher. The weight itself is dependent on the environment, and can be assigned by an administrator. Assuming a weight of 10 for the local controller, we now have global priorities as shown in the modified flow rules below. The global priority value can then be used for conflict resolution.

Listing 4.3: Assigning Global Priority in Host-based Partitioning.

```
global_priority=1000, cookie=0xa, priority=100, nw_dst=10.211.1.5,
   nw_proto=17, udp_dst=53, actions=output:1
global_priority=100, cookie=0xb, priority=100, nw_dst=10.0.0.0/8,
   nw_proto=17, udp_dst=53, actions=drop
```

Host-based partitioning is popular in several cloud deployments, owing to its simplicity. In DragonFlow [119], for example, an instance of the controller runs on every compute node in the OpenStack [38] cloud. Partitioning in DragonFlow is based on the node it runs on, and not purely tenant-based. Thus, an instance of DragonFlow would manage the OVS and flow rules that are associated with hosts running on the same compute node. The different DragonFlow instances in the cloud share information by communicating with a shared back-end database. Partitioning schemes such as those employed by DragonFlow are intuitive and most like decentralization strategies used in traditional environments.

C. Hierarchical Controllers

Hierarchical controller distribution is a variant of host-based partitioning, where some controllers handle a subset of data plane devices, while others only communicate with control plane devices. The controllers that communicate with the data plane devices can be thought of as leaf-level controllers, while higher-level controllers communicate solely with other controllers. Figure 4.3 shows a hierarchical distribution of controllers. Further, the partitioning may not be strictly host-based, as administrators could decide to run certain applications on leaf-level controllers, and other applications on higher-level controllers. For example, a DHCP application could reside on the leaf controller while a NAT application could reside on the root controller.

Since higher-level controllers do not communicate with data plane devices, except in cases when leaf-level controllers fail, control channel communication is streamlined. Leaf-level nodes can obtain global information by communicating with the higher-level controller, eliminating the need to talk with every other leaf controller. Since the root controller would have holistic knowledge of the environment, in case of conflicts flow rules originating from the root controller are preferred.

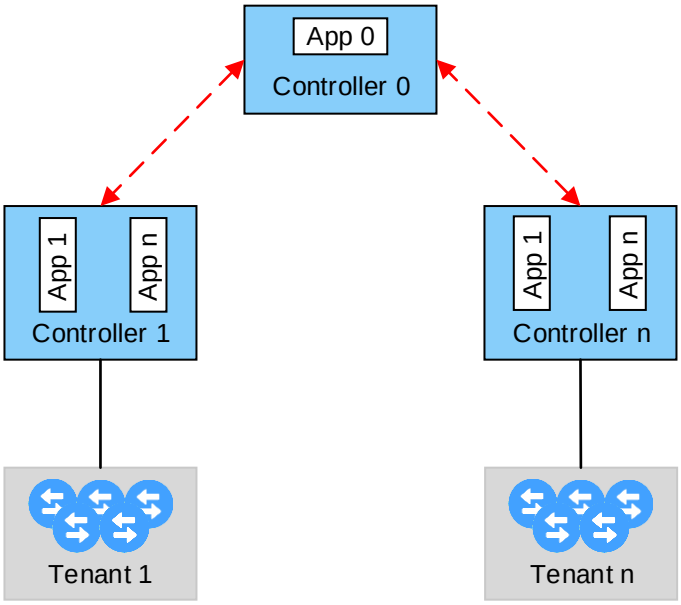

Figure 4.3: Hierarchical Controller Distribution.

Revisiting the example from host-based partitioning scheme, consider that the rule with cookie value `0xa` added onto Controller 1, while the rule with cookie value `0xb` is added on Controller 0. While Controller 1 might still permit DNS traffic into the host at IP 10.211.1.5, the root level controller might have an IDS application running on it that detected a DDoS attack, and dropped all DNS traffic to the devices on 10.0.0.0/8 subnet. Once again, we use a weight of 10, but this time for the rule on Controller 0, thereby ensuring the conflict resolution algorithm drops the attack traffic.

Listing 4.4: Assigning Global Priority in Hierarchical Partitioning.

```
global_priority=100, cookie=0xa, priority=100, nw_dst=10.211.1.5,
   nw_proto=17, udp_dst=53, actions=output:1
global_priority=1000, cookie=0xb, priority=100, nw_dst=10.0.0.0/8,
   nw_proto=17, udp_dst=53, actions=drop
```

Hierarchical controllers may also be used if there is a specific structure to the network, such as a two-tier structure. In such situations, hierarchy at the controller level can help to manage flows for different tiers in the network. However, coordination needs to be addressed to ensure efficiency in flow management.

D. Application-based Partitioning

Application-based partitioning, as shown in Figure 4.4 implements decentralization by having different applications run on different instances of the controller. As with host-based partitioning, the flow rules generated by the different applications would be known to the other controllers using the east-west communication APIs. Each data plane device in this scenario would communicate with every controller in the environment.

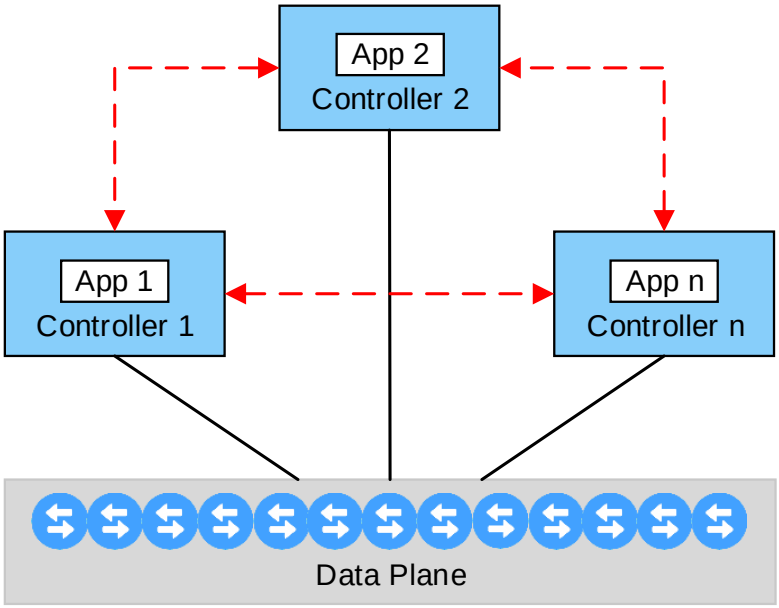

Figure 4.4: Application-based Controller Decentralization.

Associating global priority values in application-based decentralization is straightforward. It could be done by assigning a weight to each application [11] using which the global priorities of flow rules generated by all applications can be determined. For example, consider Controller 1 with security applications running on it, and Controller 2 with QoS and traffic shaping applications running on it. If security applications are prioritized with a higher weight than traffic shaping applications, flow rules with the

same priority generated by applications on Controller 1 and Controller 2 will end up with the rule generated by Controller 1 having a higher global priority.

An alternate strategy to assign global priority values would be to allocate ranges for flow rules created by applications. For example, it could be decided that any NAT rule generated by the NAT application on the controller must be within a priority of $40,000$ and $42,000$. Thus a global priority for a NAT rule would be generated by mapping the priority originally in the range $[1, 65535]$ to a global priority in the range $[40000, 42000]$.

E. Heterogeneous Partitioning

In a heterogeneous decentralized environment, appealing aspects of each of the above decentralization scenarios are combined to obtain the optimal situation for meeting the requirements. Careful consideration needs to be taken to identify the priorities of applications and controllers before deployment, to have a conflict resolution strategy.

Chapter 5

FLOW RULE CONFLICT RESOLUTION

In this Chapter, the considerations for flow rule conflict resolution are discussed. First, the flow rule conflict types discussed in Chapter 3 are categorized based on their difficulty of resolution. A conflict resolution model that addresses automatic resolution of the flow rule conflicts are presented next.

I. Conflict Severity Classification

Based on their potential for causing damage, as well as their difficulty to resolve, the flow rule conflicts formalized above can be classified into the following tiers:

A. Tier-1 Conflicts

Imbrication conflicts stem from flow rules using addresses in multiple layers. Since mapping between different layers is of transient nature in dynamic SDN environments, these conflicts are transient as well. Any resolution technique used to resolve these conflicts is, at best, made taking the current system state into account. Such a resolution might induce the system to be in a highly compromisable state at a future time, which could be exploited by attackers. To illustrate, consider once again, the topology in Figure 3.5. If a layer-2 policy and a layer-3 policy that constrains traffic between hosts A and D are present, one of the conflict resolution strategies might select the layer-2 flow rule. However, if the mapping between the layer-2 and layer-3 addresses change, the conflict resolution decision might be rendered invalid.

B. Tier-2 Conflicts

Conflicts classified as *generalization* and *correlation* stem from overlapping address spaces and incompatible actions. These conflicts stem from attempts at combining and oversimplifying flow rules. By making the address spaces used in the flow rules as fine-grained as possible, tier-2 conflicts can be eliminated.

C. Tier-3 Conflicts

Conflicts categorized as *redundancy* and *overlap* result from rules with overlapping address spaces but the same resulting action. The *shadowing* conflict stems from rules which are never invoked. In case of redundancy and overlap, the action remains the same, so choosing any either flow rule would result in the same action on the packet. Shadowed rules are never invoked owing to their lower priority. Thus, we content that while it is not ideal that these conflicts exist in the system, their presence in the system is not a security threat but an optimization issue.

II. Conflict Resolution Model

The different flow rule conflicts can be broadly categorized into **Intelligible** and **Interpretative** conflicts. The resolution strategies for each of these two categories are markedly different, and are detailed in the remainder of this Section. Tier-1 and Tier-2 conflicts are interpretative in nature, while Tier-3 conflicts are intelligible in nature.

A. Intelligible Conflicts

Flow rules that conflict with each other in the Redundancy and Overlap classifications all have the same action they can be resolved without the loss of any information. Rules that have shadowing conflicts can simply be removed, without affecting any

packet. In other words, the resolution algorithm can guarantee that any packet that is permitted by the controller prior to resolving the conflict will continue to be permitted after conflict resolution. And similarly, any packet that is being blocked prior to conflict resolution will continue to be blocked after the conflict resolution is put in place. Intelligible conflicts are resolved easily by eliminating the rules that are not applied, or by combining and optimizing the address spaces in the rules to avoid the conflict [120].

It could be argued that creative design of rules by administrators result in flow rules that deliberately conflict to optimize the number of rules in the flow table, especially when it comes to traffic shaping policies. However, such optimization strategies stem out of legacy network management techniques, and do not hold true in dynamic, large-scale cloud environments where the flow table enforcing the policies in the environment could have millions of rules.

B. Interpretative Conflicts

Conflicts that fall into Generalization, Correlation and Imbrication classification cannot be intuitively resolved without any loss of information, and are interpretative in nature. As opposed to intelligible conflicts, it is not guaranteed that any packet permitted by the controller prior to resolving the conflict will be permitted after conflict resolution. Since interpretative conflict resolution is lossy in nature, the resolution strategies are *not* a one size fits all and need to be adapted per the cloud environment in question. Removing these conflicts is a complex problem [121].

A few different resolution strategies that could be applied to resolving these conflicts are discussed below. The global priority of the rule is assigned depending on the controller decentralization strategy discussed in Chapter 4. Resolution strategies for Tier-1 conflicts are shaky at best. Since these conflicts are transient in nature, an

additional decision needs to be made as to the time duration for which the conflict resolution strategy is valid. We look at four different strategies.

1) Least Privilege: In case of any conflict, flow rules that have a deny action are prioritized over a QoS or a forward action. If conflicts exist between a higher and lower bandwidth QoS policy, the *lower* QoS policy is enforced. The least privilege strategy is traditionally the most popular strategy in conflict resolution [122].

2) Module Security Precedence: Since flow rules in an SDN-based cloud environment can be generated by any number of modules that run on the controller, an effective strategy that can be put in place is to have a security precedence for the origin of the flow rule [11]. Thus, a flow rule originating from a security module is prioritized over flow rule from an application or optimization module. The weighted global priorities are calculated as discussed in the application-based partitioning scheme discussed in Chapter 4, Section II. Table 5.1 shows sample precedence and associated global priority weight values for a few generic applications that might run in an SDN-based cloud.

Application	Precedence	Global Priority Weight
Virtual Private Network	1	3
Deep Packet Inspection	2	2.5
Network Address Translation	3	2
Quality of Service	4	1.5
Domain Name Service	5	1

Table 5.1: Security Precedence Priority Multiplier Example.

3) Environment Calibrated: This strategy incorporates learning strategies in the environment to make an educated decision on which conflicting flow rule really needs

to be prioritized. Over time, if a picture can be formed about the type of data that a certain tenant in a multi-tenant data center usually creates/retrieves, or of the applications and vulnerabilities that exist in the tenant environment, or of the reliability of the software modules inserting the flow rule; the conflict resolution module may be able to prioritize certain flow rules over others. However, these techniques falter while dealing with a dynamic cloud.

A more deliberate approach might involve quantitative security analysis of the environment with each of the conflicting rules, and picking the safest option. Metrics originally proposed by Joh and Malaiya [123] and validated by Lippmann et al. [124] provide a quantitative measurement of the probability of the Cyber Key Terrain (CKT) [125] being compromised. Interpretative conflict resolution could be as simple as determining which of the conflicting policies would reduce the compromise probability.

4) Administrator Assistance: Administrators that are willing to give up automatic conflict resolution have the option to resolve conflicts manually, so they can judge each conflict independently. Visual assistance tools incorporated as part of the Brew framework assist the administrator make a decision, and are detailed in [126].

Chapter 6

BREW: A SECURITY POLICY MANAGEMENT FRAMEWORK IN DISTRIBUTED SDN ENVIRONMENTS

In this chapter, a system overview, architecture detail for the Brew security policy management framework is provided. In addition, implementation details for the framework on an OpenDaylight (ODL) SDN controller is furnished.

I. System Overview & Models

This section describes the design requirements, assumptions, operating environment as well the security model for Brew.

A. Design Requirements & Assumptions

Brew satisfies the following design requirements:

- The flow rule set that is generated as output should be without any conflicts. This includes the need to detect conflicts in a security implementation, both intra-flow table and inter-flow table, and resolve conflicts that might result due to changing security requirements.
- Cross-path policy discrepancy must be addressed: Policy consistency across all possible paths between the source (external or internal) and the destination is required, thereby ensuring that no matter what route is taken by a packet, the same security rules apply.
- A highly desirable feature is automation, since the human element is more likely to introduce error. Ideally, human intervention should be required only in case of zero-day type situations where an intelligent decision needs to be made.

- Implementation should include a mechanism to safely distribute the security policies to all implementation points.

- The implementation should be able to adapt to a dynamic network topology and real-time updates, without compromising the overall security policy of the enterprise or operation. This requirement is especially important as one of the motivating goals behind SDN was being able to modify a network topology as demanded by the conditions.

Two major assumptions are made during implementation and testing of the framework, both pertaining to administrator trust:

- The administrator workstation is secure and uncompromised. No rogue agent is present on the workstation that can poison the flow tables. This implicit trust placed on the administrator masks potential issues that might arise in when the conflict resolution strategy looks to the administrator for resolution.

- The administrator has a global view of the environment and acts in good faith to resolve conflicts between conflicting tenants.

B. Operating Environment

The environment is any data network installation based on SDN principles, such as an IaaS cloud. In an IaaS cloud, VMs are managed by tenants. Changing business needs, changing security concerns or simply implementing new applications may result in new policies that must be implemented on packets traversing the network. When new policies are applied, it is necessary to check and resolve conflicts with existing rules as well as enterprise security policies and business SLAs.

C. Security Model

Brew seeks to alleviate concerns regarding security policy conflicts in SDN environments. However, security challenges that are not addressed in Brew include:

- The southbound interface between the controller and switches are vulnerable to threats that could degrade the availability, performance and integrity of the network.
- Attacks that consume controller resources would impact the conflict detection and resolution algorithms, which run on the controller.
- Verification of flow rule generating applications is not done. It is assumed that verified applications are generating candidate flow rules.

II. System Architecture

A. System Modules

The Brew framework uses an intuitive model to help resolve conflicts in flow rules in a distributed SDN-based cloud environment. Brew runs as an application on the controller, consisting of two inter-related modules, the OFAnalyzer and OFProcessor, that together achieve conflict free flow tables. Both modules operate at the control plane level, i.e., their operations are uninhibited by either the physical topology or the logical topology as seen by the different tenants. The OFAnalyzer font-end has two sub-processes - Flow extraction engine, and a visualization engine. The OFProcessor back-end has three sub-processes - flow prepping, conflict detection and conflict resolution. Figure 6.1 shows the flow of control and logic between these sub-processes.

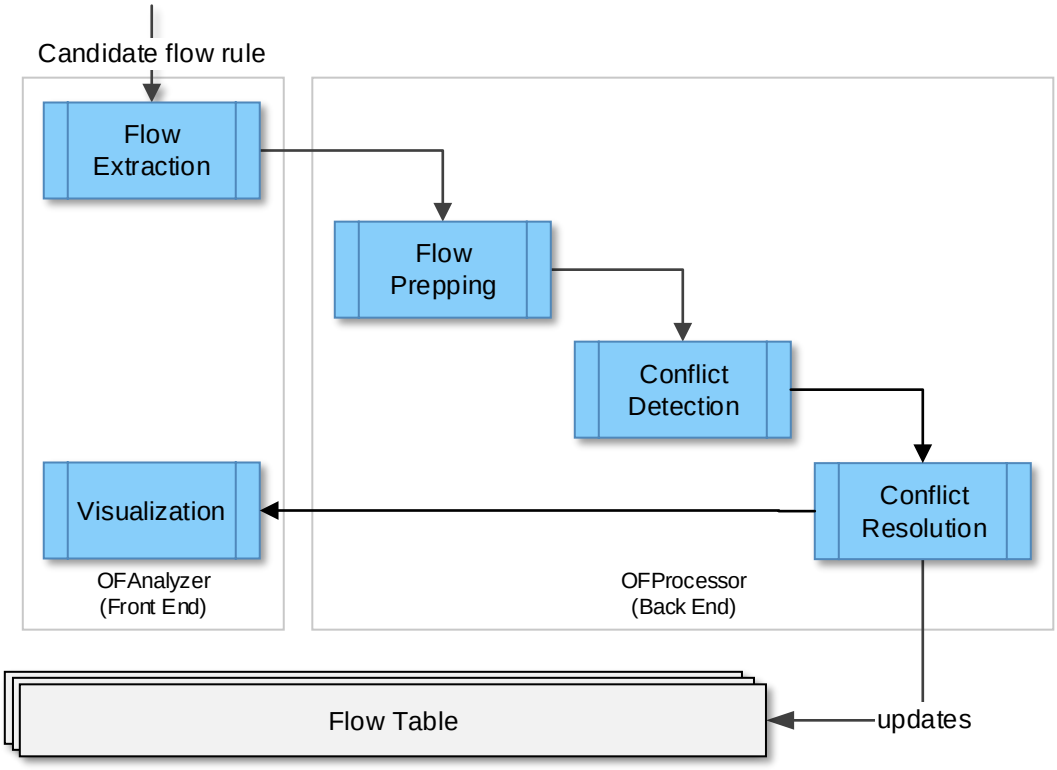

Figure 6.1: Flow of Control and Logic Between Brew Sub-Processes.

