Policy-driven Network Defense for Software Defined Networks

by

Wonkyu Han

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

Approved November 2016 by the
Graduate Supervisory Committee:

Gail-Joon Ahn, Co-Chair
Ziming Zhao, Co-Chair
Adam Doupé
Dijiang Huang
Yanchao Zhang

ARIZONA STATE UNIVERSITY

December 2016

ABSTRACT

Software-Defined Networking (SDN) is an emerging network paradigm that decouples the control plane from the data plane, which allows network administrators to consolidate common network services into a centralized module named *SDN controller*. Applications' policies are transformed into standardized network rules in the data plane via SDN controller. Even though this centralization brings a great flexibility and programmability to the network, network rules generated by SDN applications cannot be trusted because there may exist malicious SDN applications, and insecure network flows can be made due to complex relations across network rules. In this dissertation, I investigate how to identify and resolve these security violations in SDN caused by the combination of network rules and applications' policies. To this end, I propose a systematic policy management framework that better protects SDN itself and hardens existing network defense mechanisms using SDN.

More specifically, I discuss the following four security challenges in this dissertation: (1) In SDN, *generating reliable network rules* is challenging because SDN applications cannot be trusted and have complicated dependencies each other. To address this problem, I analyze applications' policies and remove those dependencies by applying grid-based policy decomposition mechanism; (2) One network rule could accidentally affect others (or by malicious users), which lead to creating of *indirect security violations*. I build systematic and automated tools that analyze network rules in the data plane to detect a wide range of security violations and resolve them in an automated fashion; (3) A fundamental limitation of current SDN protocol (OpenFlow) is a *lack of statefulness*, which is extremely important to several security applications such as stateful firewall. To bring statelessness to SDN-based environment, I come up with an innovative stateful monitoring scheme by extending existing OpenFlow specifications; (4) Existing honeynet architecture is suffering from its limited functionalities of ’data control’ and ’data capture’. To address this challenge, I design and implement an innovative next generation SDN-based honeynet architecture.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my wife Hyounyoung Kim for her unconditional supports, understanding, and love. When I was struggling for my research, she stayed on my side and has encouraged and supported me a lot. Without her, I could never achieve this degree program. My little princess, Shawn Han also gave me peace of mind. Whenever I see her beautiful and bright smile, I feel my fatigue disappears.

I want to express my deepest gratitude to my advisor, Prof. Gail-Joon Ahn. The main reason that I have chosen ASU is solely because of him. He always inspires my research interest and has supported me academically, emotionally and financially throughout my entire program. His encouragement and handful advices made me to take an one step further so that I could achieve PhD degree. I also believe that the patience and leadership that I learned from him would let me overcome any difficulties or obstacles in the future.

I also want to thank to my supervisory committee members, Prof. Ziming Zhao, Prof. Adam Doupé, Prof. Dijiang Huang, and Prof. Yanchao Zhang. Their invaluable comments and guidance greatly helped me prepare my dissertation. Through intensive discussion with them during the comprehensive exam and prospectus defense, I was able to capture missing components and improve the quality of my dissertation. Especially, Prof. Ziming Zhao and Prof. Adam Doupé were very helpful and supportive throughput my entire program by being great mentors and friends. I really appreciate their help and assistance, and I am hoping that we could remain as a good collaborator.

I am also grateful to my friend, Carlos E. Rubio-Medrano. Tea times we used to have in every morning often broadened my insights about academic world and provided me an energy to keep staying in my research. I truly wish you many blessings on your way. I would like to thank to my collaborator and colleagues as well including Prof. Hongxin Hu (Clemson University), Mike Mabey, Sukwha Kyung, Lakshmi Srinivas, Naveen Tiwari. Thanks for their assistance, supports, and hard work.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

1.1 Introduction

A major shift in network infrastructure is in progress from the hardware-based *ossified* network to the software-based *programmable* network. The compelling example of this is the advent of software-defined networks (SDNs) [47, 46, 60]. Traditional network relies on a variety of hardware devices that can be installed at a specific location in a network to provide networking functions such as firewall, IDS/IPS, load-balancer, and proxy. However, the mixture of control logic and its data processing modules inside the box makes the network more complex and hard to manage. To solve this challenge, SDN decouples the control plane from the data plane and consolidates the control logic of devices into a dedicated *SDN controller*. In this way, SDN helps network administrators easily program underlying network via specific control channels (e.g., OpenFlow [86]).

Several benefits that SDN brings are summarized as below:

- *Centralized network environment:* Compared to the traditional network environment, SDN basically decouples the control plane of a hardware from its data processing modules [86]. The separation of the control plane from the data plane allows network administrators to consolidate common network services into *SDN controller* and help them centrally *program* the entire network.