The OFAnalyzer module serves as the font-end of Brew, with a listening engine for new/modified flow rules being introduced into the system as well as flow rules that are being removed. A visualization engine serves the processed conflict data back to the administrator. In the automatic resolution mode, the visualization engine serves the conflict resolution details to the administrator.

The OFProcessor is responsible for the back-end processing in Brew. The processing is broadly compartmentalized into prepping, conflict detection and conflict resolution. The modules that accomplish these tasks are detailed in the remainder of this section.

B. OFAnalyzer Module

The OFAnalyzer module acts as the interface between the SDN controller and the OFProcessor back-end. It performs two important tasks: *a*) flow extraction; and *b*) visualization.

1) Flow Extraction Engine: The flow extraction engine intercepts any new or updated flow rule that is being injected into the controller from different modules. These rules, called candidate flow rules, are defined in Definition 6.1. In a distributed controller scenario, candidate flow rules into every controller are obtained to have complete knowledge of all possible flow rules that are present in the environment. These rules are then aggregated for purposes of processing. In addition, the flow extraction engine does a periodic pull of all the flow rules present on the data switches to ensure it has complete knowledge of all the flow rules present in the environment.

Definition 6.1. A *candidate flow rule* is a flow rule that an application or an administrator wants to insert into the flow table. It has not been completely processed and vetted, and hence is not eligible to be sent to any of the switches. A flow removal attempt would also be considered as a candidate flow rule before it has been processed.

Next, the global priority of candidate flow rules are computed by weighing the priority of the rule based on the decentralization strategy that has been employed as discussed in Chapter 4. Thus, the priority assigned by the flow extraction engine may differ from the priority of the flow rule present in the flow table.

The default OpenFlow rule specifications do not provide us all the information needed to detect and resolve flow rule conflicts. Thus, a data structure with four additional fields is added to each flow rule. These 24 bits of information shown in Figure 6.2, are: *a*) One bit identifying if the rule in question has been tagged as a

reconciled rule (required for imbrication detection); *b*) seven bits identifying the SDN controller to which the rule is going to be inserted; and *c*) sixteen bits for a global priority of the flow rule (to be used for flow rule conflict resolution).

Armed with these additional bits of information, detection and resolution of conflicts between the candidate flow rule and flow rules currently present in the flow table is now possible. Control is handed off to the flow prepping engine.

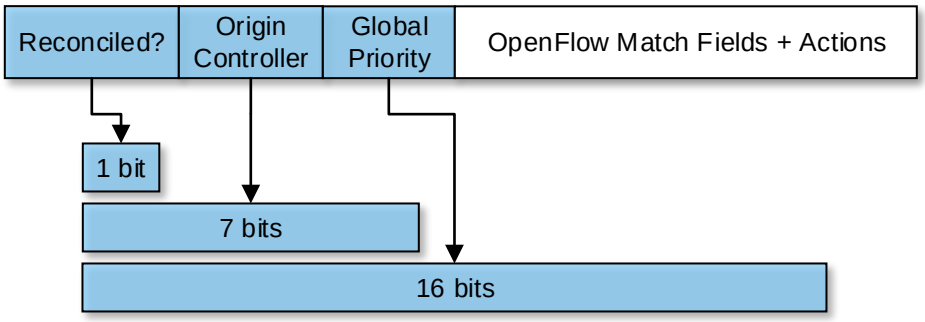

Figure 6.2: Data Structure Format.

2) Visualization Engine: The visualization engine in Brew is a module under the DLUX UI [127] that performs a REST Request to obtain the flow rules present in the environment, along with the conflict information from the OFProcessor in JSON format. JavaScript conversion routines aggregate and transform this information using various visualization techniques.

In an automatic conflict resolution mode, no conflicts would be available to be presented in the GUI. In such scenarios, the Visualization engine could be repurposed to store and present resolved conflicts in a time bounded manner.

C. OFProcessor Module

The OFProcessor module handles the back-end logic of Brew. Its functionalities are compartmentalized as flow rule prepping, conflict detection and conflict resolution.

The candidate flow rules are atomized and the `reconciled` bit in the data structure shown in Figure 6.2 is determined. Conflict detection and conflict resolution strategies follow the discussion in Chapter 3 and Chapter 5.

1) Flow Prepping Engine: Since OpenFlow permits chained flow rules by having an action for a match redirect to a different flow table, to correctly identify conflicts between flow rules, flow rules are atomized by processing the chains and ensuring that only the atomic actions of forward and drop remain. The atomization process itself follows along the lines of `ipchain` processing in Unix with modifications based on the formal model described in Chapter 3. Since QoS and packet counters can be processed along with the forward and drop actions, flow rules with QoS and traffic engineering actions are mapped to the forward action. There are two important considerations made here:

- While the actions for a flow rule can include any drop, forward, flood, set QoS parameters, change several header fields, or redirect to a different flow table, we process the actions and generically classify them into two categories; forward and drop. For example, implementing an IP mapping rule in OpenFlow would change the IP address headers and forward onto a different flow table that forwards the traffic. Such a chain is processed to include the address translation information and the final atomic action of the flow rule is set to `forward`.

- For rules which have multiple actions, the rules are duplicated to generate rules with identical priority and match conditions with a single action.

Flow rules which have only layer-2 addresses as its match conditions are next mapped to their layer-3 addresses using a process called reconciliation. A transient 1-to-1 mapping between the layer-2 and layer-3 addresses is obtained by doing an

ARP table lookup. The layer-3 addresses are then populated in the corresponding address fields in the flow rule. In cases where a mapping is not found, the layer-3 address fields is left unpopulated. Rules that have only layer-4 match conditions are also processed in a similar manner.

2) Conflict Detection Engine: As the name suggests, this process is responsible for detecting any conflicts that might exist between the prepped candidate flow rule set, and the rules that currently populate the flow rule table. In addition to identifying the conflict, it also classifies conflicts based on the categories described in Chapter 3.

Determining the existence of address space overlap between flow rules is the first step in deciding if a conflict exists between two flow rules. The address space overlap is detected using a Patricia trie lookup [128] based algorithm. The Patricia trie itself has been employed for routing table lookups within the BSD kernel since the 4.3 Reno release, and has been used previously with great success [71, 129] owing to it being an efficient search structure for finding IP string matches [130] with a good balance between running time (lookup and update) and memory space requirement. An octet wise Patricia trie lookup is conducted to look for IP address range overlap between the new rules being inserted and existing rules in the flow table in a fast and efficient manner. Once an address space overlap is determined, evaluating if a conflict exists between the flow rules can be accomplished in constant time using simple comparison operations.

3) Conflict Resolution Engine: Once the flow rule conflicts have been detected, the conflict resolution module is invoked. Intelligible conflicts are resolved automatically. In case of interpretative conflicts which cannot be resolved without loss of information, the administrators can decide the resolution strategy (discussed in Chapter 5), which then employs the global priority value to resolve conflicts.

III. Implementation

Brew was implemented on an OpenDaylight (ODL) [131] SDN controller. Section III.A describes the ODL controller with information pertinent to Brew. By modularizing the functionality of Brew into the front-end OFAnalyzer interface (customized to ODL), and a back-end OFProcessor, as shown in Figure 6.3, flexibility to have the same OFProcessor back-end work with different SDN controller specific OFAnalyzer interfaces in the future is maintained.

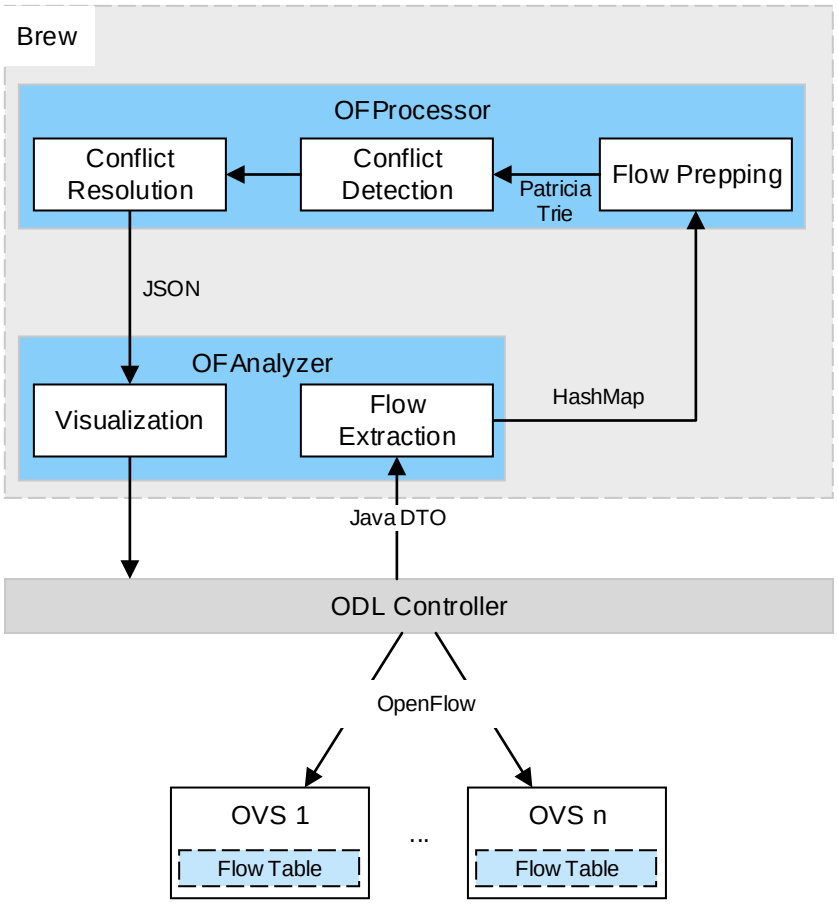

Figure 6.3: System Overview Representing Different Brew Modules.

Prior to examining the implementation of each sub-process in detail, it is important to have a basic understanding of the ODL controller, and associated tools.

A. OpenDaylight (ODL)

ODL is an open-source project under the Linux Foundation [132]. Applications running on the ODL controller use a Service Abstraction Layer (SAL) to communicate with different types of devices using a variety of communication protocols, and provide RESTful APIs for use by external applications. ODL was chosen as the controller in this implementation because of its large open-source development community, as well as indications during decision making that ODL would be adopted as an industry standard. This work extends the stable Lithium version of the controller. Figure 6.4 shows the ODL architecture including the different modules.

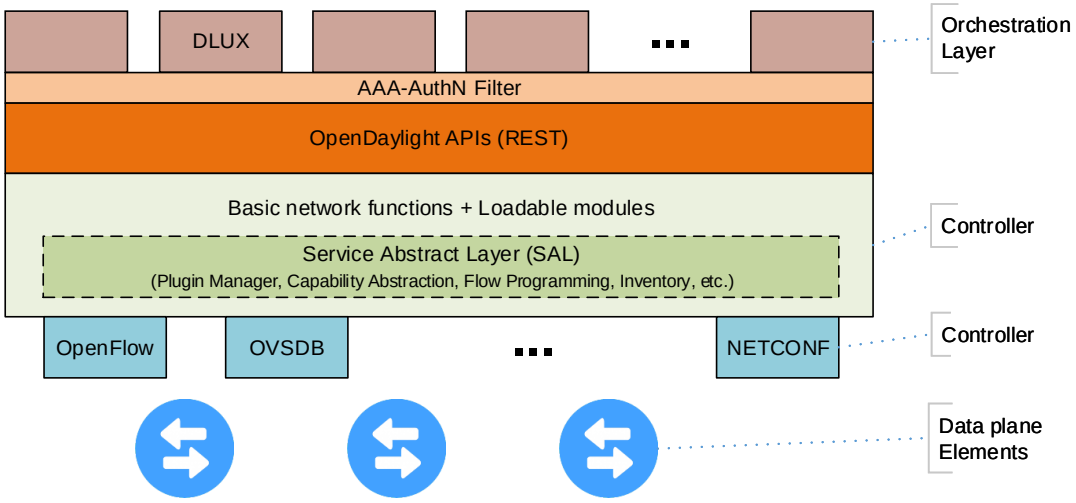

Figure 6.4: ODL Architecture.

The ODL project repository, available at [133] follows a microservices architecture to control applications, protocols, plugins and interfaces between providers and customers. It uses YANG data structures along with shared data stores and messaging

infrastructure to implement a Model Driven SAL (MD-SAL) approach to solving more complex problems. This model helps keep the controller as lightweight as possible, providing users with the ability to install protocols and services as needed. As of this dissertation, the ODL ecosystem has implementations for Switching, Routing, Authentication, Authorization and Accounting (AAA), a DLUX based Graphical User Interface (GUI) and support for protocols such as OpenFlow, NETCONF, BGP/P-CEP, SNMP, CAPWAP. Additionally, it interfaces with OpenStack [38] and OVS through the OVSDB Integration Project [134]. This modularization and separation of functionality has been implemented per the Open Services Gateway Initiative (OSGi) specification, and as such provides for service object initiation, dynamic module handling and graceful exit.

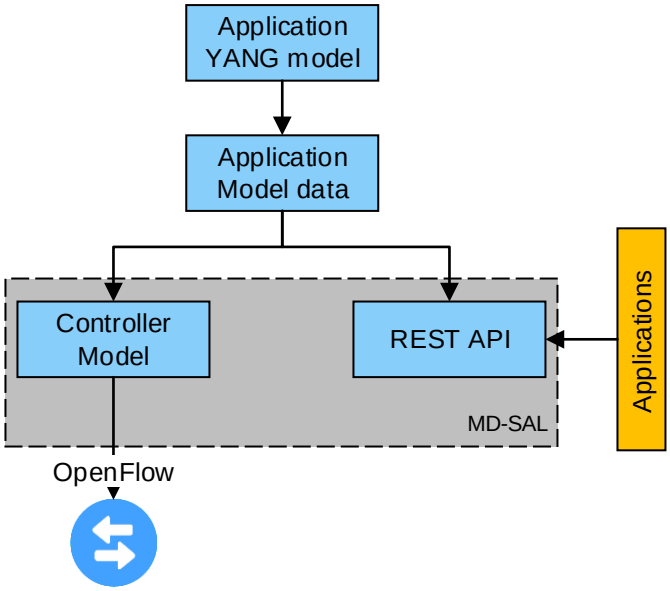

Figure 6.5: MD-SAL Application Development.

ODL uses Apache Karaf [135] as its OSGi container. Applications[1] in Karaf are independent of each other, and can be started, stopped or restarted without affect-

[1]Interchangeably called bundles or features. Karaf command line uses the keyword *feature*

ing other applications. Brew uses the `l2switch`, `openflowplugin`, `openflowjava`, `yangtools`, `netconf` and `dlux` features. RESTCONF [136] provides a RESTful API to perform Create, Retrieve, Update and Delete (CRUD) operations using NETCONF, which itself is a means to configure network elements in a vendor agnostic manner using the YANG modeling language. Figure 6.5 shows the relationship between the different protocols and modeling languages in a MD-SAL development paradigm [126]. A new application development requires defining the application's model using YANG.

ODL maintains two different data stores, as shown in Figure 6.6. Classified broadly on the type of data maintained in them, they are: *a*) configuration data store; and *b*) operational data store. Since the data is stored in a tree format, the configuration and operational data stores are interchangeably called the configuration and operational trees.

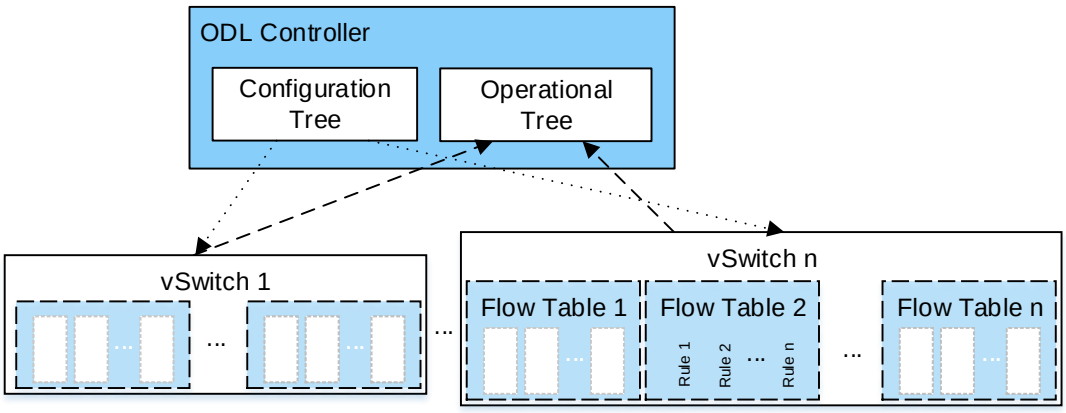

Figure 6.6: ODL Data Stores.

The configuration data store on each ODL controller contains data that describes the changes to be made to the flow rules on the switches. It represents the intended state of the system, and is populated by administrators, or applications on the controller. The configuration data store contains information about every device

present in the environment, flow tables associated with the devices, and the flow rules in every flow table. To give administrators and other applications the ability to populate this data store, it has read/write permission. The operational data store matches the configuration data store in structure, but contains information that the controller discovers about the network through periodic queries. It represents the state of the system as understood by the data plane components in the environment. As opposed to the configuration data store, the operational data store has read-only permissions. The use of dual data stores is primarily to maintain global knowledge of the environment while supporting a multiple controller scenario. For example, if Controller 1 has a new flow rule that is used by an OVS to direct traffic, Controller 2 would learn of this flow rule when it populates its operational data store with all the flow rules present in the environment. This would happen irrespective of the communication between the two controllers.

B. Flow Extraction Engine

ODL provides RESTful APIs to add to, remove from, and update flows in the configuration data store, and to view the operational data store. These APIs can be accessed by the administrator through a Web UI, CLI, or other RESTful clients like POSTMAN [137], and by other applications running on the controller through the same APIs. The flow rules are then sent to the switches using the OpenFlow protocol.

Brew modifies the native behavior of ODL. It listens for potential changes being piped into the system using the RESTful APIs, and processes them. Natively, applications running on the controller attempt to add flows using the RESTful APIs for the controller. When the flow is successfully added to the configuration data store on the controller, a `dataChanged` notification is issued to the Flow Programmer service.

The flow extraction engine in Brew defines an object pointing to the `Flow.class` under `opendaylight-flow-types.yang` model in ODL. This object is used to register as a listener in the `DataBrokerService` Document Object Model (DOM) tree, which enables the object to learn about changes to the configuration data store when the `onDataChanged` function from the Flow Programmer Service returns true. The required data is received using a Java Data Transfer Object (DTO). The flows extracted are structured as shown in Listing 6.1 using Algorithm 1, and stored in a local HashMap.