- *High-speed networking:* SDNs implement high speed traffic forwarding by applying simple ”match-action” rules in the data plane. According to a recent report from Google [70], SDN helped them achieve the utilization of WAN at close to 100% utilization whereas other state-of-the-art network techniques only showed about 30% to 40% network utilization. In addition, recent researches have shown that SDN controllers can handle 1.6 million requests per second with the response time of 2ms on average [118] and SDN switches has achieved high throughput: 10.1 million packets (64B) per second with 100K flow rules [92].

- *Flexible and programmable functions:* To lower the barrier to network innovation, SDN allows network administrators to easily program the data plane via specific control channels such as OpenFlow [31]. An OpenFlow switch supports multiple flow tables to process incoming packets and enforce multiple actions using *set-field* actions. In this way, SDN transforms the switch into multi functional device that acts as a route, firewall, and/or NAT device. Such flexibility and programmability can significantly help network administrators design and manage networks.

- *Synergy with virtualization techniques:* SDN is even more powerful when it comes with virtualization techniques such as network functions virtualization (NFV [61, 125]). A majority of cloud platforms including OpenStack [24], Xen [29] and CloudStack [7] supports a software switch (Open vSwitch [25]) to enable the connectivity between each tenant. SDN can be considered to effectively manage and program the tenant network by dynamically steering network traffics. Recent studies [53, 100] also show that combining SDN with NFV is promising and synergistic.

However, these benefits enabled by SDN come at a cost of security. I introduce four important security challenges in emerging SDN-based network environment. First, *generating reliable policies* in SDN is challenging. More specifically, some network rules generated by an arbitrary SDN application cannot be trusted. There may exist SDN applications (e.g., firewall, load-balancer, route applications) that jointly manage the same network flow. These applications maintain their own policies and necessary network rules will be composed by the SDN controller to implement these policies. However, complex relations among applications' policies may create the *insecure* and *inefficient* generation of network rules. For example, suppose that the controller composes two SDN applications (load-balancer and firewall) sequentially. In this case, rewritten packets by the load-balancer could enable malicious packets to bypass the firewall since the firewall would not be able to see original packet headers. Hence, the careless composition of application policies may cause security breaches in the network. In case network administrators want to compose two SDN applications in parallel to enforce their policies simultaneously. Indeed, some policies in the load-balancer are unnecessary to be composed if the firewall blocks the same network traffic. Therefore, it becomes inefficient to *always* compose the multiple policies.

Second, network rules in the data plane (i.e., switches) create complex relations, which lead to *indirect security violations*. OpenFlow, as the prevailing SDN standard, allows various `set-field` actions that can dynamically change the packet headers in a path. Adversaries could take advantage of this feature to strategically enable flow rules that would evade network security mechanisms. For example, the firewall application has a policy to deny network packets from *host A* to *host C*, namely *flow 1*. Suppose that another application running on the controller establishes a *flow 2* for the connection between *host A* and *host D* by installing a set of network rules, which do not violate the firewall policy. Adversaries then install a new network rule that rewrites the source address (SRC) of the packets to *host B* and the destination address (DST) of the packets to *host C*. In this case, if host A sends a packet to *host D*, the packet will be first processed by the malicious network rule and eventually delivered to *host C*, which violates the firewall policy. This type of network attacks is newly introduced in SDN-based network.

Third, a fundamental limitation of OpenFlow is *the lack of statefulness*. Current OpenFlow mechanism allows SDN controller to generate network policies based on the first packet (e.g., TCP SYN) of a new flow. Because the controller is unaware of subsequent packets of the flow, including state changing packets (e.g., TCP FIN), the controller has no knowledge of the state of connections in the network. This per-flow based OpenFlow workflow is insufficient for enabling state-based network applications (e.g., stateful firewalls) and monitoring suspicious behaviors occurring inside of a flow (e.g., man-in-the-middle attacks). For example, a firewall could specify "packets from server B to host A are allowed, if and only if host A initiates the connection to server B." This stateful policy is incredibly useful for network firewalls when they specify that a web server should accept incoming connections but never initiate an outgoing connection. However, it is impossibly hard to build a stateful firewall in an SDN-based network without the support of stateful packet inspection. In addition, adversaries can attempt unauthorized access to an active connection by performing man-in-the-middle attacks including TCP sequence inference attacks by spoofing packets. If adversaries successively infers the sequence number of the next packet, they could terminate active connections by setting the TCP flags with FIN. With current OpenFlow mechanism, it is challenging to detect this type of attacks occurring in between end hosts.