Listing 6.1: Extracted Flow Rule Structure.

```
grouping flow {
    container match {
        uses match:match;
    }
    container instructions {
        uses instruction-list;
    }
    <snip>
    }
}
```

```
grouping match {
    <snip>
    container "ethernet-match" {
        uses "ethernet-match-fields";
    }
    container "ip-match" {
        uses "ip-match-fields";
    }
    choice layer-3-match {
        case "ipv4-match" {
            uses "ipv4-match-fields";
        }
        case "ipv4-match-arbitrary-bit-mask"{
            uses "mask:ipv4-match-arbitrary-bitmask-fields";
        }
        case "arp-match" {
            uses "arp-match-fields";
        }
        case "tunnel-ipv4-match" {
            uses "tunnel-ipv4-match-fields";
        }
    }
    <snip>
}
```

The flow extraction engine extracts flow rules from both the configuration and operational data stores maintained by ODL. As discussed in Section III.A, since flow rules sent by all applications reside in configuration data store before they are sent to the devices, and the flow rules existing in the environment are present in the operational data store, listening to flow rules from both data stores helps the OFAnalyzer maintain a complete view of the flow rules present in the environment, especially in a distributed controller scenario. The source controller of the rules is noted so as to eliminate duplication. In addition, the flow extraction engine also listens for candidate flow rules from different applications running on the controllers, and stores them in the HashMap.

Once the flows have been extracted, each flow is given a unique identifier, making it easier to track the flow when analyzing conflicts, and for visualization purposes. The HashMap is then passed to the OFProcessor. This process is repeated for every controller present in the system.

In Algorithm 1, if the DTO contains elements of a candidate flow, its header fields are extracted. In addition, new fields of the data structure shown in Figure 6.2 is added to the flow rule fields. If the flow rule is configured using only layer-2 headers or layer-4 headers, then it is marked as being reconciled. This tag is then used in the conflict detection engine for determining the imbrication conflict.

C. Flow Prepping Engine

When the instruction set of a flow entry does not contain a redirection to another flow table, OpenFlow pipeline processing stops and the actions in the action set are executed. The actions are applied in the pre-determined order, per OpenFlow specifications [86] as follows: *a*) copy TTL inwards; *b*) pop all tags on the packet; *c*) push-MPLS tag onto the packet; *d*) push-PBB tag onto the packet; *e*) push-VLAN

Algorithm 1: Flow Extraction Engine

Input : *Rule* c, *Controller* C, *Strategy* s

Output : *procRule* r

Procedure `FlowExtract()`

```
1  r ← c
2  if c.l3match == ∅ then
3      if !c.l2match == ∅ then
4          r.reconciled ← TRUE
5  r.origController ← C
6  r.globalPriority ← getGlobalPriority(r.priority, C, s)
7  return r
```

tag onto the packet; *f*) copy TTL outwards; *g*) decrement TTL; *h*) apply all set-field actions to the packet; *i*) apply QoS actions; *j*) apply group actions; and *k*) forward the packet on the specified port. Trivially, anytime the final action is a deny, the remainder of the tags do not matter. However, an action set can only contains a maximum of one action of each type, unless defined as an `Apply-Actions` instruction. Unlike the action set, the actions of an `Apply-Actions` list are executed in the order specified by the list, and are applied immediately to the packet.

The flow prepping engine is primarily responsible for converting the candidate flow rule hashmap into a format that the flow rule conflict detection module expects. Knowing the processing preference of an OpenFLow action set, in the ODL based implementation the flow prepping engine atomizes the flow rule action sets, and stores the flow rule match address into a Patricia trie data structure.

Flow rule atomization involves the following steps:

- If a flow rule has an `Apply-Actions` list, then traverse the entire flow table pipeline and append to the actions set. If the flow rule uses a simple action set, then the complete chain of actions is compiled by processing the entire pipeline.

- Once the complete set of actions for a flow rule is determined, a duplicate flow rule with identical priority is created for each action, such that all rules have only one action.

- The flow rule actions, are next converted to one of two terminal actions - deny and forward. All actions aside from an explicit deny are considered to be a forward action, albeit with potentially modified header values.

Next, the address used in layer-3 match for each of the flow rules are stored in a Patricia trie data structure. Each edge in the Patricia trie is labeled with a bit, with each leaf node corresponding to the stored string. This string would be a concatenation of bits on a path from the root to this node. The trie provides storage and processing efficiency in due to its ability to collapse chains of nodes that have only one child simply by indicating how many bits should be skipped (i.e., what the length of the collapsed chain is). Each source and destination layer-3 address is stored into an octet wise Patricia trie. Thus each flow rule will have eight different tries associated with it[2]. To preserve semantic information, the leaf nodes in the Patricia trie store the unique identifier for every flow rule that has a match for that specific octet. Figure 6.7 shows the data structure being used to store the data set $\{1, 61, 63, 128, 223, 241, 248\}$.

D. Conflict Detection Engine

The Patricia trie data structure is used as shown in Algorithm 2 to determine and classify conflicts. Since we know that the layer-3 addresses are fixed length, we can follow along a path from the root to a matching node to obtain flow entries that match the layer-3 address space (packet space) of the flow being processed. In cases of wildcard matches, all child nodes of the matching node will represent flow

[2]Assuming an IPv4 based environment. An IPv6 environment would be modeled with one Patricia trie per hextet/quartet in each address

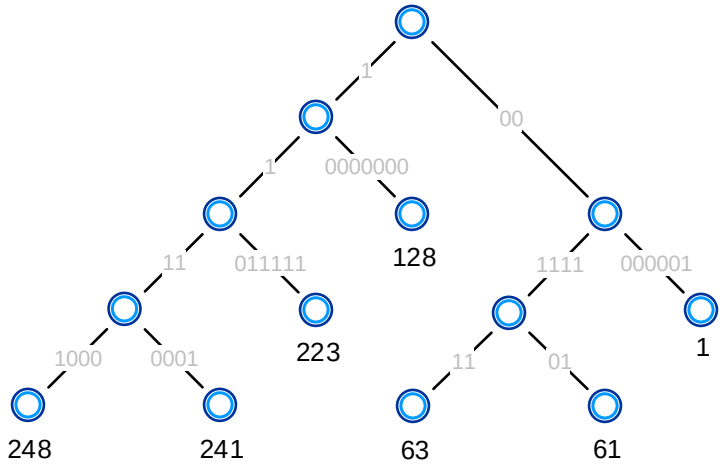

Figure 6.7: Use of Patricia Trie Data Structure for Octet-wise Representation of Layer-3 Address.

entries conflicting with the input flow. All detected conflicts are classified as shown in Figure 3.3 and Figure 3.4. Since we formally describe any overlaps involving reconciled rules as imbrication conflicts, we classify them as such and process them separately from non-reconciled rules.

Consider a single candidate flow rule r that is being processed by the conflict detection engine. First and foremost, the engine checks if the reconciled flag of r is unset. A Patricia trie search is done to determine if there is any flow rule in the existing table that has overlapping addresses. This search itself is the intersection of the results of eight different Patricia trie searches.

After the search space has been winnowed, a pairwise comparison is done between the flow rule r and γ, from the winnowed flow table set. If r and γ have differing layer-4 protocols, then no conflict exists between those rules, and the loop continues to the next rule in the winnowed flow table. If it has been determined that r and γ have the same protocol, the overlap in their address spaces is determined.

Algorithm 2: Conflict Detection Engine

Input : *Rule* r, *FlowTable* f
Output : Conflict-free *FlowTable* f'
Procedure `ConDet()`

```
1  if !r.reconciled then
2      F ← SearchPatricia(r.l3addr)
3      while Rule γ ∈ F do
4          if r.protocol != γ.protocol then
5              return AddFlow (f, r)
6          else if r.addr ⊆ γ.addr then
7              if r.action == γ.action then
8                  return ConRes (r, γ, f, Redundancy)
9              else if r.priority == γ.priority then
10                 return ConRes (r, γ, f, Correlation)
11             else if r.priority < γ.priority then
12                 return ConRes (r, γ, f, Shadowing)
13         else if γ.addr ⊆ r.addr then
14             if r.action == γ.action then
15                 return ConRes (r, γ, f, Redundancy)
16             else if r.priority == γ.priority then
17                 return ConRes (r, γ, f, Correlation)
18             else if r.priority > γ.priority then
19                 return ConRes (r, γ, f, Generalization)
20         else if r.addr ∩ γ.addr != ∅ then
21             if r.action == γ.action then
22                 return ConRes (r, γ, f, Overlap)
23             else
24                 return ConRes (r, γ, f, Correlation)
25 else
26     while Rule γ ∈ f do
27         if r.protocol == γ.protocol then
28             if r.addr ∩ γ.addr != ∅ then
29                 return ConRes (r, γ, f, Imbrication)
30     return AddFlow (f, r)
```

Classification of the conflicts looks at the following conditions:

- If r has an address space that is a subset of γ and their actions are the same, a redundancy conflict has been detected.
- If r has an address space that is a subset of γ and their actions are different then their relative global priorities determine the conflict. If priority of r is lower than γ the conflict is classified as shadowing, but if they have the same priority then the conflict is classified as a correlation.
- If γ has an address space that is a subset of r and their actions are the same, a redundancy conflict has been detected.
- If γ has an address space that is a subset of r and their actions are different then their relative global priorities determine the conflict. If priority of γ is lower than r the conflict is classified as generalization, but if they have the same priority then the conflict is classified as a correlation.
- If r and γ have overlapping address spaces but neither is a subset of the other, then similar action between the two flow rules will be classified as an overlap and different actions are classified as correlation.

Once the conflicts have been identified, the conflict detection engine encodes this information using CSV, with each comma separated value showing the unique identifier of the rule that has a conflict, and the type of conflict. The snippet in Listing 6.2 shows the results from using rule #1 and rule #2 from Table 3.1 as a candidate flow rule to the flow table consisting of rules #3 through rule #9.

Listing 6.2: Detected Conflicts Encoded as a CSV

```
1, 3.overlap; 4.shadow; 5.shadow; 6.correlation; 7.generalization;
   9.correlation
2, 3.generalization; 4.generalization; 5.generalization; 6.
   generalization; 7.generalization; 9.generalization
```

E. Conflict Resolution Engine

Once the conflicts between different flow rules have been detected, the conflict resolution process attempts to resolve these. The intelligible conflicts are resolved trivially and the interpretative conflicts are resolved using the resolution strategy that was determined by the administrator. Since resolution of interpretative conflicts is lossy, Brew has a manual mode, where administrator input using the conflict visualization functionality offered in the OFAnalyzer to help guide an informed decision. Visualization aids such as Figure 6.8 and Figure 6.9 assist administrators in making an educated decision regarding resolution of interpretative conflicts.

F. Visualization Engine

Visualization of conflict data from the OFProcessor was performed using `D3.js` JavaScript [138] and JSON libraries [139]. Results from the OFProcessor are obtained as a list of JSON objects, and are prepared for visualization using JavaScript conversion routines. Multiple visualization schemes then display this information to the administrator in a manner of his/her choosing, with a goal to display the information in a manner that is both intuitive and concise. The visualization engine is implemented as a module under the DLUX [127] user interface.

A hierarchical edge bundling [140] is used to represent the rule relationships using `D3.js`. This scheme highlights the overall relationship between all the flow entries while simultaneously reducing clutter. Figure 6.8 shows an example of the hierarchical edge bundling structure showing conflicts in a flow table, with the color of link distinguishing between the relationship between the rules. By hovering over the rule numbers that populate the perimeter of the circle in Figure 6.8, all flow rules that conflict that specific rule are highlighted. The color schemes indicate the priority of

Algorithm 3: Conflict Resolution Engine

Input : *Rule* r, *Rule* γ, *FlowTable* f, *String* $ConflictType$

Output : Conflict-free *FlowTable* f'

Procedure `ConRes()`

```
1  if ConflictType == Shadowing || ConflictType == Redundancy then
2      return f
3  else if ConflictType == Correlation then
4      if γ.globalPriority > r.globalPriority then
5          r.addr ← r.addr − γ.addr
6          f′ ← AddFlow (f, r)
7      else
8          f′ ← RemoveFlow (f, γ)
9          γ.addr ← γ.addr − r.addr
10         f′ ← AddFlow (f, r)
11         f′ ← AddFlow (f, γ)
12 else if ConflictType == Generalization then
13     f′ ← RemoveFlow (f, γ)
14     γ.addr ← γ.addr − r.addr
15     f′ ← AddFlow (f, γ)
16     f′ ← AddFlow (f, r)
17 else if ConflictType == Overlap then
18     r.addr ← r.addr + γ.addr
19     f′ ← RemoveFlow (f, γ)
20     f′ ← AddFlow (f, r)
21 else if ConflictType == Imbrication then
22     if γ.globalPriority > r.globalPriority then
23         r.addr ← r.addr − γ.addr
24         f′ ← AddFlow (f, r)
25     else
26         f′ ← RemoveFlow (f, γ)
27         γ.addr ← γ.addr − r.addr
28         f′ ← AddFlow (f, r)
29         f′ ← AddFlow (f, γ)
30 return f′
```

the conflicting rules. Rules highlighted in green have higher priority than the selected rule, indicating to the administrator that modifying this rule would not affect the others. Rules in red have a lower priority than the selected rule, serving to remind the administrator that any change to this rule would affect packet processing. Clicking on a rule number would provide details on the conflicts by loading the Reingold-Tilford tree [141] for that rule. Figure 6.9 shows a screenshot of an interactive Reingold-Tilford tree that presents the conflict details for a single flow rule in an aesthetically pleasing and tidy fashion. Hovering over the leaves of the tree would display more details about the rules, so the administrator can now make an informed decision by cross checking with those rules. Based on the assumption that the administrator is familiar with the coloring scheme described above, the administrator would learn details about the conflicts in red, by clicking on the rule number, and attempt to resolve them first (assuming administrator based resolution). Further details about the implementation of the visualization engine is detailed by Natarajan [126].

IV. Evaluation

The modules described in Section III were implemented in JAVA. The `L2Switch` project was employed to connect the ODL Lithium controller to the OVS. While OVS and ODL Lithium support both OpenFlow 1.0 and OpenFlow 1.3, testing was done only using OpenFlow 1.3. Our implementation correctly identifies flow rule conflicts and classifies them. Both intelligible and interpretative conflicts are automatically resolved using the least privilege resolution strategy.

Brew was evaluated experimentally using a 2.5 GHz Intel Core i7 machine with eight dedicated cores and 16 GB DDR3 memory. The system was hardware virtualization enabled (VT-x), and running Ubuntu 14.04 OS running Linux kernel 3.1.

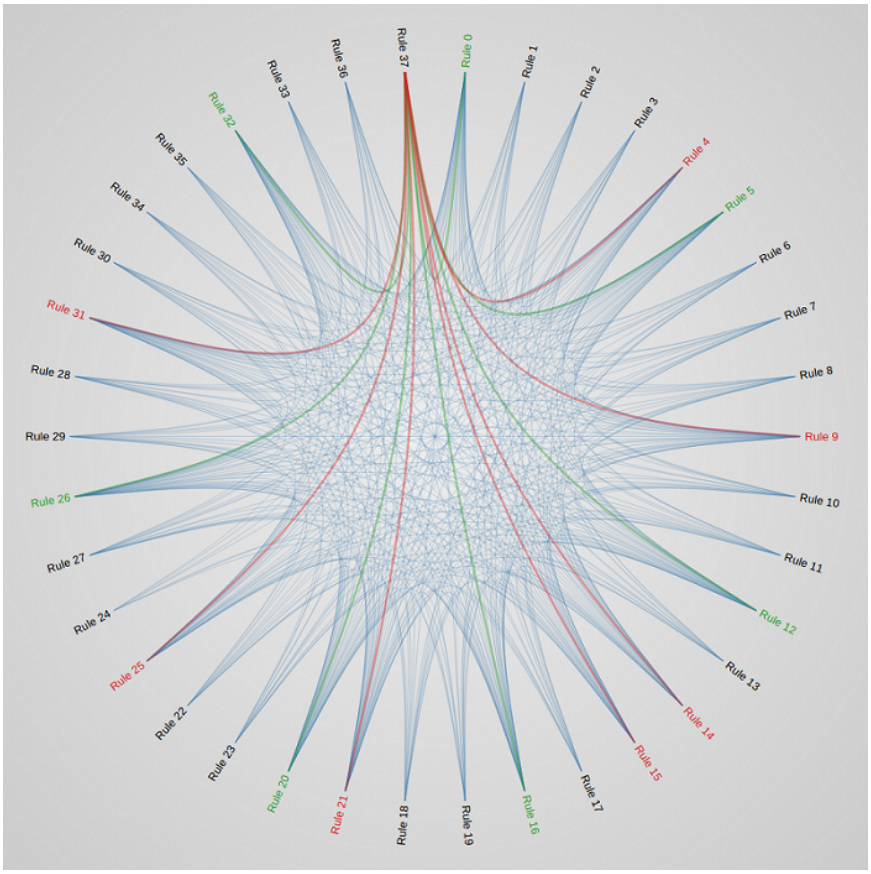

Figure 6.8: Conflict Visualization Based on Hierarchical Edge Bundling Showing a Spiro-graph.

A. Theoretical Evaluation

Both the conflict detection and resolution algorithms grow in a linear in time, except for the Patricia trie lookup and insertion time. The time complexity of a lookup on a Patricia trie depends on the length of the string (constant in our case) and the number of flow rules; for a total runtime of $\mathcal{O}(n)$ [128], where n is the number of entries in the flow table. In an environment with n atomic flow rules existing in the environment; and k new candidate atomic flow rules introducing change into the environment, Brew would have a run time of $\mathcal{O}(n.k)$.

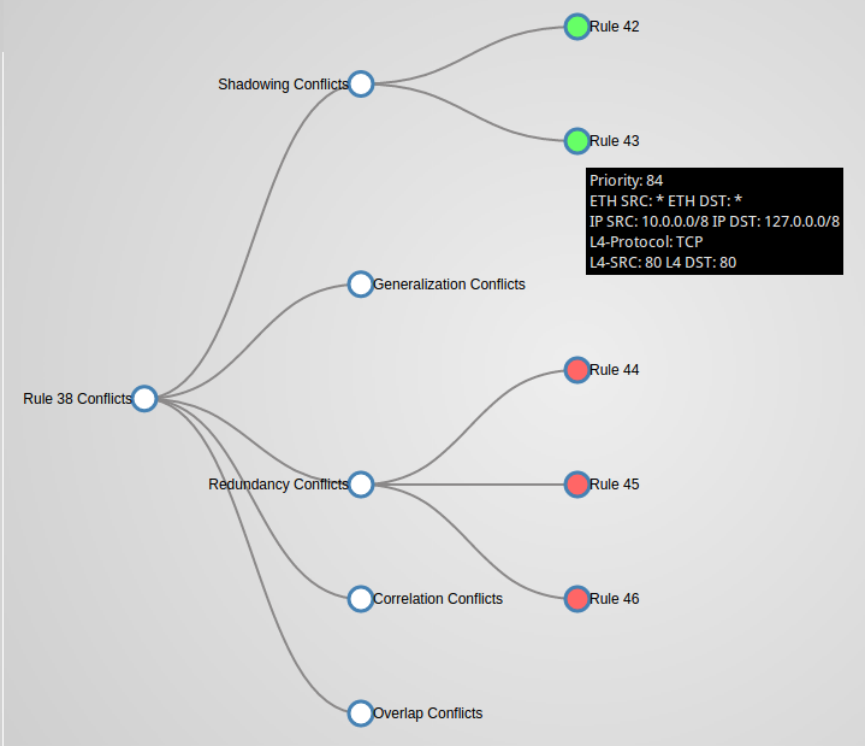

Figure 6.9: Conflict Visualization Based on Reingold-Tilford Tree.

B. Correctness Verification

OFAnalyzer was evaluated for correctness by providing it with several of rules that were known to have conflicts. Brew correctly identified flow rule conflicts and classified them, including transient cross-layer conflicts. The relationship between the different conflicts were displayed using the visualization techniques discussed. The classification was manually verified to be accurate.

A simple network with topology consisting of eight virtual hosts in different VLANs, connected to seven different OVS was implemented on Mininet [142], a tool used to create rich topologies and instantiate OVS and virtual hosts, using a python script. Figure 6.10 shows the topology used. A singular ODL controller acted as the SDN controller. The `l2switch` project in ODL uses `openflowplugin` to communicate between the OVS and the controller using the OpenFlow protocol.

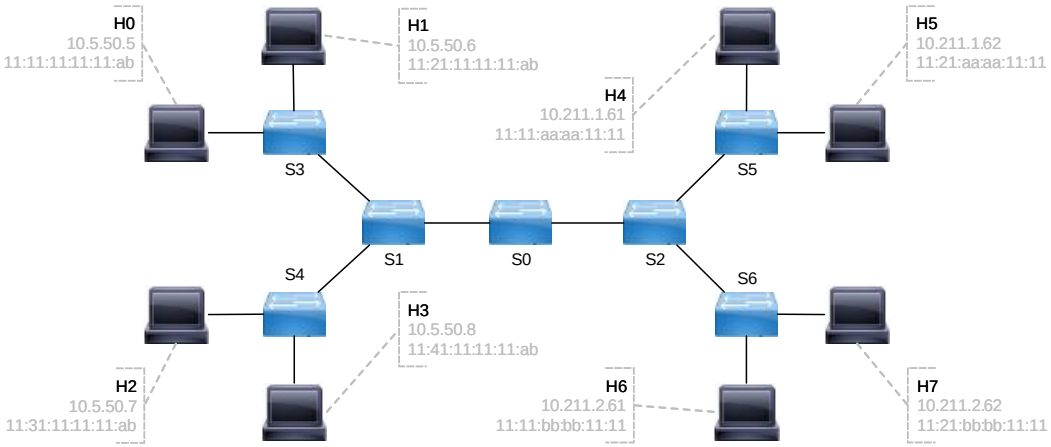

Figure 6.10: Topology Used for Brew Correctness Verification.

A flow rule table consisting of 100 flow rules was implemented to obtain the desired security policy and constrain traffic flow. The test dataset had all conflicts present in an effort to have Brew identify it correctly. The frequency of the different types of conflicts in this dataset was as follows: *a*) Shadowing - 10%; *b*) Redundancy - 10%; *c*) Correlation - 20%; *d*) Overlap - 20%; *e*) Generalization - 20%; and *f*) Imbrication - 20%. Manual verification showed that all present conflicts were detected by Brew.

C. Performance Overhead

Once correctness of our work was verified and validated, we analyzed the performance overhead of conducting inline rule conflict analysis. Once again, the topology shown in Figure 6.10 was used for the experiment. The different link bandwidths were enforced using the `tc` command on Linux. This setup allows us a fine-grained control on the network. A large text file of size 1 GB was sent from host A to host D, with a script attempting to add flow rules into the environment. Figure 6.11 shows the time taken to transfer the file when flow rules were: *a*) inserted in a native environment, without the Brew module running; *b*) inserted with Brew running, but the rules

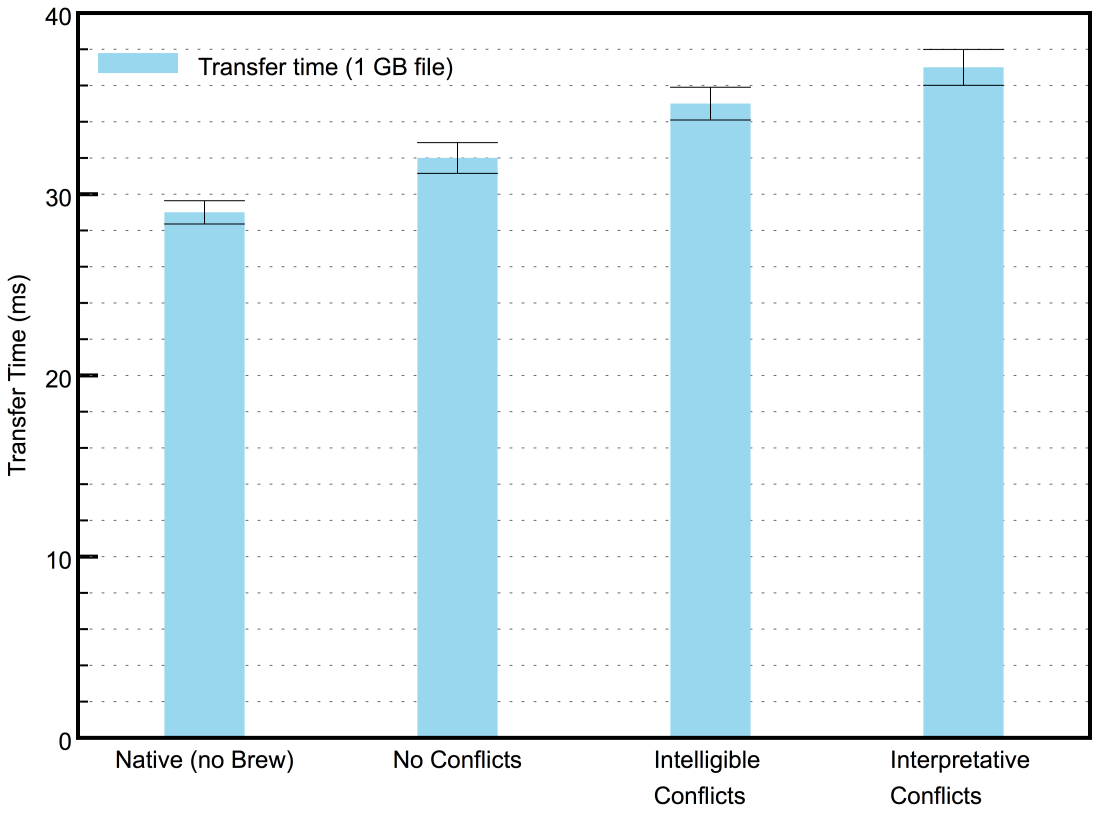

Figure 6.11: Network Performance Overhead.

themselves were conflict free; *c*) inserted with Brew running, and had intelligible conflicts; and *d*) inserted with Brew running, and had interpretative conflicts which were resolved using least privilege resolution strategy. Each test was run 100 times, and the transfer time was averaged to obtain the file transfer time for each case, with the error bars indicating the range of the time taken.

As expected, when interpretative conflicts were to be resolved, the transfer took longer, due to additional computational needs on the system. Scrutinizing the data showed that the identification and resolution of intelligible conflicts earlier on in the chained processing added to this impact. However, the presence of the Brew module itself caused about 10% increase in transfer time (average of 100 test runs).

In large SDN-based cloud environments, this trade-off would be acceptable since having a conflict free flow table will not only ensure greater confidence in security, but also optimal packet forwarding processing times. However, in small to medium size environments, this overhead could be substantial enough to deter adoption.

D. Scalability Evaluation

The same input file that was used for verifying correctness (containing 100 atomic flow rules), on the topology shown in Figure 6.10 revealed the processing time of about 5.6 μs per rule.

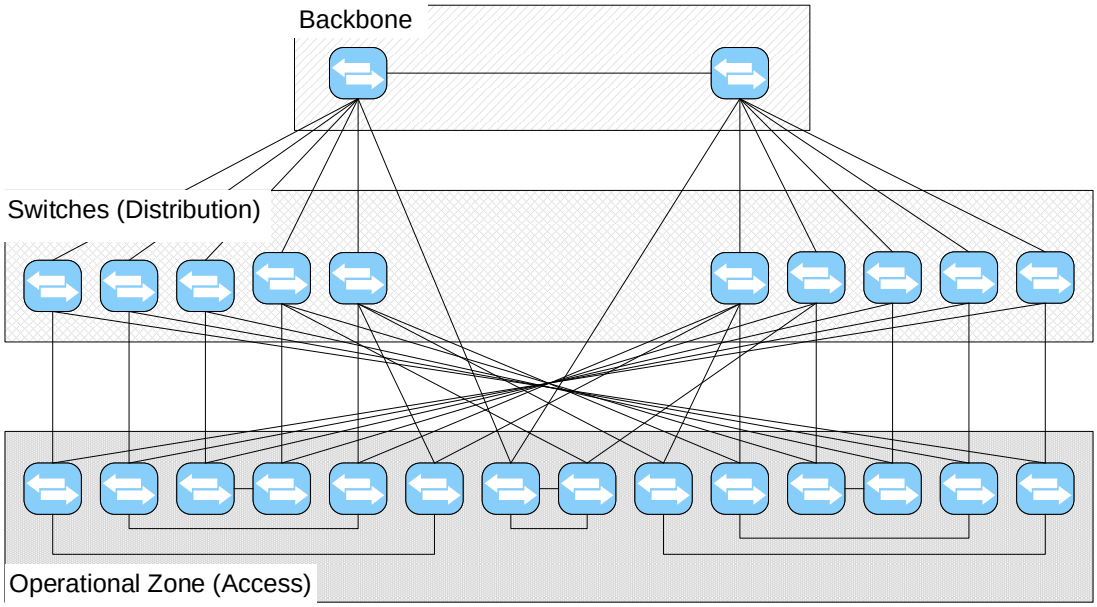

Figure 6.12: Topology for Scalability Testing (Replicating Stanford University Backbone Network).

Next, a real-world topology was used to test scalability. The Stanford University backbone network [74] was used as a representative mid-size enterprise network. The network consists of fourteen access-layer[3] routers connected using ten distribution

[3]lowest level of the Cisco three-tier networking model [143]

switches to two backbone routers. The snapshot of the routing tables and configuration files showed over 12,900 routes, 757,000 forwarding entries, 100 VLANs and 900 access-list rules. This network was replicated in Mininet using OVS to replace all the switches and routers, but retaining the connectivity information, as shown in Figure 6.12. Translating all relevant rules into equivalent OpenFlow rules resulted in approximately $8,900$ atomic flow rules, which were then used to run scalability tests. These $8,900$ flow rules were used as the source to extrapolate and generate flow rule tables of size $10,000$ to $100,000$. The extrapolation process randomly picked out the rules from the $8,900$ atomic flow rule set.

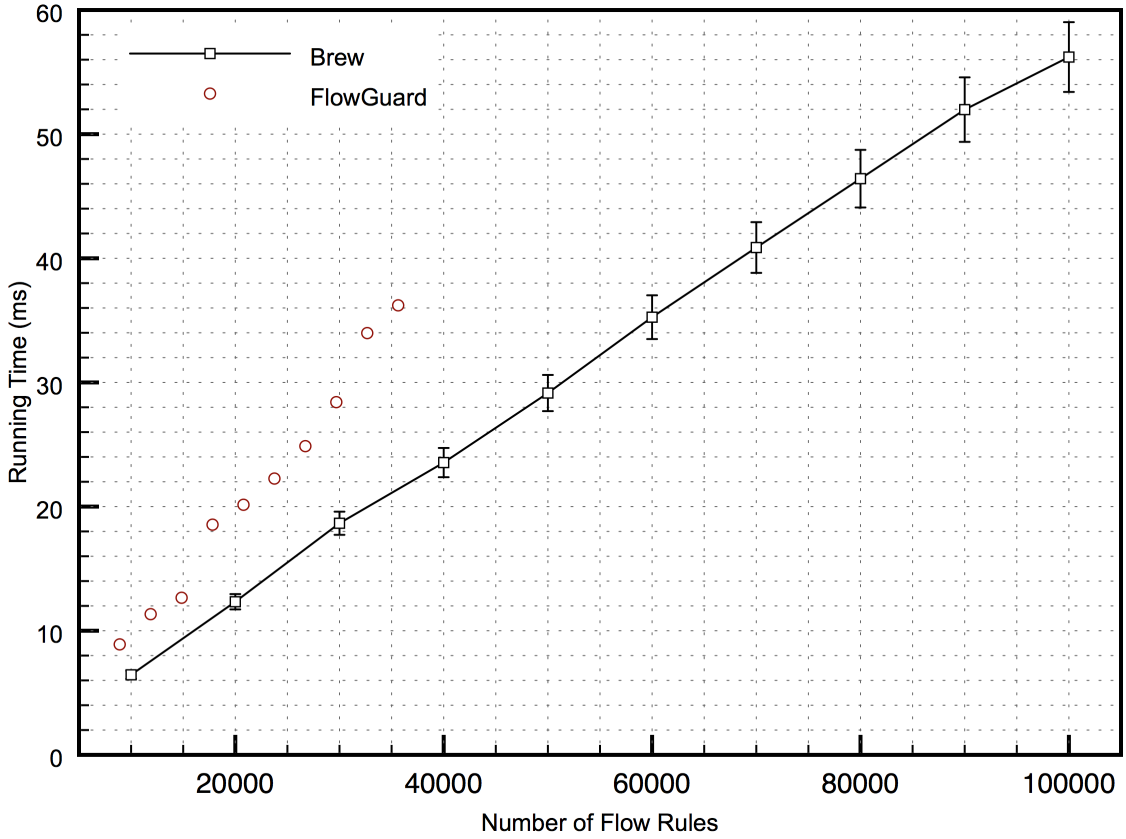

Figure 6.13: Increase in Running Time with Increase in Flow Table Size.

With an input file containing about $10,000$ atomic flow rules, the processing time was about 5.6 ms. Rules were further replicated and inserted into the system to observe growth of computation time. Figure 6.13 shows results from our experiment runs using different input flow table sizes. Ten different test runs were conducted on flow tables of size varying from $10,000$ to $100,000$ rules, and the resulting running times were averaged to get the results in the plot. The results clearly show a $\mathcal{O}(n)$ running time and reveals that Brew effectively identifies flow rule conflicts, and takes corrective action in spite of the large data sets. Comparative running times for FlowGuard are obtained from [12]. Run times for FortNox are not available and the algorithm complexity is not discussed, but evaluation appears to suggest linear growth; albeit considerably slower (approximately 8 ms per $1,000$ flow rules, as opposed to 0.56 ms per $1,000$ flow rules for our system). Running time evaluation for VeriFlow also appears to be linear, but considerably greater, with 1 ms per 10 flow rules.

Interestingly, none of the conflicts detected from the Stanford topology was categorized as Imbrication. This can be attributed to all the rules using layer-3 addresses for matching, as is customary in traditional environments.

E. Effect of Decentralization Strategies

We studied the impact of the distributed environment on policy conflict detection and resolution using two different experiments. First, we changed the controllers from a single controller to $2, 5$, and 10 using the host-based partitioning scheme. Using the same Stanford backbone network data set [74], the effect of moving from a single controller to a decentralized controller was studied. At each data point of $20, 40, 60, 80$ and 100 thousand total rules, the controllers were assigned flow rules equally. As expected, the running time increased with an increase in the number of controllers, with Brew taking more than twice as much as it did in a single controller environment.

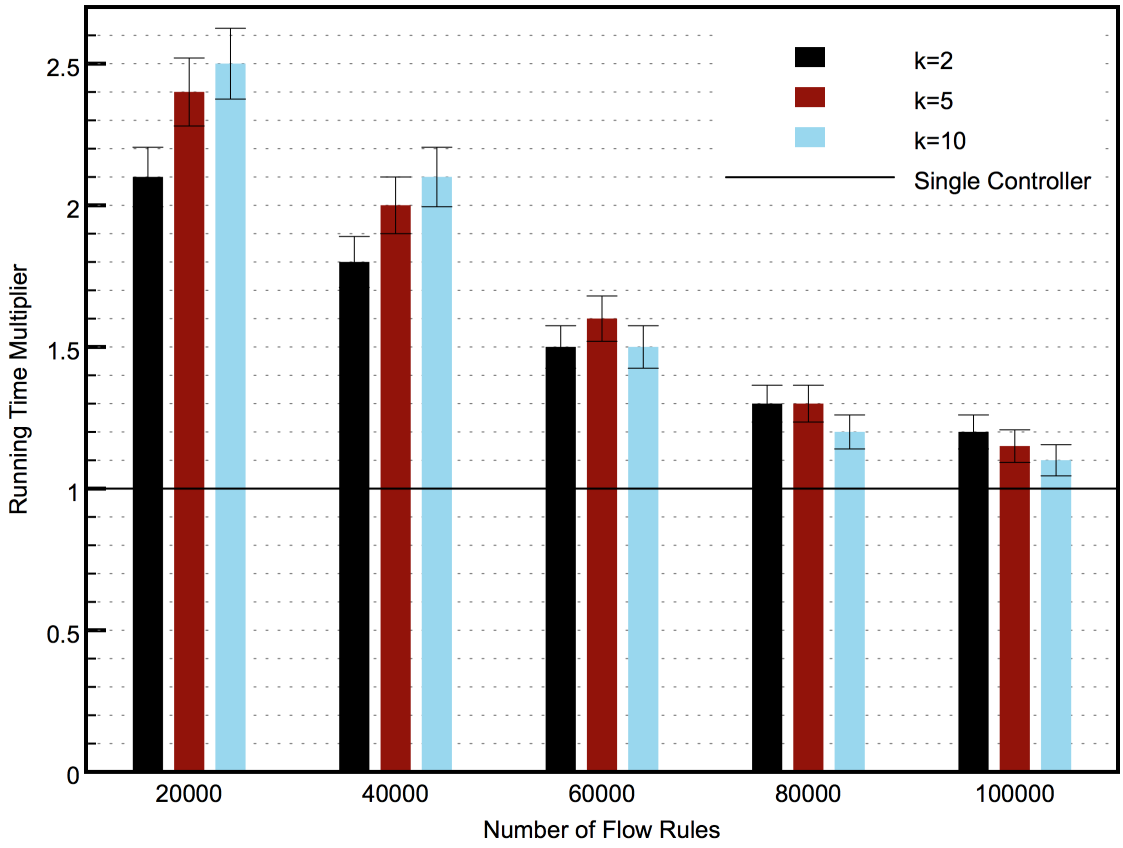

Figure 6.14: Change in Running Time for Brew with k-controllers.

However, with an increase in the number of flow rules, the running time in a distributed controller environment asymptotically approached the single controller scenario, as shown in Figure 6.14. This is attributed to each controller having to deal with fewer flow rules as the number is increased. The graph also confirms the intuitive fact that with increasing number of flow rules, having more controllers improves performance.