Lastly, the applicable domains of SDN up to date are heavily restricted to data center networks, and campus networks. This makes following question arise: *"Can SDN help improve security of other networks?"* More specifically, I am interested in adopting SDN into traditional honeynet architecture. The latest honeynet architecture (Gen-III [115, 33]) basically employs a custom firewall called *honeywall* as the gateway of the network to take control of inbound/outbound traffic. However, current honeynet architecture is suffering from its limited functionalities of ’data control’ and ’data capture’. Existing data control mechanism cannot monitor internal propagation of malware in the network, and it does not support honeypot transitions from one to another (e.g., a low-interaction honeypot to a high-interaction honeypot). The data capture capability of traditional honeynet is also restricted as it is vulnerable to fingerprinting attacks.

1.2 Dissertation Statement

To overcome aforementioned challenges, I envision that analyzing network policies and enforcing them in an appropriate manner are imperative in both protecting SDN-based network itself and hardening existing network defense mechanisms. I thus develop following hypothesis for this dissertation:

“Systematic policy management is imperative in software-defined networks (SDNs), and it can help improve security of SDN-based network environment and existing security measures.”

To validate the hypothesis above, I propose a systematic policy management framework for SDN that solves the security challenges and enables policy-driven network defense schemes. To address the first challenge, *generating reliable network rules* in SDN, I propose the grid-based policy decomposition mechanism called RPM that breaks dependency relations across different SDN applications. This mechanism globally examines all application-specific policies to identify overlapping policies and generate disjointed matching space to generate reliable network rules. For the second challenge, which is *indirect security violations*, I propose FlowGuard that categorizes each of violation cases into partial and entire violation and provides automatic and effective resolution mechanisms. Based on the class of violations, it performs four different resolution strategies: dependency breaking, update rejecting, flow removing, and packet blocking. To address the *lack of statefulness* in SDN, I design an innovative *per-connection* monitoring scheme named StateMon by making lightweight extension of current OpenFlow specification (*OpenConnection*). OpenConnection-enabled switch maintains customized flow table to enforce connection-based actions at the data plane and sends them to the controller to inform state-changing events while the controller centrally maintains the states of each connection in the network. To address the last challenge, I introduce HoneyProxy as a next generation SDN-based honeynet. HoneyProxy globally monitors all internal traffic via SDN controller to prevent internal malware propagation, and it enables a novel connection management mechanism across different honeypots to support honeypot transitions. HoneyProxy also improves the data capture capability in existing honeynet by circumventing fingerprinting attacks through multicasting malicious traffic to relevant honeypots and selecting the response which does not contain fingerprinting indicator(s) from them.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows. Chapter 2 first describes generic OpenFlow workflow with essential backgrounds of this dissertation. Next, I elaborate security challenges in generating network rules for SDN in Chapter 3. Chapter 4 discusses detection methodologies and resolution strategies to address *indirect security violations*, which is based on an analysis of stateless network rules. In Chapter 5, I present stateful network monitoring schemes for SDN. SDN-based intelligent honeynet will be presented in Chapter 6 and Chapter 7 concludes this dissertation.