Next, we tested a distributed controller scenario using the application partitioning paradigm. Flows were injected into the controller with weighted priorities giving flows generated from a simulated security application highest preference. The OF-Analyzer extracted flows from the different controllers, and the OFProcessor used the global priorities to make decisions as expected. Similar tests were also run using

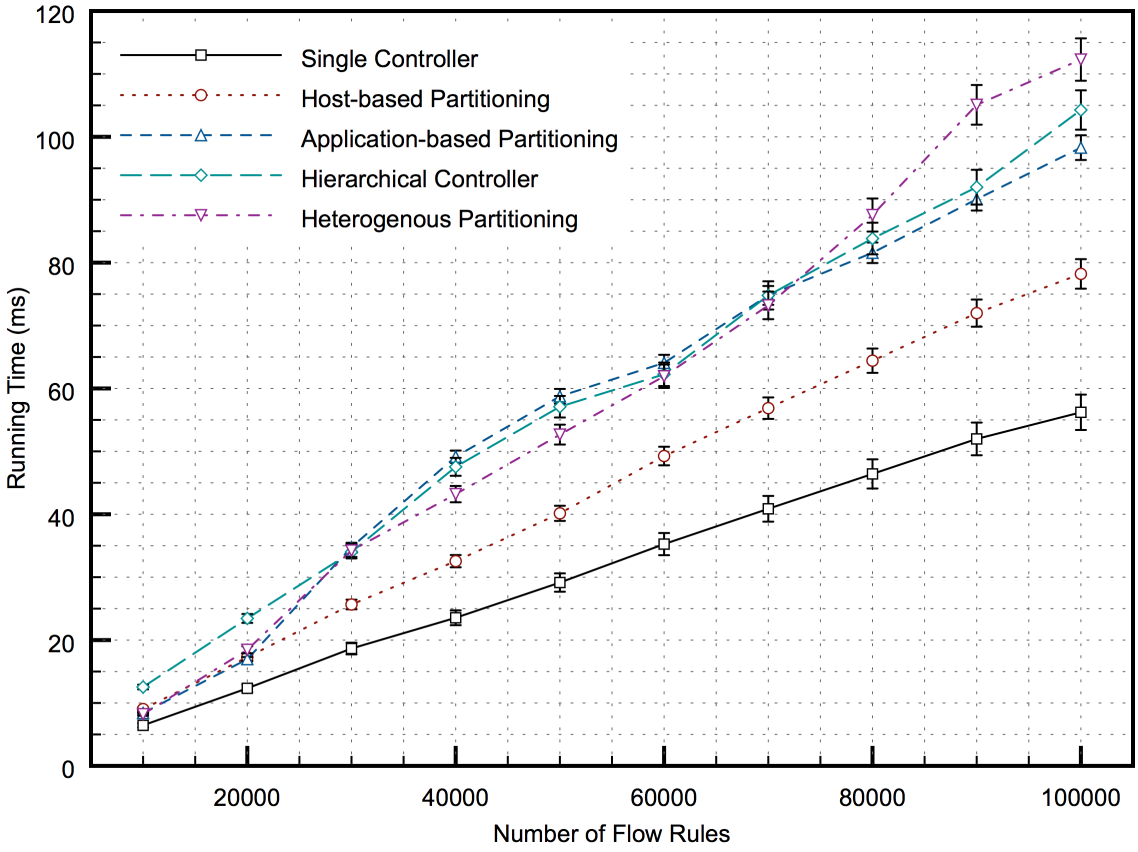

Figure 6.15: Dependence of Flow Rule Conflict Resolution Times on Decentralization Strategies for Increasing Number of Flow Rules.

the hierarchical controller paradigm with results matching expectations. Figure 6.15 shows the running times for the conflict detection algorithm over the same input set of flow rules running on an application partitioning, host partitioning and hierarchical distribution strategies. While all scenarios show a near linear growth in running times with the number of flow rules in the table, the host-based partitioning scenario was noticeably faster. We attribute this to the presence of a distributed mesh control plane for the application and hierarchical/heterogeneous partitioning scenarios while having a hierarchical control plane in the host-based controller partitioning scenario.

Chapter 7

CONFLICT-FREE COUNTERMEASURE GENERATION FOR MTD IN DISTRIBUTED SDN CLOUDS

Techniques that assess the security state of the entire environment, and proactively make changes to reduce the likelihood of being compromised are known broadly as MTD initiatives. MTD techniques are facilitated by platforms that provide flexible and programmable features, and SDN is ideal for this effort. The ease of programmability in SDN makes it a great platform for implementations involving application deployment, dynamic topology changes, and decentralized network management in a multi-tenant data center environment. Implementing MTD measures in an SDN-based environment, however, leads to scenarios where conflicts could occur between newly inserted or modified flow rules to the existing policies. Verifying and resolving these conflicts are time consuming, and gets even harder in a distributed SDN environment, making it unsuitable in highly dynamic environments. In this work, we provide a flow rule conflict avoidance mechanism that eliminates this problem in MTD implementations in highly dynamic SDN environments. Our appliance, named CaCTuS, is implemented on an OpenDaylight controller and ensures that no two flow rules have conflicts at any layer, thereby ensuring system stability. Evaluation results show that CaCTuS could craft MTD countermeasures that were provably conflict free with rules in the existing flow table, with an acceptable 3% run time overhead.

I. Problem Statement

MTD [144, 145] is a transformative approach to security of multi-tenant cloud environment that leverages dynamism to create an environment with a changing attack surface. By presenting attackers with an unpredictable target, cloud service providers

hope to make it difficult for an exploit to have the desired malicious behavior [146]. The flexibility and programmability afforded in the SDN paradigm can be conformed to achieve a dynamic defensive strategy based MTD [147] by systematically selecting countermeasures to prevent or mitigate attacks [148, 149].

In an IaaS cloud, VMs are managed by tenants and may contain various vulnerabilities, thus making them easy targets for attackers. Chung et al. [150] present an MTD approach to automate an iterative three-step procedure to counter network attacks: *a*) network intrusion detection; *b*) threat analysis; and *c*) countermeasure selection and deployment. Chowdhary et al. [102] use an attack graph based vulnerability analysis model to enumerate all possible attack scenarios, allowing the cloud system to select countermeasures before identified vulnerabilities are exploited. While policy conflicts between the proposed countermeasures and existing rule set is checked, and resolved, the temporal nature of such changes and its timeliness is not considered. In other words, since verifying and deploying the countermeasure takes a non-zero amount of time, the system should have strategies in place to reduce the time to verify and deploy a countermeasure and ensure the system is safe from scenarios where it attempts to implement countermeasures faster than the system can propagate the changes through the environment.

The dynamism in MTD gives rise to the need for a framework to accurately, and in a timely fashion, examine the complex relationships between various hosts and ensure that any changes made to the environment do not conflict with security policies. While the timeliness challenge in MTD is actively explored [151], the discussion of timeliness is limited to ensuring that the MTD system should evolve faster than time for reconnaissance. Timeliness for system stability is ignored. In other words, ensuring that the system stabilizes at a fast-enough rate that customers and users do not feel the adverse effects of the underlying system changes is not researched in any detail.

Further, in case of an SDN-based cloud environments where each countermeasure is implemented using accompanying flow rule changes, timeliness is dependent on: *a*) verifying that the flow rules do not conflict with existing flow rules; *b*) adding the flow rules to the controller; and *c*) propagating the changes to the OVS. Since the last two factors are organic to any SDN environment, reducing the time required for system stabilization requires ensuring that the generated countermeasures do not conflict with existing flow rules, thereby eliminating the need to run the conflict-checking process. Since the best of these algorithms are linear in nature to with number of flow rules in the table, in large distributed clouds, conflict checking of policies is time consuming.

Tackling the timeliness issue in implementing MTD countermeasures in SDN environments is a two-fold problem. First, ensuring a conflict-free environment is time consuming. And second, there is no proven way to resolve cross-layer policy conflicts. Since there are no current solutions that address either of these issues, an ideal solution would seek to avoid the problem in the first place. As the adage goes, prevention is better than cure. Thus, the MTD system should seek to produce countermeasures that are provably conflict-free with current security policies.

A final consideration in implementing MTD countermeasures to SDN environments involves usability. If the implemented MTD countermeasure were to impact the functionality of the system or affect user experience because of either the technique being used, or because of the overhead, its adoption will be limited [151]. To that end, ensuring that the system stabilizes before introducing new changes into the environment is paramount. To summarize, through this work the following challenges need to be addressed:

- Reduce the time to deploy countermeasures in an MTD system by ensuring that the network address dynamism is within allowed ranges that guarantee no address space overlap.

- Propose a mapping schema between layer-2 and layer-3 to pre-empt and resolve the cross-layer conflicts.
- Ensure that the environment stabilizes and passes user traffic prior to a new countermeasure being implemented.

II. Moving Target Defense (MTD)

MTD techniques are a proactive approach to security of multi-tenant cloud environment that leverages the dynamism in computer systems to create an environment that has a changing attack surface, with the basic idea being either: *a*) hiding system properties that are required by attackers to leverage an exploit; or *b*) alter system properties periodically that makes all previous reconnaissance done by attackers moot. Existing Moving Target Defense schemes can be broadly classified into [152]: *a*) diversity based MTD; and *b*) dynamicity based MTD. The first class essentially seeks to enlarge the attack surface while the latter consists of moving the attack surface at runtime to force a re-evaluation by the attacker. This work pertains to the latter.

Dynamicity based MTD techniques introduce temporal changes to the environment, which has the effect of rendering previous reconnaissance done by attackers useless. Network based MTD schemes are discussed in [153, 154, 155] but they require coordination from the communicating parties, and hence are not suitable for a cloud environments that are gaining an ever larger share of the market. Antonatos et al. propose a network address space randomization scheme to offer an IP hopping approach [156] to protect against hitlist worms. Al-Shaer proposes a more generalized system that enables IP randomization [157] to keep attackers from knowing the true location of the systems. Address space randomization was then customized for an SDN environment [149]. Duan et al. present random route mutation technique which enables random changes of the routes in a network [158].

While analyzing MTD schemes, Hobson et al. [151] identifies timeliness as an important challenge. This is a two-pronged challenge, requiring the MTD system to evolve faster than time for reconnaissance, and at the same time ensuring that the system stabilizes at a fast enough rate that customers and users do not feel the adverse effects of the dynamicity. This work focuses on a timeliness aware SDN-based implementation of network dynamicity based MTD systems, where the dynamicity is introduced by IP randomization.

III. System Model

In this section, details of the conflict-free countermeasure generation framework, including design considerations, assumptions, system models and architecture details are discussed. The functionalities were implemented into a holistic conflict-free countermeasure generation suite called CaCTuS (Conflict-free Countermeasure generaTion Suite). Figure 7.1 shows the black-box model for CaCTuS. Section III.B provides a detailed system level description and Section IV provides implementation details of CaCTuS.

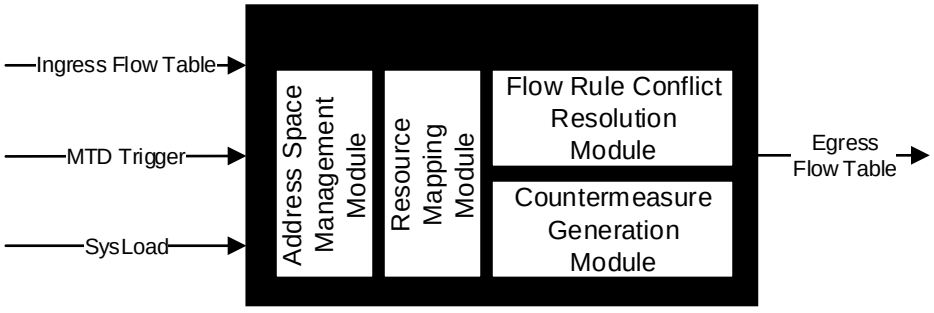

Figure 7.1: Black-box Model for CaCTuS.

A. System Assumptions

This work considers only address hopping based network MTD approach. The assumptions made are:

- The network address change is transparent to benign users, who do not feel any impact from the network dynamicity.
- There exists an authentication mechanism between applications that need to access the API from the address space management module. Since this work focused on establishing functionality, optimal authentication mechanisms were not considered. However, without an authentication mechanism in place, exposing the available address space to tenant applications could be misused by malicious insiders.
- There exists a mechanism to obfuscate the relationship between the layer-3 address and the systems present in the data center to prevent reconnaissance done by external adversaries. This could be accomplished by having a NAT type service at the egress of the data center network.

B. System Components

In this Section, the components that help achieve conflict-free countermeasure generation using a modularized approach are discussed. The system consists of five interrelated modules, namely an Address Space Management Module (ASMM), a Resource Mapping Module (RMM), a Countermeasure Generation Module (CGM) and a Flow Rule Conflict Resolution Module (FCRM).

1) Address Space Management Module (ASMM): The ASMM is primarily responsible for allocating layer-3 addresses to requesters in contiguous blocks. While the

implementation in CaCTuS uses IPv6 specifically, it can just as easily by converted to an IPv4 implementation. Algorithm 4 shows the logical details of the ASMM.

The address space allocation itself uses the Buddy memory allocation [159] algorithm to manage the address space. Borrowing from the memory allocation technique, the network address space is divided into blocks, of specific size n. Each block has 2^n available addresses. Instead of attempting to conserve address spaces, just as in memory allocation, a block is always split into two equal blocks that are both half the size of the larger block. This, in effect, creates two blocks that are unique buddies to each other. When both the buddy blocks are not being used, they are merged together to form the larger block they were split from. In this implementation, the value is set as $n = 2$, giving a block size of 4 IPs, thus, leading to a total of 30 orders[1] for IPv4. When using IPv6 in the address management module, where the address space is 2^{128}, gave us 126 possible orders.

When the CGM requests a new address space, the ASMM looks for two pieces of information, i.e. the number of addresses required, and list of current address space being used. Similar to the Buddy memory allocation algorithm, if a contiguous address space of an equal or larger size is available, it is provided to the GCM. If not, a block larger than the requested address space is split into half. This process continues recursively until a suitable address space is found.

2) Resource Mapping Module (RMM): Functions of the RMM involve implementing a mapping function between layer-2 addresses and layer-3 addresses, or maintaining a large MAC-address table. However, deviating from traditional practices, a preset layer-3 address is assigned to every possible layer-2 address. As seen in Chapter 5, resolving cross-layer conflict is challenging, owing to the transient nature of the mapping between

[1]Order is an integer such that the size of a block of order α is proportional to 2^α

Algorithm 4: Address Space Management Module (ASMM)

```
Procedure ASMM()
    Input  : requestSize x, currentBlock B
    Output : addressSpace A
1   if B == NULL then
2       AllocateL3Address(x)
3   else
4       x ← x + B.size
5       FreeL3Address (B)
6       AllocateL3Address(x)

Procedure AllocateL3Address()
    Input  : requestSize x
    Output : addressSpace A
1   reqBlock ← x/blockSize
2   while slot do
3       if slot.size ≤ reqBlock * 2 then
4           if slot.size ≥ reqBlock then
5               slot.useStatus ← 1
6               return slot
7       else
8           if slot.size ≥ reqBlock then
9               while slot.size ≥ reqBlock do
10                  addSlot(slot.startAddress, slot.size/2)
11                  addSlot(slot.startAddress + slot.size/2, slot.size/2)
12                  delSlot(slot.startAddress, slot.size)
13                  slot.size ← slot.size/2
14              slot.useStatus ← 1
15              return slot

Procedure FreeL3Address()
    Input  : addressSpace slot
1   slot.useStatus ← 0
2   flag ← 1
3   while flag do
4       if slot.buddy.useStatus == 0 then
5           combineBuddy(slot.startAddress, slot.size)
6       else
7           flag ← 0
```

layer-2 and layer-3 addresses. However, by removing the transient nature of that relationship, conflict resolution and conflict prevention becomes a manageable problem. This is accomplished by strictly using IPv6 for layer-3 addresses.

IPv6 specifications call for a 64-bit interface identifier that is either automatically generated from layer-2 address, or assigned manually. The interface identifier address is assigned manually, albeit using the layer-2 address. The 48-bit layer-2 address is turned into a 64 interface identifier by inserting `FF:FE` in the middle, as per IPv6 convention. However, the address is not made globally unique by flipping the 7th most significant bit, since this work only uses the addresses locally, with a mapping back to IPv4 at egress.

3) Countermeasure Generation Module (CGM): A countermeasure is an action or a series of actions intended to thwart attacks or make the system sturdier against attacks, wherein network configurations and traffic policies are changed. When needed, one (or more) candidate countermeasures are chosen amongst many potential countermeasures for deployment after weighing attributes such as cost, time to deploy, and potential impact to system performance or availability [148]. While common network-based countermeasures involve actions on several OSI layers, this work is limited to network address hopping.

When invoked on schedule, the CGM uses input from the ASMM to generate countermeasures that are conflict-free. With regards to the current flow rules, any selected address change can have: *a*) overlapping address space, but different flow rule action; *b*) disjoint address space, but same flow rule action; *c*) disjoint address space and different flow rule action. In the latter two situations, it is guaranteed that the environment remains free of flow rule conflicts since the address space is disjointed. In situations where the same address space is returned from the ASMM, the CGM

requests a new address space. This simple asynchronous check was introduced to alleviate concerns in a distributed controller scenario, where due to the controller code the ASMM might not be in sync.

When the system is under attack, the CGM can quickly introduce change into the environment, with the trade-off that conflicts may be introduced into the environment. These conflicts may be resolved at a later time using the Brew framework from Chapter 6.

4) Flow Rule Conflict Resolution Module (FCRM): Since flow rules with conflicting actions are avoided, the FCRM only has to reconfigure the conflicting flow rules to remove address space overlaps. The Brew framework discussed in Chapter 6 serves as the FCRM.

IV. Implementation

Figure 7.2 shows the logic flow in the system, implemented in an SDN environment using ODL controller. The CMG module generates the flow rules required that would implement MTD. MTD countermeasures in the system are invoked in one of two different ways: *a*) if the time since the implementation of the last MTD measure is above a certain threshold; or *b*) an attack trigger has been set due to the environment detecting it is under attack. The strategy in both cases is not markedly different. However, when the system is under attack, the CGM picks the MTD countermeasure most likely to stop the attack without concern for its potential to cause conflict. Thus, the system prioritizes the implementation of *any* MTD strategy over finding the optimal strategy. When the system is not under attack, the system utilizes the ASMM to generate an MTD solution over address spaces that guarantee no conflict.

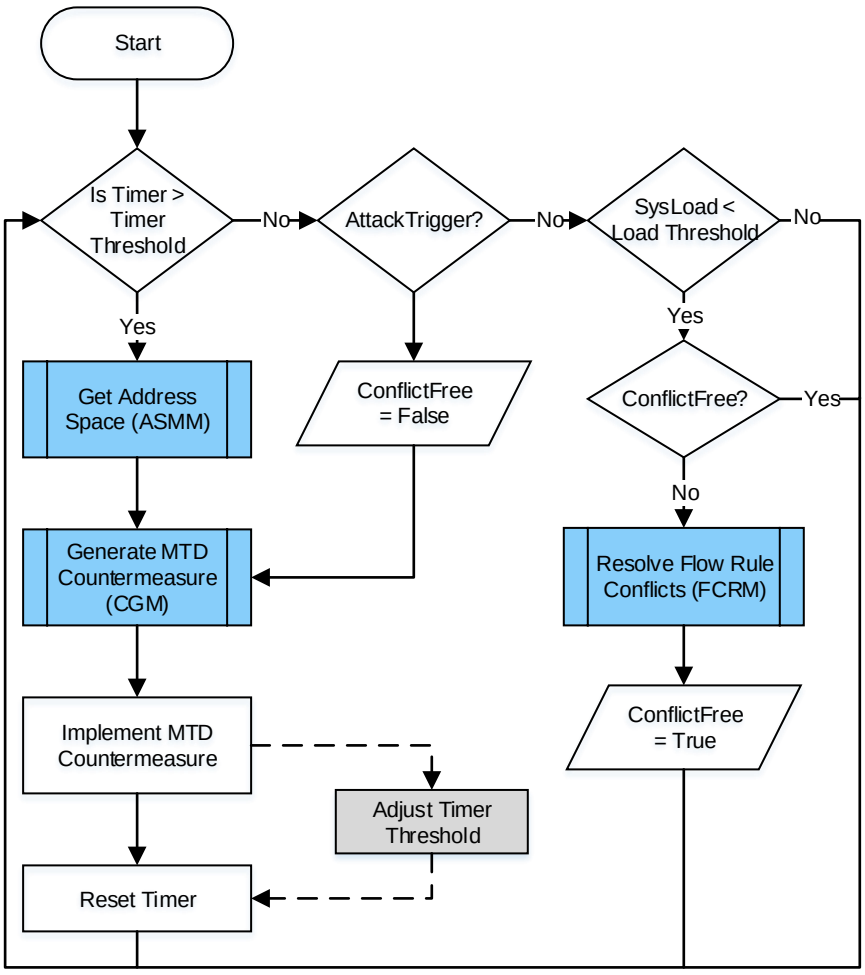

Figure 7.2: System Logic Flow in CaCTuS.

In dynamic SDN environments, a potential issue is countermeasure *bounce*. Analogous to a flickering light bulb, situations may arise in which the system may react to a transient attack event by reconfiguring from state S_0 to a different state S_1, only to encounter another event and attempt to revert to the original state S_0. Since the system stabilization after each reconfiguration takes a minimum of non-zero time t, such a scenario would cause system instability potentially placing the system in a death spiral. This situation is handled by implementing a wait timer for generation of new countermeasures. While this value is currently configurable, learning this value

for an environment using machine learning techniques is has great potential.

In situations where the best MTD measure was not implemented due to either the system being under attack, the potential for conflicts being present in the system exists. These are addressed by the FCRM by combining/splitting out flow rules that are affected by the conflicts.

V. Evaluation of CaCTuS

Evaluation of CaCTuS was done in a multi-faceted manner. First, the hypothesis that generating multiple flow rules would result in address space overlap which in turn would lead to flow rule conflicts was tested. Next, the effectiveness and correctness of CaCTuS was evaluated. Following that, the running time for CaCTuS was compared to flow rule conflict resolution mechanisms, and the effect of block size on computation time and address space utilization was studied. Finally, the overhead due to CaCTus was measured in an environment using file transfer time as the metric, and a comparison was made to the overhead added by the Brew module. The results show the promise CaCTuS holds in a highly dynamic SDN environment implementing address hopping as an MTD strategy.

The first experiment evaluated the prevalence of address space overlaps during countermeasure generation. To effectively demonstrate the probability of conflict, IPv4 was used instead of IPv6, since an address space collision using IPv6 address space would require about 2.6×10^{18} flow rules for a 1% collision (birthday paradox). The result is shown in Figure 7.3.

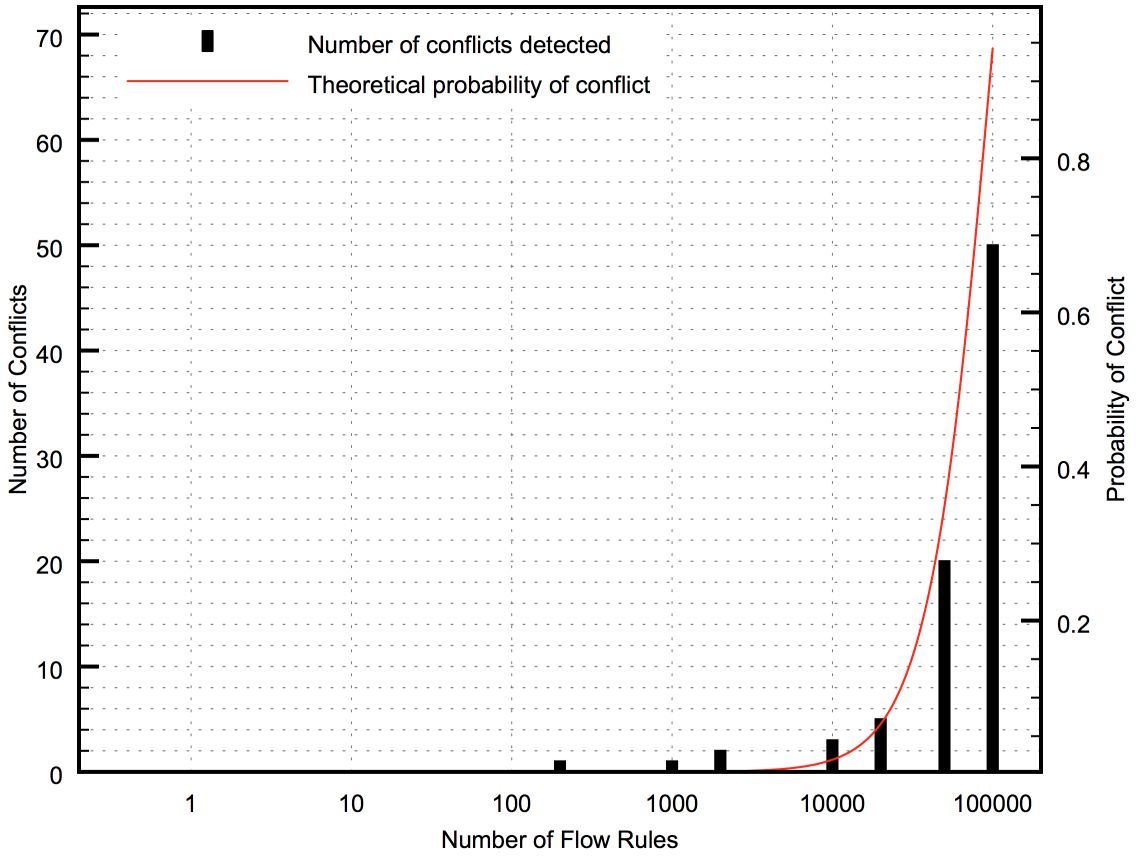

Figure 7.3: Change in Probability of Flow Rule Conflicts with Flow Table Size.

The next experiment focused on the effectiveness of the flow rule conflict prevention. The topology shown in Figure 6.10 was implemented on Mininet using a python script. ODL Lithium was used as the OpenFlow controller and the `L2Switch` project was employed to connect to the OVS. Since the test topology consisted of only eight hosts, the network address space was limited to having only 16 possible addresses. Given the simplified system, it is expected that flow rules use overlapping address spaces about half the time. Correctness was provably demonstrated over the course of 100 test runs, when CaCTuS produced MTD schemes without any conflicts, while natively 47 of the 100 runs produced flow rules using an overlapping address space.

Next, the running time of CaCTuS was compared against flow rule conflict detection tools and plotted in Figure 7.4. CaCTuS compares favorably to Brew (discussed in Chapter 6) and FlowGuard [12] at larger number of flow rules. This appears intuitive when looked into in conjunction with results from Figure 7.3 since the probability of conflicts occurring in a system rise exponentially with an increase in the number of flow rules. Experiments performed were based on a real-world network topology derived from the Stanford network as obtained by Kazemian et al. [74]. The ASMM module grew in a $\mathcal{O}(\log n)$ manner. As with Brew, the experiments were run on a 2.5 GHz Intel Core i7 machine with 16 GB DDR3 memory.

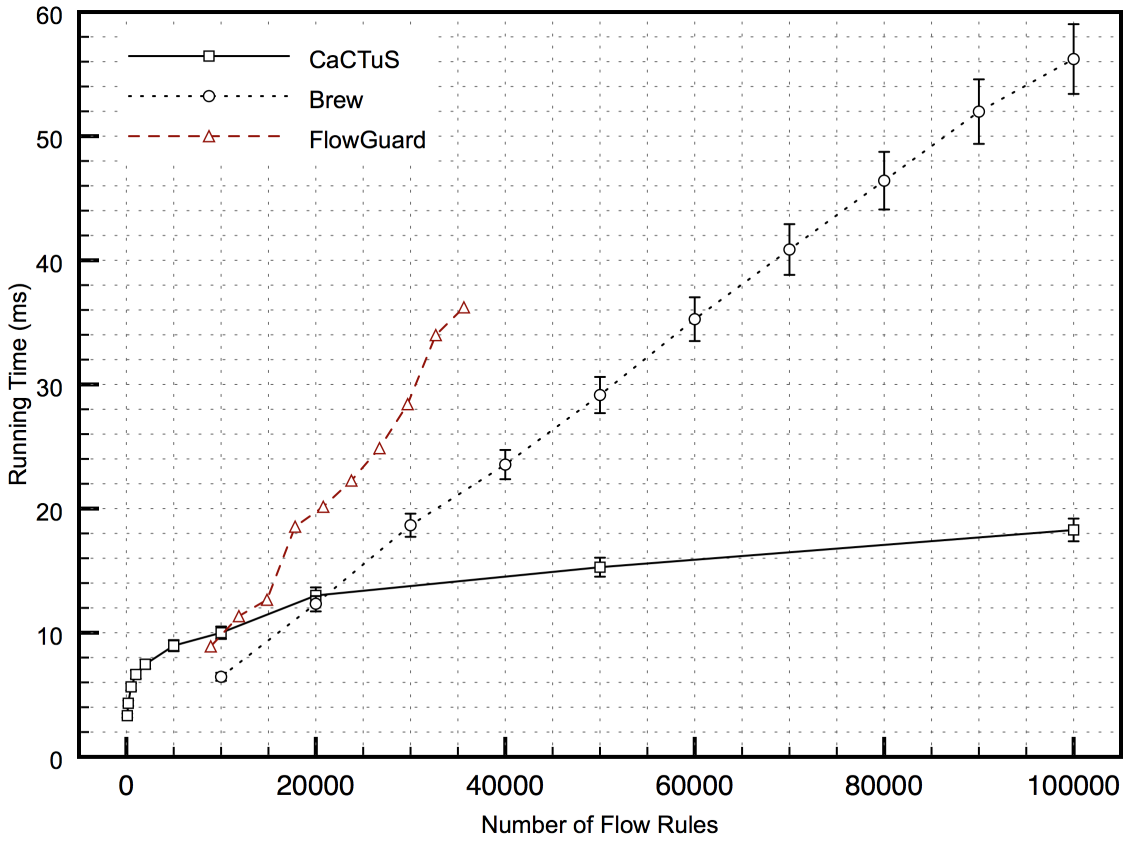

Figure 7.4: Comparison of CaCTuS Running Time with Brew and FlowGuard Versus Size of Flow Rule Tables.

With an input file containing about $10,000$ flow rules, the processing time was about 11 ms. Rules were further replicated and inserted into the system to observe growth of computation time. Ten different test runs were conducted on flow tables of size varying from $10,000$ to $100,000$ rules, and the resulting running times were averaged to get the results in the plot. Error bars indicate the max-min value from the different runs.

The next experiment consisted of evaluating the effect of address block size in the ASMM, to computation time, and total space utilization. Once again, for testing purposes, we utilized IPv4 instead of IPv6, with block sizes of $2, 4$ and 8. We tested the computation time and total space utilization for 100 thousand, 500 thousand and 1 million randomly generated address spaces. The results reaffirmed what has been known about the Buddy Algorithm and showed that the smaller the block size, the higher the total space utilization. However, smaller block size also accounted for higher computational overhead. Thus, selecting the optimum block size is dependent on user preferences.

The final consideration was performance overhead of the CaCTuS module. Using transfer of a large 1GB file between hosts A and D in Figure 6.10 as the parameter, CaCTuS was compared to Brew to measure the overhead to file transfer time. Figure 7.5 shows the time taken to transfer the file in cases where the rules being inserted *a*) natively; *b*) through Brew; and *c*) through CaCTuS. Each test was run 100 times, and the transfer time was averaged to obtain the file transfer time for each case, with the error bars indicating the range of the time taken. CaCTuS added about a 3% increase in transfer time (average of 100 test runs), which is markedly better than Brew.

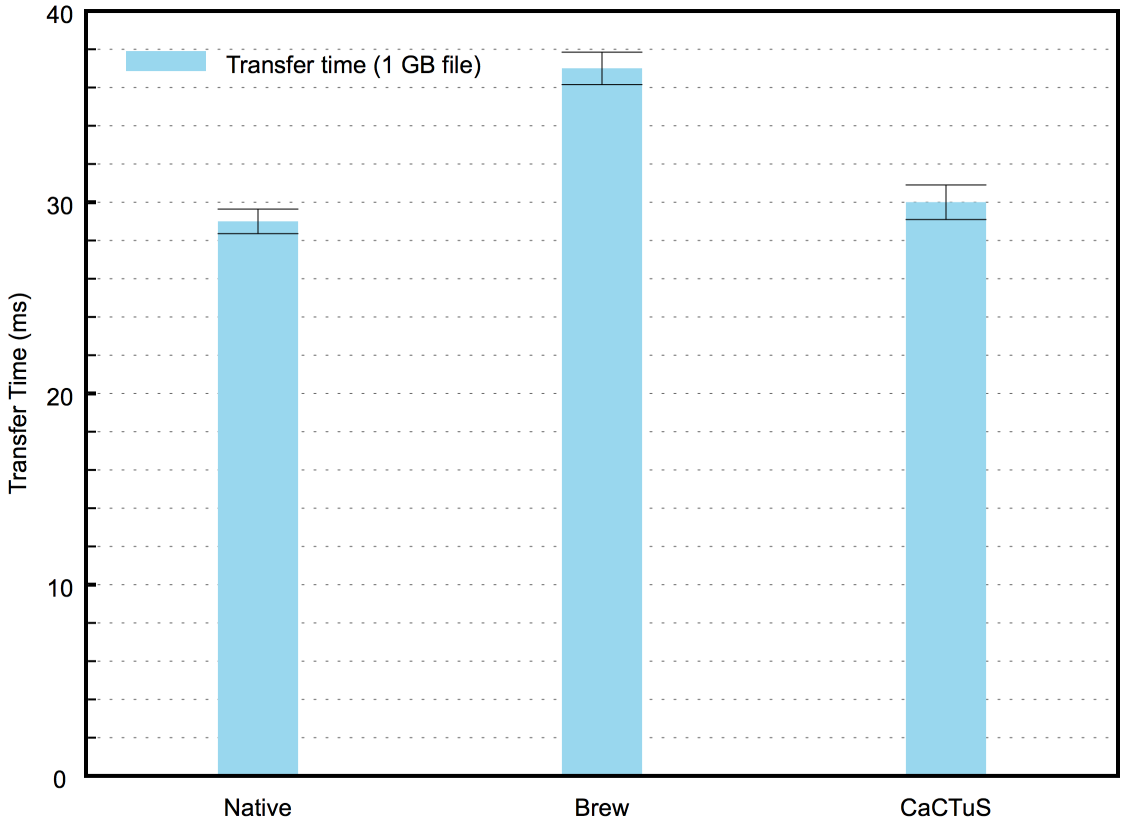

Figure 7.5: Network Performance Overhead for CaCTuS.

Chapter 8

CONCLUSION

Network trends such as the ever-increasing number of mobile users, BYOD environments, increased focus on addressing insider threats and the need to control flow of information between tenants in a hosted data center environment pose a new set of challenges that can be addressed by SDN. Recent advances in SDN creates a unique opportunity to enable complex scientific applications to run on a dynamic and tailored infrastructure that includes compute, storage and network resources. This approach provides the performance advantages of strong infrastructure support with little management and deployment costs. With several threat vectors for SDN environments already identified, and new threats being developed/discovered every day, comprehensive security implementation in SDN environments is an issue that needs to be dealt with actively and urgently [118].

To effectively implement security policies, it is often necessary to use several sets of rules on firewalls and other security devices. The presence of multiple devices, especially in a multi-vendor distributed environment leads to conflicts in configuration, which very likely degrade the network security policy and leave room for error. Identifying and removing these conflicts is a serious and complex problem [121]. Multi-tenant data center environments where there are multiple origination points for higher level security policies exacerbates this problem.

Traditional approaches to addressing security issues in such dynamic and distributed environments tried to implement security on individual components, and did not consider security holistically. In a multi-tenant SDN-based cloud environment, the presence of various such security applications and network nodes interacting with each other makes it extremely difficult to manage policies and track policy conflicts.

In this dissertation, a formalism is presented to describe and classify all potential flow rule conflicts in an SDN-based environment including the introduction of a new class that describes cross-layer flow rule conflicts. A methodology that realizes this formalism is presented, and implemented using a controller-based algorithm in a framework called Brew. Flow rules in a distributed controller environment are extracted, and intra- and inter-table flow rule conflicts are detected utilizing cross-layer conflict checking. Further, security enforcement is augmented by including strategies to resolve conflicts in an SDN-based cloud environment. Automatic and assisted conflict resolution mechanisms are addressed and a novel visualization scheme for conflict representation is presented. The run time complexity for the framework is linear, and hence scalable to large SDN-based clouds. However, realizing the need for flow rule conflict avoidance to overcome the timeliness factor for network based MTD, CaCTuS seeks to produce non-overlapping address spaces thereby ensuring that MTD countermeasures being implemented into a system are demonstrably conflict free.

To summarize, this dissertation describes work done to:

- Extend firewall rule conflict classification in a traditional environment to SDN flow rule conflicts by identifying cross-layer conflicts (which tend to be transient in nature).
- Include techniques for global prioritization of flow rules in a decentralized environment depending on the decentralization strategy.
- Detects flow rule conflicts in a multiple, decentralized controller based SDN-based cloud environments.
- Description of strategies for unassisted resolution of the flow rule conflicts, with analysis of their benefits and deficiencies.

- Present a novel visualization scheme implemented to help the administrators view flow rule conflicts visually.
- Introduce an address space management framework to generate conflict free countermeasures while implementing an address hopping based network MTD solution.

I. System Limitations

As with any prototype, Brew is not without limitations. In its current avatar, it is unsuitable in a highly dynamic environment because the conflict resolution model that considers temporal nature of the mapping between different address layers in a dynamic SDN cloud is abecedarian. Currently, the topology of the environment is not considered while detecting conflicts, which might be valuable information to possess while determining conflicts in a multi-tenant environment with similar internal addressing schemes.

While CaCTuS can ensure generation of conflict free countermeasures, it is currently limited to address hopping based MTD solution. To make it truly robust, its functionality needs to be expanded to guaranteeing conflict free countermeasures for all network based MTD solutions.

II. Future Work

The work completed thus far has enabled us to obtain a solid foundation to having a platform that can ensure a conflict free security policy implementation across a distributed SDN environment. However, prior to project maturation, we seek to address some deficiencies in the work including scalability and usability concerns. We anticipate on improving Brew in the ways described below.