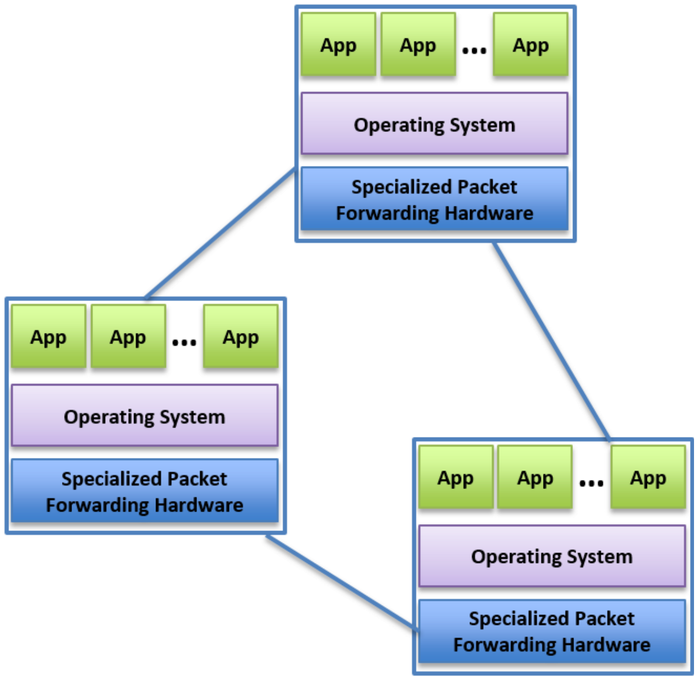
Chapter 2

BACKGROUND

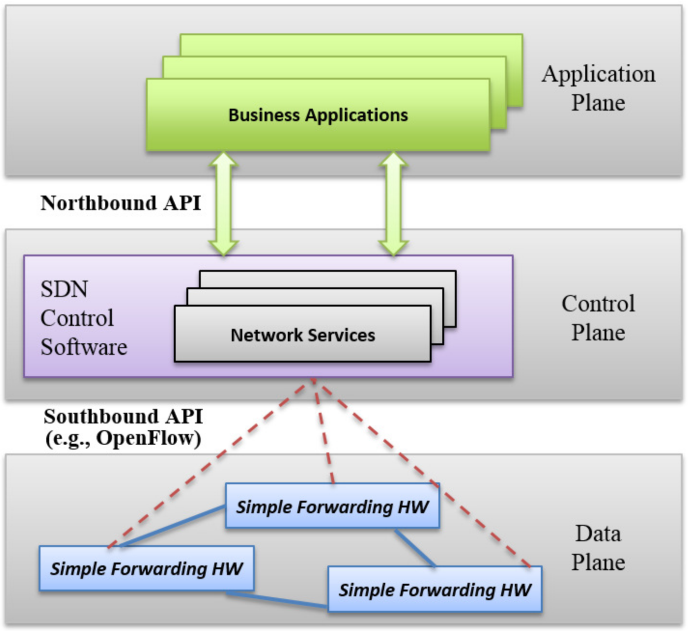
2.1 Software-Defined Network (SDN) and OpenFlow

As illustrated in Figure 2.1, each network box in the conventional network has specialized packet forwarding hardware to process incoming and outgoing packets. Network operating system and applications run on this hardware to control and manage it in an appropriate fashion. However, such an architecture implements rather *"closed"* network environment since all components are bundled in a single box. Consequently, it not only becomes extremely difficult for network programmers to change the configurations but also requires high maintenance costs once these boxes are deployed in the network. In addition, it might not be fully compatible with each other due to different versions of software and/or different vendors.

On the other hand, SDN is trying to break those tight relation between hardware and software by decoupling the data plane from the control plane. SDN takes network operating systems out of the box and consolidates them into the centralized control plane. In this way, network applications can also be consolidated so as to run on top of SDN control plane together. The control plane exposes northbound APIs to the application plane to communicate each other. In the data plane, instead of having specialized forwarding hardware, SDN only needs simplified forwarding hardware because southbound APIs are standardized under the SDN protocols such as OpenFlow [23]. Through OpenFlow, SDN enables *"open"* and programmable network infrastructure and lets network programmers implement business logic and enforce them in a centralized manner.


App
App
...
App
Operating System
Specialized Packet Forwarding Hardware
App
App
...
App
Operating System
Specialized Packet Forwarding Hardware
App
App
...
App
Operating System
Specialized Packet Forwarding Hardware

(a) Conventional network architecture



(b) SDN architecture

Figure 2.1: Architectural Difference Between Conventional Network and SDN.

As the first and widely adopted standard for SDN, OpenFlow [114] essentially implements SDN concepts, and makes the entire network to become directly programmable as well as the underlying infrastructure to be abstracted for network applications. With OpenFlow, only the data plane exists in the network device, and all control decisions are communicated to the device through a logically-centralized controller. To understand how OpenFlow implements SDN concepts, we provide an overview of the current OpenFlow workflow. When an OpenFlow-enabled switch receives a packet, it first checks its *flow tables* to find matching rules. If no such rules exist, this means it is the first packet of a new flow. The switch then forwards the packet to the controller, and it is the controller's job to decide how to handle the flow and to install flow table rules in the appropriate switches [1] . Specifically, the switch encapsulates the raw packet within an `OFPT_PACKET_IN` message to send it to the controller, then the controller installs corresponding rules called *flow entries* into the switches along the controller's intended path for the flow. Once these flow entries are installed, all subsequent packets of this flow are directly forwarded by the switches without sending the packet to the controller.

For example, in Figure 2.2, host A wants to initiate a TCP connection with web server B. The first packet (TCP SYN) sent by host A is checked by the ingress switch S1 and forwarded to the controller because S1 has no matching flow entry for the packet. The controller allows the flow from host A to server B by installing flow entries fe_1, fe_2, and fe_3, into switches S1, S2, and S3, respectively. The flow from host A to server B is called as a *forward flow.* Using the same process, the response packet (TCP SYNACK) generated by server B will trigger the controller to install fe_4, fe_5, and fe_6 into S3, S2, and S1, respectively. The flow from server B to host A

[1]The controller has a global view of the network, so it can calculate the best routing path that the new flow should take in the network.