In the short term, making Brew work with SDN controllers other than ODL would back up one of the design requirements that the framework be SDN controller agnostic. This would not only help enhance adoption of the framework, but it would help add controller diversity in an SDN based MTD implementation. Since a dynamic SDN cloud could have several new flow rules generated every second, we plan on studying the effect of using multiple analyzers to share the work load to parallelize processing. However, parallelization would take resources away from the resource constrained controller. Thus, we would need to find the circumstances which would warrant parallelization, and its benefits. To increase robustness and to be more complete, adding the topology as an input into Brew might be able to make the conflict detection more thorough.

Our current implementation uses eight octet-wise Patricia trie lookup to identify address space overlap. We plan on investigating the size of flow tables that would start showing benefits for going this route. Alternately, using hashed Patricia tries, or other data structures which might be more efficient for lookups will also be considered. Alternate ways to potentially increase the efficiency of the conflict detection engine to sub linear running time using more efficient and customized data structures will be researched.

Next, we plan on integrating results from studies that incorporate stateful functionality into the SDN environment. Evolving from a pure packet filter based security application to one that can have rules based on connection state would greatly enhance the effectiveness of security policy that can be implemented. Research from the traditional networking environment suggests this may not be very complex [160].

Verifying that new flow policies adhere to organizational security policies is a twofold problem: *a*) establish that the newly generated flow rules do not conflict with existing flow rules; and *b*) ensure that the conflict-free flow rule table adhere to the

high-level organizational policies. Brew addresses the first of these problems. To ensure compliance with higher level organizational policies, the work in this dissertation needs to be adapted/extended to work in the area of regulatory compliance. Such work usually uses a policy specification language based on a restricted subset of First Order Temporal Logic (FOTL) which can capture the high-level requirements, and encode what adherence to the policies mean [161, 162]. Combining this work with FOTL would greatly increase the future applications of the framework.

Further, we plan on considering flow rule optimization based on rule positioning and examine adaptive prioritization of rules. Including role-based and attribute-based policy conflicts is a natural extension of this work. Future visualization work includes upgrades to provide newer features to assist in scalability. A zoom-in/zoom-out feature aiding in the visualization process and graphs depicting the statistical data gathered from the switches using the OpenFlow protocol could be added. Since a one-size fits all solution rarely works, flavors of the Brew framework tailored for host based SDN firewalls and a mobile (lightweight) solution for tactical clouds would be a potentially interesting extension of the current work. And finally, using machine learning algorithms to identify MTD timer thresholds would help CaCTuS deliver environment-optimized results.

As it stands, Brew is a standalone flow rule conflict detection and resolution framework. However, work is currently underway to integrate Brew into a MTD solution, along with CaCTuS to ensure that countermeasures generated do not cause conflicts in the environment. This would also involve making CaCTuS more robust by guaranteeing conflict free rules over MTD techniques other than address hopping.

REFERENCES

[1] *ITU releases 2014 ICT figures*, 2014. [Online]. Available: https://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2014-e.pdf

[2] "Defense in Depth," National Security Agency, Tech. Rep. [Online]. Available: https://www.nsa.gov/ia/_files/support/defenseindepth.pdf

[3] N. C. Damianou, "A Policy Framework for Management of Distributed Systems," PhD Dissertation, Imperial College, 2002.

[4] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On Scalability of Software-Defined Networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 136–141, 2013.

[5] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC 10)*. ACM, 2010, pp. 267–280.

[6] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and others, "Onix: A Distributed Control Platform for Large-Scale Production Networks." in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, vol. 10. USENIX Association, 2010, pp. 1–6.

[7] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically Centralized?: State Distribution Trade-Offs in Software Defined Networks," in *Proceedings of the 1st Workshop on Hot Topics in Software Defined Networking (HotSDN 2012)*. ACM, 2012, pp. 1–6.

[8] A. Tootoonchian and Y. Ganjali, "HyperFlow: A Distributed Control Plane for OpenFlow," in *Proceedings of the 2010 Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN '10)*. USENIX Association, 2010, pp. 3–3.

[9] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an Elastic Distributed SDN Controller," *ACM SIGCOMM Computer Communication Review*, vol. 43, pp. 7–12, 2013.

[10] E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan, "Conflict Classification and Analysis of Distributed Firewall Policies," *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 10, pp. 2069–2084, 2005.

[11] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A Security Enforcement Kernel for Openflow Networks," in *Proceedings of the 1st Workshop on Hot Topics in Software Defined Networking (HotSDN 2012)*. ACM, 2012, pp. 121–126.

[12] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "FlowGuard: Building Robust Firewalls for Software-Defined Networks," in *Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking (HotSDN 2014)*. ACM, 2014, pp. 97–102.

[13] Y. Bartal, A. Mayer, K. Nissim, and A. Wool, "Firmato: A Novel Firewall Management Toolkit," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy*. IEEE, 1999, pp. 17–31.

[14] S. Pisharody, A. Chowdhary, and D. Huang, "Security Policy Checking in Distributed SDN based Clouds," in *Proceedings of the 2016 IEEE Conference on Communications and Network Security (CNS)*. Philadelphia, USA: IEEE, Oct. 2016.

[15] K. Ingham and S. Forrest, "A History and Survey of Network Firewalls," University New Mexico, Technical Report TRCS-2002-37, 2002.

[16] J. Carlin, "Where Business and Cyberterror Collide: The View from the Justice Department," Oct. 2016.

[17] S. M. Bellovin, "Distributed Firewalls," *;login:*, vol. 24, no. 5, pp. 37–39, 1999.

[18] M. Sloman, J. Magee, K. Twidle, and J. Kramer, "An Architecture for Managing Distributed Systems," in *Proceedings of the 4th Workshop on Future Trends of Distributed Computing Systems*. IEEE, Sep. 1993, pp. 40–46.

[19] B. Moore, "Policy Core Information Model (PCIM) Extensions," IETF, RFC 3460, Jan. 2003. [Online]. Available: https://tools.ietf.org/html/rfc3460

[20] D. C. Verma, "Simplifying Network Administration Using Policy-Based Management," *IEEE Network*, vol. 16, no. 2, pp. 20–26, 2002.

[21] W. Han and C. Lei, "A Survey on Policy Languages in Network and Security Management," *Computer Networks*, vol. 56, no. 1, pp. 477–489, Jan. 2012.

[22] H. Mahon, Y. Bernet, S. Herzog, and J. Schnizlein, "Requirements for a Policy Management System," IETF, Internet Draft, 1999.

[23] S. J. Shepard, "Policy-Based Networks: Hype and Hope," *IT Professional*, vol. 2, no. 1, pp. 12–16, Jan. 2000.

[24] Z. Fu, S. F. Wu, H. Huang, K. Loh, F. Gong, I. Baldine, and C. Xu, "IPSec/VPN Security Policy: Correctness, Conflict Detection, and Resolution," in *Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY 2001)*, ser. Lecture Notes in Computer Science, vol. 1995. Springer, 2001, pp. 39–56.

[25] F. Caldeira and E. Monteiro, "A Policy-Based Approach to Firewall Management," in *Proceedings of the IFIP TC6 / WG6.2 & WG6.7 Conference on Network Control and Engineering for QoS, Security and Mobility (Net-Con '02)*. Springer, 2003, pp. 115–126.

[26] J. D. Moffett and M. S. Sloman, "Policy Hierarchies for Distributed Systems Management," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 9, pp. 1404–1414, 1993.

[27] J. D. Moffett, "Requirements and Policies," in *Proceedings of the Workshop on Policies for Distributed Systems and Networks (POLICY 1999)*. UK: HP-Laboratories Bristol, 1999.

[28] J. Strassner and S. Schleimer, "Policy Framework Definition Language," IETF, Internet Draft, Nov. 1998.

[29] J. Bailey, G. Papamarkos, A. Poulovassilis, and P. T. Wood, "An Event-Condition-Action Language for XML," in *Web Dynamics*. Springer, 2004, pp. 223–248.

[30] S. Das, G. Parulkar, N. McKeown, P. Singh, D. Getachew, and L. Ong, "Packet and Circuit Network Convergence with Openflow," in *Optical Fiber Communication Conference*. Optical Society of America, 2010, p. OTuG1.

[31] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[32] K. Bakshi, "Considerations for Software-Defined Networking (SDN): Approaches and Use Cases," in *Proceedings of the 2013 IEEE Aerospace Conference*. IEEE, 2013, pp. 1–9.

[33] R. Kawashima, "vNFC: A Virtual Networking Function Container for SDN-enabled Virtual Networks," in *Proceedings of the 2nd Symposium on Network Cloud Computing and Applications (NCCA 2012)*. IEEE, 2012, pp. 124–129.

[34] J. Quittek, T. Zseby, B. Claise, and S. Zander, "Requirements for IP Flow Information Export (IPFIX)," IETF, RFC 3917, 2004. [Online]. Available: https://tools.ietf.org/html/rfc3917

[35] D. Kreutz, F. M. Ramos, P. E. Verissimo, 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, 2015.

[36] "Open Network Foundation." [Online]. Available: https://www.opennetworking.org/sdn-resources/openflow

[37] "Open vSwitch." [Online]. Available: http://openvswitch.org/

[38] "OpenStack." [Online]. Available: http://www.openstack.org/

[39] "CloudStack." [Online]. Available: https://cloudstack.apache.org/

[40] E. Lupu and M. Sloman, "Conflict Analysis for Management Policies," in *Integrated Network Management V*, ser. IFIP - The International Federation for Information Processing. Springer, 1997, pp. 430–443.

[41] E. C. Lupu and M. Sloman, "Conflicts in Policy-Based Distributed Systems Management," *IEEE Transactions on Software Engineering*, vol. 25, no. 6, pp. 852–869, 1999.

[42] D. Eppstein and S. Muthukrishnan, "Internet Packet Filter Management and Rectangle Geometry," in *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '01)*. Society for Industrial and Applied Mathematics, 2001, pp. 827–835.

[43] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

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

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

[46] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra, "Fireman: A Toolkit for Firewall Modeling and Analysis," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*. IEEE, 2006, pp. 15–pp.

[47] H. Hu, G.-J. Ahn, and K. Kulkarni, "Fame: A Firewall Anomaly Management Environment," in *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration (SafeConfig '10)*. ACM, 2010, pp. 17–26.

[48] V. Capretta, B. Stepien, A. Felty, and S. Matwin, "Formal Correctness of Conflict Detection for Firewalls," in *Proceedings of the 2007 ACM Workshop on Formal Methods in Security Engineering (FMSE '07)*. ACM, 2007, pp. 22–30.

[49] L. Kagal, "Rei: A Policy Language for the Me-Centric Project," HP Laboratories, Palo Alto, Technical Report, 2002.

[50] G. H. v. Wright, "Deontic Logic," *Mind*, vol. 60, no. 237, pp. 1–15, 1951.

[51] B. Zhang, E. Al-Shaer, R. Jagadeesan, J. Riely, and C. Pitcher, "Specifications of a High-Level Conflict-Free Firewall Policy Language for Multi-Domain Networks," in *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT '07)*. ACM, 2007, pp. 185–194.

[52] S. Pozo, R. Ceballos, and R. M. Gasca, "AFPL, an Abstract Language Model for Firewall ACLs," in *Proceedings of the 8th International Conference on Computational Science and Its Applications (ICCSA 2008)*. Springer, 2008, pp. 468–483.

[53] A. Mayer, A. Wool, and E. Ziskind, "Fang: A Firewall Analysis Engine," in *Proceedings of the 2000 IEEE Symposium on Security and Privacy.* IEEE, 2000, pp. 177–187.

[54] T. Tran, E. S. Al-Shaer, and R. Boutaba, "PolicyVis: Firewall Security Policy Visualization and Inspection," in *LISA*, vol. 7, 2007, pp. 1–16.

[55] F. Mansmann, T. Gbel, and W. Cheswick, "Visual Analysis of Complex Firewall Configurations," in *Proceedings of the 9th International Symposium on Visualization for Cyber Security.* ACM, 2012, pp. 1–8.

[56] "Floodlight." [Online]. Available: http://www.projectfloodlight.org/floodlight/

[57] T. Javid, T. Riaz, and A. Rasheed, "A Layer2 Firewall for Software Defined Network," in *Proceedings of the 2014 Conference on Information Assurance and Cyber Security (CIACS).* IEEE, 2014, pp. 39–42.

[58] M. Suh, S. H. Park, B. Lee, and S. Yang, "Building Firewall Over the Software-Defined Network Controller," in *Proceedings of the 16th International Conference on Advanced Communication Technology (ICACT2014).* IEEE, 2014, pp. 744–748.

[59] A. Shieha, "Application Layer Firewall Using OpenFlow," Master's Thesis, University of Aleppo, 2014.

[60] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Avant-Guard: Scalable and Vigilant Switch Flow Management in Software-Defined Networks," in *Proceedings of the 20th ACM Conference on Computer and Communications Security.* ACM, 2013, pp. 413–424.

[61] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: Detecting Security Attacks in Software-Defined Networks." in *Proceedings of the Network and Distributed System Security Symposium 2015 (NDSS 15).* ISOC, 2015.

[62] L. Cholvy and F. Cuppens, "Analyzing Consistency of Security Policies," in *Proceedings of the 1997 IEEE Symposium on Security and Privacy.* IEEE, 1997, pp. 103–112.

[63] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder Policy Specification Language," in *Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY 2001)*, ser. Lecture Notes in Computer Science, vol. 1995. Springer, 2001, pp. 18–38.

[64] E. Al-Shaer and S. Al-Haj, "Flowchecker: Configuration Analysis and Verification of Federated Openflow Infrastructures," in *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration (SafeConfig '10).* ACM, 2010, pp. 37–44.

[65] S. Hazelhurst, "Algorithms for Analysing Firewall and Router Access Lists," *CoRR*, vol. cs.NI/0008006, 2000. [Online]. Available: http://arxiv.org/abs/cs.NI/0008006

[66] 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 (IMC 05)*. USENIX Association, 2005, p. 32.

[67] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker, and others, "Composing Software-Defined Networks," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*. USENIX Association, 2013, pp. 1–13.

[68] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A Network Programming Language," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*, vol. 46. ACM, 2011, pp. 279–291.

[69] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions Using Flowtags," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*. USENIX Association, 2014.

[70] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, "FRESCO: Modular Composable Security Services for Software-Defined Networks." in *Proceedings of the Network and Distributed System Security Symposium 2013 (NDSS 13)*. ISOC, Feb. 2013.

[71] S. Natarajan, X. Huang, and T. Wolf, "Efficient Conflict Detection in Flow-Based Virtualized Networks," in *Proceedings of the 2012 International Conference on Computing, Networking and Communications (ICNC 2012)*. IEEE, 2012, pp. 690–696.

[72] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying Network-Wide Invariants in Real Time," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*. USENIX Association, 2013, pp. 15–27.

[73] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real Time Network Policy Checking Using Header Space Analysis," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*. USENIX Association, 2013, pp. 99–111.

[74] P. Kazemian, G. Varghese, and N. McKeown, "Header Space Analysis: Static Checking for Networks," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*. USENIX Association, 2012, pp. 113–126.

[75] S. H. Yeganeh and Y. Ganjali, "Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications," in *The Beacon Openflow Controller*. ACM, 2012, pp. 19–24.

[76] K. Phemius, M. Bouet, and J. Leguay, "DISCO: Distributed SDN Controllers in a Multi-Domain Environment," in *Proceedings of the 2014 IEEE Network Operations and Management Symposium (NOMS 2014)*. IEEE, May 2014, pp. 1–2.

[77] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and others, "ONOS: Towards an Open, Distributed SDN OS," in *Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking (HotSDN 2014)*. ACM, 2014, pp. 1–6.

[78] M. F. Bari, A. R. Roy, S. R. Chowdhury, Q. Zhang, M. F. Zhani, R. Ahmed, and R. Boutaba, "Dynamic Controller Provisioning in Software-Defined Networks," in *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*. IEEE, 2013, pp. 18–25.

[79] G. Yao, J. Bi, Y. Li, and L. Guo, "On the Capacitated Controller Placement Problem in Software-Defined Networks," *IEEE Communications Letters*, vol. 18, no. 8, pp. 1339–1342, Aug. 2014.

[80] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The Internet Topology Zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, Oct. 2011.

[81] P. Xiao, W. Qu, H. Qi, Z. Li, and Y. Xu, "The SDN Controller Placement Problem for WAN," in *Proceedings of the 2014 IEEE/CIC International Conference on Communications in China (ICCC)*. IEEE, 2014, pp. 220–224.

[82] Y. Hu, W. Wang, X. Gong, X. Que, and S. Cheng, "On the Placement of Controllers in Software-Defined Networks," *The Journal of China Universities of Posts and Telecommunications*, vol. 19, pp. 92–171, 2012.

[83] ——, "Reliability-aware controller placement for Software-Defined Networks," in *Proceedings of the 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. IEEE, May 2013, pp. 672–675.

[84] V. V. Vazirani, *Approximation Algorithms*. Springer Science & Business Media, 2013.

[85] N. Beheshti and Y. Zhang, "Fast Failover for Control Traffic in Software-Defined Networks," in *Proceedings of the 2012 IEEE Global Communications Conference (GLOBECOM 2012)*. IEEE, Dec. 2012, pp. 2665–2670.

[86] "OpenFlow Switch Specification v1.3.1," Open Networking Foundation, Tech. Rep., Sep. 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf

[87] J. Liu, Y. Li, H. Wang, D. Jin, L. Su, L. Zeng, and T. Vasilakos, "Leveraging Software-Defined Networking for Security Policy Enforcement," *Information Sciences*, vol. 327, pp. 288–299, 2016.

[88] D. A. Joseph, A. Tavakoli, and I. Stoica, "A Policy-Aware Switching Layer for Data Centers," in *Proceedings of the 2008 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '08)*. ACM, 2008.

[89] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi, "The Middlebox Manifesto: Enabling Innovation in Middlebox Deployment," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks (HotNets-X)*. ACM, 2011, p. 21.

[90] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

[91] J. Lee, J. Tourrilhes, P. Sharma, and S. Banerjee, "No More Middlebox: Integrate Processing into Network," *ACM SIGCOMM Computer Communication Review*, vol. 40, pp. 459–460, 2010.

[92] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella, "Toward Software-Defined Middlebox Networking," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets-XI)*. ACM, 2012, pp. 7–12.

[93] I. Alsmadi and D. Xu, "Security of Software Defined Networks: A Survey," *Computers & Security*, vol. 53, pp. 79–108, 2015.

[94] J. G. V. Pena and W. E. Yu, "Development of a Distributed Firewall Using Software Defined Networking Technology," in *Proceedings of the 4th International Conference on Information Science and Technology (ICIST 2014)*. IEEE, 2014, pp. 449–452.

[95] 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 2013 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '13)*. ACM, 2013, pp. 27–38.

[96] X. Liu, H. Xue, X. Feng, and Y. Dai, "Design of the Multi-Level Security Network Switch System Which Restricts Covert Channel," in *Proceedings of the IEEE 3rd International Conference on Communication Software and Networks (ICCSN 2011)*. IEEE, 2011, pp. 233–237.