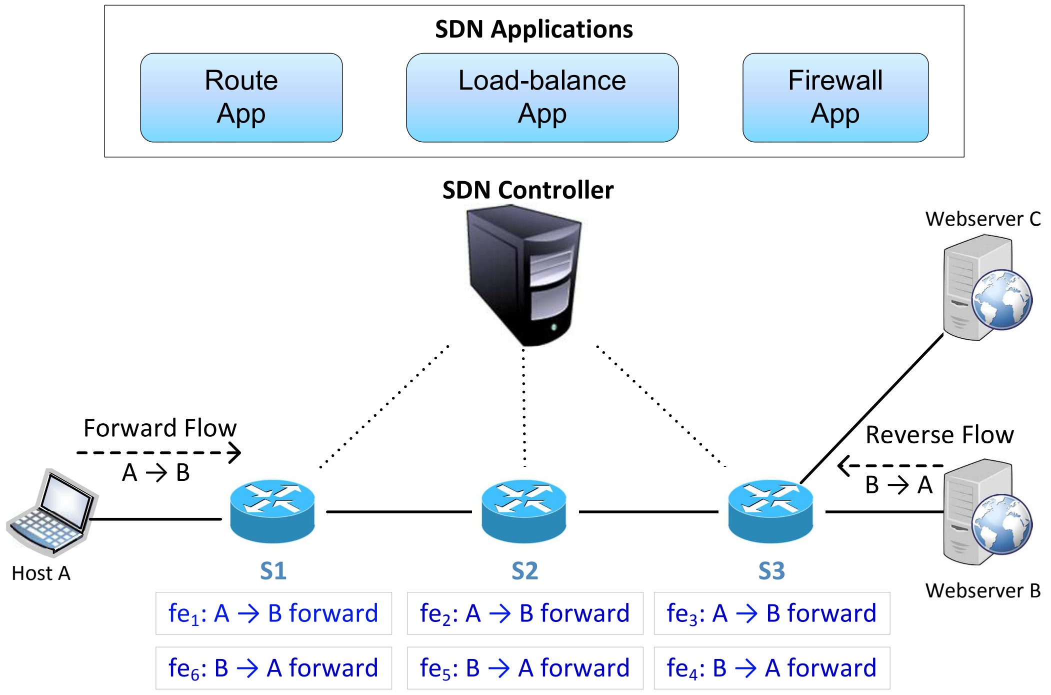


Figure 2.2: Example of a Standard OpenFlow Connection.

is called as a *reverse flow*. Upon the completion of these bi-directional flows, host A can establish a TCP connection with web server B.

2.2 Security Challenges in SDN

There exist several unique security challenges in SDN, which stem from its architectural differences from the conventional network [105, 39, 78, 79, 93, 34]. First, current southbound APIs (e.g., OpenFlow [31]) allow various *set-field* actions that can dynamically change the packet headers. *Adversaries* could take advantage of this feature to strategically enable flow rules that would evade network security mechanisms (e.g., firewalls) [95]. In addition, flow rules may overlap each other in a flow table, indicating the intra-table dependency of flow rules [74]. The rules in a firewall policy may also overlap each other [122]. These rule dependencies could also be leveraged by *malicious* SDN applications to bypass existing security measures such as firewalls.

Second, unprecedented threats are emerging in the control layer due to the nature of SDN, which is centralization. In particular, attackers residing in the data plane may target specific SDN controller by launching DoS attacks or poisoning the network visibility. DoS attacks [110, 53] targeting the controller can be a significant threat since it may result in disrupting all network services running on the same controller. Other attacks such as topology poisoning [65] that tricks a real topology by manipulating LLDP packets would also be possible. In the worst case scenario, attackers would be able to exploit the entire communication channel and hijack sessions between SDN switches and the controller (man-in-the-middle introduced in [105, 39]). Upon successful exploitation, adversaries could take full control of the data plane in the network by manipulating OpenFlow messages destined to SDN switches.

Third, the application plane in SDN is not only suffering from *design flaws* but also vulnerable to *information disclosure and rootkits*. There exist no standard requirement nor rules in implementing the controller or SDN applications, thus design flaws or vulnerabilities may arise depending on their implementation. For example, Rosemary [111] exploits design flaws of existing controller that allows SDN applications to run in the same privilege zone with the controller. By calling system exit function from SDN applications, attackers can eventually crash network services running on the controller. In addition, user-defined or application-specific implementation makes finding malicious applications to become extremely difficult. Malicious users may exploit these inherent limitations to develop and distribute malicious SDN applications to SDN markets such as SDN Dev Center [19]. Consequently, malicious applications could leak network information [105] and install rootkits into the controller [103]. This is why SDN applications cannot be fully trusted and need to be validated.

In this dissertation, we are not trying to solve all types of security challenges in SDN. But we believe many security challenges can be addresses via well-managed network policies. In particular, we are targeting: (1) malicious SDN applications, which will be discussed in Chapter 3; (2) flow modification attacks, which is possibly mitigated using stateless policy management (Chapter 4); and (3) man-in-the-middle attacks, which can be mitigated by stateful policy management (Chapter 5).

2.3 Policy Management and Policy Verification Tools for SDN

A couple of verification tools [35, 74, 73, 76, 85] for checking network invariants and policy correctness in OpenFlow networks have been proposed. Anteater [85] detects violations of network invariants using a SAT solver through transferring the data-plane information to boolean expressions and converting network invariants into instances of SAT problem. FlowChecker [35] translates network policies into boolean expressions and uses Binary Decision Diagram (BDD) to model the network state for checking network invariants. However, both Anteater and FlowChecker are static in nature and could not scale well to dynamic changes in the network. VeriFlow [76] and NetPlumber [74] are capable of checking the compliance of network updates with specified invariants in real time. VeriFlow uses graph search techniques to verify network-wide invariants and deals with dynamic changes. NetPlumber utilizes Header Space Analysis (HSA) [73] in an incremental manner to ensure real-time response for checking network policies through building a dependency graph. Even though these tools can be potentially used to detect network policy violations, they could not provide automatic and effective violation resolution. Also, they ignore rule dependencies within security constraints, such as firewall policies, for compliance checking.

Policy verification tools discussed above are able to check network reachability and potentially utilized for tracking flow paths in OpenFlow networks. However, Anteater [85] and FlowChecker [35] are indeed offline systems and cannot be applied for real-time flow tracking. VeriFlow [76] can perform reachability checking in real time, but it does not support dynamic packet modifications. Another option for flow tracking would be FlowTags [54], which can additionally deal with dynamic transformations in the presence of legacy middleboxes (e.g., proxies). However, FlowTags needs to alter existing OpenFlow architecture. In this dissertation, we leverage the mechanism introduced by NetPlumber [74] to track flow paths for network policy violation detection, since NetPlumber provides a couple of features that fit for our purpose, such as support for arbitrary header modifications, automatic rule dependency detection, and real-time response.

As another work, FortNOX [95] was proposed as a software extension aiming to provide security constraint enforcement for OpenFlow controllers, being able to identify *indirect* security violations. FortNOX was then used as a security enforcement kernel for FRESCO [109], an OpenFlow security application development framework. However, we cannot directly adopt FortNOX approach due to several reasons. On one hand, the rule conflict analysis algorithm provided by FortNOX records rule relations in alias sets, which are unable to accurately track all flows. In particular, the conflict detection algorithm in FortNOX only conducts *pairwise* conflict analysis between new flow rule(s) and each single security constraint without considering *rule dependencies* within flow tables [74, 76] and among security constraints (represented as a firewall policy in our approach) [63, 122]. On the other hand, when FortNOX detects a security violation caused by new rule(s) installed by a non-security application, it simply rejects the rule(s) without offering a fine-grained violation resolution.

Chapter 3

MEDIATING POLICY CONFLICTS FOR SDN APPLICATIONS

3.1 Introduction

Traditional network environment is ill-suited to meet the requirements of today's enterprises, carriers, and end users. Software-Defined Networking (SDN) is recently introduced as a new network paradigm which is able to provide unprecedented programmability, automation, and network control by decoupling the control and data planes, and logically centralizing network intelligence and state [49]. In SDN, network applications can communicate with the SDN controller via an open interface and define network-wide policies based on a global view of the network provided by the controller. The SDN controller, which resides in the control plane, manages network services and provides an abstract view of the network to the application plane. At the same time, the controller translates policies defined by applications into actual packet processing rules which are identifiable by the data plane. As the first standard for SDN, OpenFlow [86] helps generate a set of flow rules to enforce network-wide policies in physical devices. Each flow rule specifies a *pattern* that matches on bits in the packet header, *actions* that are performed on matching packets to describe packet forwarding, packet modification or packet dropping, a *priority* that disambiguates among overlapping patterns, and *timeouts* that allow a switch to delete expired rules.

The multi-layered SDN architecture significantly helps manage and process network flows. However, each layer of SDN architecture heavily relies on complicated network policies and managing those policies in SDN requires not only dedicated cautions but also considerable efforts. Our study reveals that such a multi-layered architecture brings great challenges in policy management for SDN as follows:

- **Policy management in SDN application plane**: An SDN application could employ multiple modules, such as access control, load-balancing, routing and monitoring, to process the same flow and generates various functional policies by composing rules produced by those modules [57, 89]. However, policy composition is not a trivial task, since rules generated by a single module may overlap each other (intra-module dependency) and rules from one module may also overlap with rules from other modules (inter-module dependency). Thus, policy composition should address issues caused by both intra-module and inter-module dependencies.