[97] J. Franois, L. Dolberg, O. Festor, and T. Engel, "Network Security Through Software Defined Networking: A Survey," in *Proceedings of the 7th Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm 2014)*. ACM, 2014, p. 6.

[98] P. A. Porras, S. Cheung, M. W. Fong, K. Skinner, and V. Yegneswaran, "Securing the Software Defined Network Control Layer," in *Proceedings of the Network and Distributed System Security Symposium 2015 (NDSS 15)*. ISOC, 2015.

[99] M. Kuniar, P. Pereni, and D. Kosti, "What You Need to Know About SDN Flow Tables," in *International Conference on Passive and Active Network Measurement*. Springer, 2015, pp. 347–359.

[100] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka, "Verifying Information Flow Goals in Security-Enhanced Linux," *Journal of Computer Security*, vol. 13, no. 1, pp. 115–134, 2005.

[101] I. Ahmad, S. Namal, M. Ylianttila, and A. Gurtov, "Security in Software Defined Networks: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2317–2346, 2015.

[102] A. Chowdhary, S. Pisharody, and D. Huang, "SDN Based Scalable MTD Solution in Cloud Network," in *Proceedings of the 3rd ACM Workshop on Moving Target Defense (MTD 2016)*, ser. MTD '16. New York, NY, USA: ACM, 2016, pp. 27–36. [Online]. Available: http://doi.acm.org/10.1145/2995272.2995274

[103] J. Yackoski, H. Bullen, X. Yu, and J. Li, "Applying Self-Shielding Dynamics to the Network Architecture," in *Moving Target Defense II*. Springer, 2013, pp. 97–115.

[104] C. Basile, A. Cappadonia, and A. Lioy, "Algebraic Models to Detect and Solve Policy Conflicts," in *Proceedings of the 7th International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security (MMM-ACNS 2007)*. Springer, 2007, pp. 242–247.

[105] D. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and X. Xiao, "Overview and Principles of Internet Traffic Engineering," IETF, RFC 3272, 2002. [Online]. Available: https://tools.ietf.org/html/rfc3272

[106] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "A Roadmap for Traffic Engineering in SDN-Openflow Networks," *Computer Networks*, vol. 71, pp. 1–30, Oct. 2014.

[107] Y. Hu, W. Wang, X. Gong, X. Que, and S. Cheng, "Balanceflow: Controller Load Balancing for Openflow Networks," in *Proceedings of the 2012 IEEE 2nd International Conference on Cloud Computing and Intelligent Systems (CCIS 2012)*, vol. 2. IEEE, 2012, pp. 780–785.

[108] D. Erickson, "The Beacon Openflow Controller," in *Proceedings of the 2nd Workshop on Hot Topics in Software Defined Networking (HotSDN 2013)*. ACM, 2013, pp. 13–18.

[109] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI '10)*, vol. 10. USENIX Association, 2010, pp. 19–19.

[110] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-Overhead Datacenter Traffic Management Using End-Host-Based Elephant Detection," in *Proceedings of the 30th International IEEE Conference on Computer Communications (INFOCOM 2011)*. IEEE, 2011, pp. 1629–1637.

[111] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine Grained Traffic Engineering for Data Centers," in *Proceedings of the Seventh Conference on Emerging Networking Experiments and Technologies (CoNEXT '11)*. ACM, 2011, p. 8.

[112] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-Performance Networks," *ACM SIGCOMM Computer Communication Review*, vol. 41, pp. 254–265, 2011.

[113] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable Flow-Based Networking with DIFANE," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 351–362, 2010.

[114] A. Chowdhary, S. Pisharody, A. Alshamrani, and D. Huang, "Dynamic Game Based Security Framework in SDN-enabled Cloud Networking Environments," in *Proceedings of the 2017 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV Security '17)*. ACM, 2017, pp. 53–58. [Online]. Available: http://doi.acm.org/10.1145/3040992.3040998

[115] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, "Applying NOX to the Datacenter," in *Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*. ACM, 2009.

[116] "Wire Speed to PPS." [Online]. Available: https://kb.juniper.net/InfoCenter/index?page=content&id=KB14737

[117] Y. Jimenez, C. Cervello-Pastor, and A. J. Garcia, "On the Controller Placement for Designing a Distributed SDN Control Layer," in *Proceedings of the 2014 IFIP Networking Conference (Networking 2014)*. IEEE, 2014, pp. 1–9.

[118] S. Sezer, S. Scott-Hayward, P.-K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are We Ready for SDN? Implementation Challenges for Software-Defined Networks," *IEEE Communications Magazine*, vol. 51, no. 7, pp. 36–43, 2013.

[119] "DragonFlow." [Online]. Available: https://wiki.openstack.org/wiki/Dragonflow

[120] P. K. Khatkar, "Firewall Rule Set Analysis and Visualization," Master's Thesis, Arizona State University, 2014.

[121] J. G. Alfaro, N. Boulahia-Cuppens, and F. Cuppens, "Complete Analysis of Configuration Rules to Guarantee Reliable Network Security Policies," *International Journal of Information Security*, vol. 7, no. 2, pp. 103–122, Apr. 2008.

[122] F. B. Schneider, "Least Privilege and More," *IEEE Security & Privacy*, vol. 1, no. 5, pp. 55–59, 2003.

[123] H. Joh and Y. K. Malaiya, "Defining and Assessing Quantitative Security Risk Measures Using Vulnerability Lifecycle and CVSS Metrics," in *Proceedings of the 10th International Conference on Security and Management (SAM '11)*, 2011, pp. 10–16.

[124] R. P. Lippmann, J. Riordan, T. Yu, and K. Watson, "Continuous Security Metrics for Prevalent Network Threats: Introduction and First Four Metrics," Massachusetts Institute of Institute of Technology Lincoln Laboratory, Tech. Rep. MIT-LL-IA-3, May 2012.

[125] D. Raymond, G. Conti, T. Cross, and M. Nowatkowski, "Key Terrain in Cyberspace: Seeking the High Ground," in *Proceedings of the 6th International Conference on Cyber Conflict (CyCon 2014)*. IEEE, 2014, pp. 287–300.

[126] J. Natarajan, "Analysis and Visualization of OpenFlow Rule Conflicts," Master's Thesis, Arizona State University, 2016.

[127] M. M. Coulombe, H. Singh, E. Karlson, and M. Venugopal, "OpenDaylight dlux," Sep. 2013. [Online]. Available: https://wiki.opendaylight.org/view/OpenDaylight_dlux:Main

[128] D. R. Morrison, "PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric," *Journal of the ACM (JACM)*, vol. 15, no. 4, pp. 514–534, 1968.

[129] P. Gupta and N. McKeown, "Algorithms for Packet Classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32, 2001.

[130] K. Poornaselvan, S. Suresh, D. Chidambaram, and C. Gayathri, "Efficient IP Lookup Algorithm," in *Special Topics in Computing and ICT Research - Strengthening the Role of ICT in Development*, 2007, vol. 3, pp. 111–122.

[131] "OpenDaylight," 2010. [Online]. Available: https://www.opendaylight.org/

[132] "The Linux Foundation." [Online]. Available: https://www.linuxfoundation.org/

[133] "OpenDaylight Project Repository," May 2014. [Online]. Available: https://github.com/opendaylight/l2switch

[134] B. Pfaff and B. Davie, "The Open vSwitch Database Management Protocol," IETF, RFC 7047, 2013. [Online]. Available: https://tools.ietf.org/html/rfc7047

[135] "Apache Karaf." [Online]. Available: http://karaf.apache.org/

[136] A. Bierman, M. Bjorklund, and K. Watsen, "RESTCONF Protocol," IETF, RFC 8040, Jan. 2017. [Online]. Available: https://tools.ietf.org/html/rfc8040

[137] "Postman," 2013. [Online]. Available: https://www.getpostman.com/

[138] M. Bostock, "D3," 2016. [Online]. Available: https://d3js.org/

[139] "JSON." [Online]. Available: http://www.json.org/

[140] D. Holten, "Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 741–748, 2006.

[141] E. M. Reingold and J. S. Tilford, "Tidier Drawings of Trees," *IEEE Transactions on Software Engineering*, no. 2, pp. 223–228, 1981.

[142] "Mininet." [Online]. Available: http://mininet.org

[143] "Cisco Data Center Infrastructure 2.5 Design Guide," in *Cisco Validated Design I.* Cisco Systems, Inc, 2007.

[144] "Trustworthy Cyberspace: Strategic Plan for the Federal Cybersecurity Research and Development Program," White House, Tech. Rep., 2011.

[145] B. Schmerl, J. Cmara, G. A. Moreno, D. Garlan, and A. Mellinger, "Architecture-Based Self-Adaptation for Moving Target Defense," Technical Report CMU-ISR-14-109. Carnegie Mellon University, Tech. Rep., 2014.

[146] D. Evans, A. Nguyen-Tuong, and J. Knight, "Effectiveness of Moving Target Defenses," in *Moving Target Defense.* Springer, 2011, pp. 29–48.

[147] R. Saha and A. Agarwal, "SDN Approach to Large Scale Global Data Centers," *Proceedings of the Open Networking Summit, Santa Clara, California, USA*, 2012.

[148] C.-J. Chung, T. Xing, D. Huang, D. Medhi, and K. Trivedi, "SeReNe: On Establishing Secure and Resilient Networking Services for an SDN-Based Multi-Tenant Datacenter Environment," in *Proceedings of the 2015 IEEE International Conference on Dependable Systems and Networks Workshops (DSN-W).* IEEE, 2015, pp. 4–11.

[149] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "Openflow Random Host Mutation: Transparent Moving Target Defense Using Software Defined Networking," in *Proceedings of the 1st Workshop on Hot Topics in Software Defined Networking (HotSDN 2012).* ACM, 2012, pp. 127–132.

[150] C.-J. Chung, P. Khatkar, T. Xing, J. Lee, and D. Huang, "NICE: Network Intrusion Detection and Countermeasure Selection in Virtual Network Systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 10, no. 4, pp. 198–211, 2013.

[151] T. Hobson, H. Okhravi, D. Bigelow, R. Rudd, and W. Streilein, "On the Challenges of Effective Movement," in *Proceedings of the 1st ACM Workshop on Moving Target Defense (MTD 2014).* ACM, 2014, pp. 41–50.

[152] J. Xu, P. Guo, M. Zhao, R. F. Erbacher, M. Zhu, and P. Liu, "Comparing Different Moving Target Defense Techniques," in *Proceedings of the 1st ACM Workshop on Moving Target Defense (MTD 2014)*. ACM, 2014, pp. 97–107.

[153] D. Kewley, R. Fink, J. Lowry, and M. Dean, "Dynamic Approaches to Thwart Adversary Intelligence Gathering," in *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings*, vol. 1. IEEE, 2001, pp. 176–185.

[154] M. Atighetchi, P. Pal, F. Webber, and C. Jones, "Adaptive Use of Network-Centric Mechanisms in Cyber-Defense," in *Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on*. IEEE, 2003, pp. 183–192.

[155] M. Krzywinski, "Port Knocking from the Inside Out," *SysAdmin Magazine*, vol. 12, no. 6, pp. 12–17, 2003.

[156] S. Antonatos, P. Akritidis, E. P. Markatos, and K. G. Anagnostakis, "Defending Against Hitlist Worms Using Network Address Space Randomization," *Computer Networks*, vol. 51, no. 12, pp. 3471–3490, Aug. 2007.

[157] E. Al-Shaer, "Toward Network Configuration Randomization for Moving Target Defense," in *Moving Target Defense*. Springer, 2011, pp. 153–159.

[158] Q. Duan, E. Al-Shaer, and H. Jafarian, "Efficient Random Route Mutation Considering Flow and Network Constraints," in *Proceedings of the 2013 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2013, pp. 260–268.

[159] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review," in *Memory Management*. Springer, 1995, pp. 1–116.

[160] L. Buttyn, G. Pk, and T. V. Thong, "Consistency Verification of Stateful Firewalls Is Not Harder Than the Stateless Case," *Infocommunications Journal*, vol. 64, no. 1, pp. 2–8, 2009.

[161] O. Chowdhury, A. Gampe, J. Niu, J. von Ronne, J. Bennatt, A. Datta, L. Jia, and W. H. Winsborough, "Privacy Promises That Can Be Kept: A Policy Analysis Method with Application to the HIPAA Privacy Rule," in *Proceedings of the 18th ACM Symposium on Access Control Models and Technologies (SACMAT '13)*. ACM, 2013, pp. 3–14.

[162] Y. Shvartzshnaider, S. Tong, T. Wies, P. Kift, H. Nissenbaum, L. Subramanian, and P. Mittal, "Crowdsourced, Actionable and Verifiable Contextual Informational Norms," *CoRR*, vol. abs/1601.04740.

APPENDIX A

FLOW RULE MATCH FIELDS

The following lists all the default match fields for a rule in OpenFlow v1.3.1 [86].

Argument	Description	Required Field?
OFPXMT_OFB_IN_PORT	Switch input port	✓
OFPXMT_OFB_IN_PHY_PORT	Switch physical input port	
OFPXMT_OFB_METADATA	Metadata passed between tables	
OFPXMT_OFB_ETH_DST	Ethernet destination address	✓
OFPXMT_OFB_ETH_SRC	Ethernet source address	✓
OFPXMT_OFB_ETH_TYPE	Ethernet frame type	✓
OFPXMT_OFB_VLAN_VID	VLAN ID	
OFPXMT_OFB_VLAN_PCP	VLAN priority	
OFPXMT_OFB_IP_DSCP	IP DSCP (6 bits in ToS field)	
OFPXMT_OFB_IP_ECN	IP ECN (2 bits in ToS field)	
OFPXMT_OFB_IP_PROTO	IP protocol	✓
OFPXMT_OFB_IPV4_SRC	IPv4 source address	✓
OFPXMT_OFB_IPV4_DST	IPv4 destination address	✓
OFPXMT_OFB_TCP_SRC	TCP source port	✓
OFPXMT_OFB_TCP_DST	TCP destination port	✓
OFPXMT_OFB_UDP_SRC	UDP source port	✓
OFPXMT_OFB_UDP_DST	UDP destination port	✓
OFPXMT_OFB_SCTP_SRC	SCTP source port	
OFPXMT_OFB_SCTP_DST	SCTP destination port	
OFPXMT_OFB_ICMPV4_TYPE	ICMP type	
OFPXMT_OFB_ICMPV4_CODE	ICMP code	
OFPXMT_OFB_ARP_OP	ARP opcode	
OFPXMT_OFB_ARP_SPA	ARP source IPv4 address	
OFPXMT_OFB_ARP_TPA	ARP target IPv4 address	
OFPXMT_OFB_ARP_SHA	ARP source hardware address	
OFPXMT_OFB_ARP_THA	ARP target hardware address	
OFPXMT_OFB_IPV6_SRC	IPv6 source address	✓
OFPXMT_OFB_IPV6_DST	IPv6 destination address	✓
OFPXMT_OFB_IPV6_FLABEL	IPv6 Flow Label	
OFPXMT_OFB_ICMPV6_TYPE	ICMPv6 type	
OFPXMT_OFB_ICMPV6_CODE	ICMPv6 code	
OFPXMT_OFB_IPV6_ND_TARGET	Target address for ND	
OFPXMT_OFB_IPV6_ND_SLL	Source link-layer for ND	
OFPXMT_OFB_IPV6_ND_TLL	Target link-layer for ND	
OFPXMT_OFB_MPLS_LABEL	MPLS label	
OFPXMT_OFB_MPLS_TC	MPLS TC	
OFPXMT_OFP_MPLS_BOS	MPLS BoS bit	
OFPXMT_OFB_PBB_ISID	PBB I-SID	
OFPXMT_OFB_TUNNEL_ID	Logical Port Metadata	
OFPXMT_OFB_IPV6_EXTHDR	IPv6 Extension Header	

Table A.1: Flow Table Match Fields.

APPENDIX B

FLOW RULE MATCH FIELDS

The following is a list of actions that may be associated with flow entries in OpenFlow v1.3.1 [86].

Argument	Description
OFPAT_OUTPUT	Output to switch port
OFPAT_COPY_TTL_OUT	Copy TTL "outwards"from next-to-outermost to outermost
OFPAT_COPY_TTL_IN	Copy TTL "inwards"from outermost to next-to-outermost
OFPAT_SET_MPLS_TTL	MPLS TTL
OFPAT_DEC_MPLS_TTL	Decrement MPLS TTL
OFPAT_PUSH_VLAN	Push a new VLAN tag
OFPAT_POP_VLAN	Pop the outer VLAN tag
OFPAT_PUSH_MPLS	Push a new MPLS tag
OFPAT_POP_MPLS	Pop the outer MPLS tag
OFPAT_SET_QUEUE	Set queue id when outputting to a port
OFPAT_GROUP	Apply group
OFPAT_SET_NW_TTL	IP TTL
OFPAT_DEC_NW_TTL	Decrement IP TTL
OFPAT_SET_FIELD	Set a header field using OXM TLV format
OFPAT_PUSH_PBB	Push a new PBB service tag (I-TAG)
OFPAT_POP_PBB	Pop the outer PBB service tag (I-TAG)
OFPAT_EXPERIMENTER	Experimenter defined

Table B.1: Flow Table Actions.

APPENDIX C

LIST OF ABBREVIATIONS

The following is a complete list of various abbreviations and acronyms used throughout this dissertation listed alphabetically.

AAA	Authentication, Authorization and Accounting
API	Application Programming Interface
ARP	Address Resolution Protocol
BGP	Border Gateway Protocol
BDD	Binary Decision Diagrams
BYOD	Bring Your Own Device
CAPWAP	Control And Provisioning of Wireless Access Points
CKT	Cyber Key Terrain
CLI	Command Line Interface
CRUD	Create, Read, Update and Delete
CSV	Comma Separated Values
DDoS	Distributed Denial-of-Service
DHCP	Dynamic Host Configuration Protocol
DPI	Deep Packet Inspection
DTO	Data Transfer Object
ECA	Event-Condition-Action
FPA	Firewall Policy Advisor
GUI	Graphical User Interface
IaaS	Infrastructure-as-a-Service
IDS	Intrusion Detection System
IETF	Internet Engineering Task Force
IPFIX	IP Flow Information eXport
IPS	Intrusion Prevention System
MDL	Model Definition Language
MPLS	Multi Protocol Label Switching
MTD	Moving Target Defense
NAT	Network Address Translation
NIC	Network Interface Card
ODL	OpenDaylight
ONF	Open Network Foundation
OSGi	Open Service Gateway Initiative
OSI	Open Systems Interconnection
OVS	Open Virtual Switch
PBB	Provider Backbone Bridge
PCEP	Path Computation Element Protocol
PCIM	Policy Core Information Model
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PM	Policy Manager
PR	Policy Repository
QoS	Quality of Service
REST	Representational State Transfer
SaaS	Security-as-a-Service
SDN	Software-Defined Networks
SEK	Security Enforcement Kernel
SLA	Service Level Agreement
SNMP	Simple Network Management Protocol
TCP	Transmission Control Protocol
TE	Traffic Engineering
TTL	Time to Live
UI	User Interface
VM	Virtual Machine
VLAN	Virtual Local-Area Network
VPN	Virtual Private Network