- **Policy management in SDN control plane**: In SDN control plane, there may exist multiple SDN applications running on top of a controller and they might jointly process the same traffic flow. In such a situation, flow rules generated by different application for processing the same flow may also overlap each other (inter-application dependency) and lead to policy conflicts [56, 95].

- **Policy management in SDN data plane**: In SDN data plane, different flows may go through the same switches and flow rules defining different flows in the same flow table may also overlap each other. We call this situation *intra-table dependency*. In this case, unintended flow path modification could happen.

To address above challenges, we propose a framework for robust policy management (RPM) in SDN with respect to three planes in SDN architecture. In SDN application plane, we introduce a policy segmentation mechanism to compute and resolve intra-module and inter-module dependencies and enable a secure and efficient policy generation. Our novel policy segmentation mechanism generates a number of disjoint segments which are able to identify various dependencies and thereby allow to automatically remove those dependencies. In SDN control plane, our framework identifies inter-application dependencies with the help of policy segmentation mechanism and resolves the dependency relations through two ways in terms of different situations. On one hand, if different applications are desired to collaboratively process the same flow, our framework composes policies produced by those applications . On the other hand, when the applications are mutually exclusive and each time only one application is allowed to process the flow, our framework breaks inter-application dependencies by assigning policies from different application with different priorities. Lastly, we propose an flow isolation mechanism for removing intra-table dependencies to address conflicting flows in SDN data plane.

The major contributions of this chapter are summarized as follows:

- We present various challenges in SDN policy management with respect to three planes, application, control and data planes, in the multi-layered SDN architecture.
- We propose a comprehensive framework to enable a robust policy management in SDN based on three layers of SDN architecture. A set of systematic resolution strategies are introduced for different layers in our framework.
- We provide a prototype implementation of our framework in an open SDN controller. We evaluate our solution using a real-world network configuration and an emulated OpenFlow network. Our experimental results show that our implementation has low performance overhead to enable effective policy management for SDN.

This chapter is organized as follows. Section 5.3.2 overviews our framework and presents policy management challenges and corresponding resolution strategies based on three planes in SDN architecture. In Section 6.6, we introduce our implementation details and evaluations followed by related work discussed in Section 6.9. Section 6.10 concludes the chapter.

3.2 Robust Policy Management (RPM) Framework

3.2.1 Overview

We first present our RPM framework which enables robust policy management for SDN in terms of three planes of SDN architecture: (1) SDN application plane; (2) SDN control plane; and (3) SDN data plane.

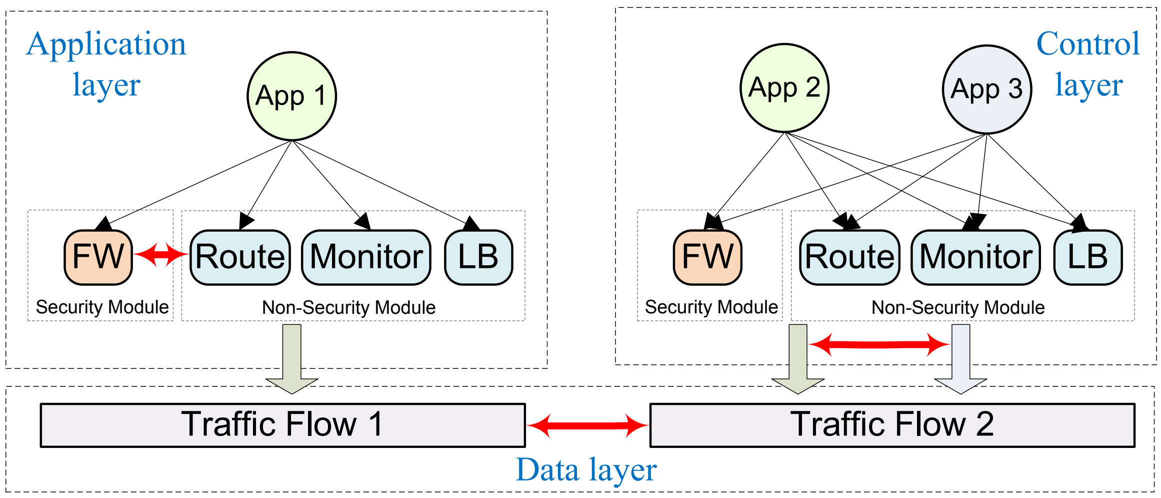


Figure 3.1: Multi-layered SDN Policy Management: (1) Application Plane; (2) Control Plane; and (3) Data Plane.

In SDN application plane, a main issue comes from policy composition where intra-module and inter-module dependencies should be addressed. Partially or entirely overlapped rules in a module make nontrivial intra-module dependencies and complicate the process of policy composition. In addition, inter-module dependencies between security and non-security modules may cause security challenges due to incomplete policy composition. As illustrated in Figure 3.1, we investigate inefficient and insecure policy composition issues and introduce our policy generation algorithm along with a policy segmentation mechanism to address such issues.

In SDN control plane, multiple applications processing the same flow may cause inter-application dependencies. As shown in Figure 3.1, `App 2` and `App 3` is processing the same flow, `Flow 2`, and the flow policies produced by two applications may conflict with each other. In order to identify inter-application dependencies, we recall the policy segmentation mechanism in the first layer and obtain overlapping segments which indicates the rule dependency relations. We consider two resolution strategies in this layer. First, we compose policies generated by different applications and allow them to jointly process the same flow. Second, we break inter-application dependencies by assign dependent rules with different priorities.

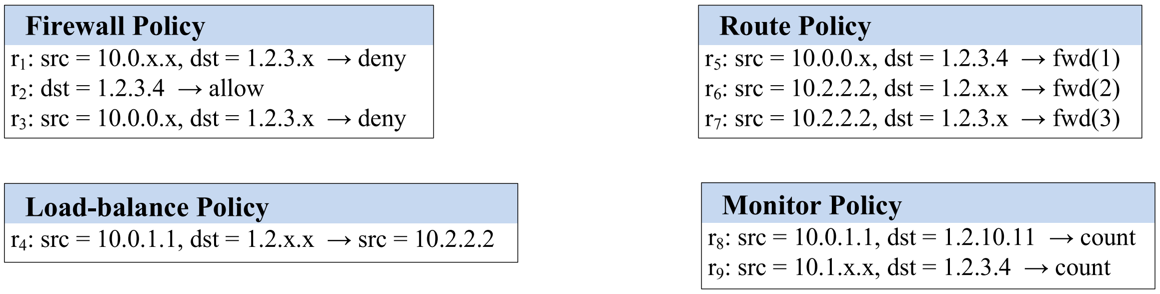
In SDN data plane, OpenFlow-enabled switches store flow rules into the flow tables by their priorities. A rule defining one flow, such as `Flow 1` in Figure 3.1, with a lower priority might be affected by a rule for another flow, such as `Flow 2` in Figure 3.1, with a higher priority, causing intra-table dependency. Since intra-table dependencies might change the behaviors of associated flows, our framework provides two flow isolation mechanisms, flow rerouting and flow tagging, to address such a issue.

3.2.2 Policy Management in SDN Application Plane

In this section, we first explore various considerations and policy management challenges in SDN Application Plane. We then present our fine-grained policy composition mechanism in the RPM framework.

Considerations and Challenges

An SDN application generally employs several network modules to build multi-functional policies. While an application with multiple modules processes a traffic flow, fundamental considerations are intra-module and inter-module dependencies. To illustrate policy composition issues, we adopt two kinds of composition operators introduced in [89]. "*Parallel*" composition operator (|) means the *union* of two modules and generates a set of packet processing rules which should be applied to the same flow simultaneously. "*Sequential*" composition operator ($\gg$) stands for serialization of modules so that the matching rules would be performed one by one on a flow. We next investigate several policy management challenges in SDN application plane.



Firewall Policy

r_1: src = 10.0.x.x, dst = 1.2.3.x → deny
r_2: dst = 1.2.3.4 → allow
r_3: src = 10.0.0.x, dst = 1.2.3.x → deny

Route Policy

r_5: src = 10.0.0.x, dst = 1.2.3.4 → fwd(1)
r_6: src = 10.2.2.2, dst = 1.2.x.x → fwd(2)
r_7: src = 10.2.2.2, dst = 1.2.3.x → fwd(3)

Load-balance Policy

r_4: src = 10.0.1.1, dst = 1.2.x.x → src = 10.2.2.2

Monitor Policy

r_8: src = 10.0.1.1, dst = 1.2.10.11 → count
r_9: src = 10.1.x.x, dst = 1.2.3.4 → count

Figure 3.2: Sample Policies Defined by Four Different Network Modules.

1. *Intra-module dependency*: Assume that there exist four different modules, Firewall, Load-balance (LB), Route and Monitor, which can be used by an SDN application. And each module produces several rules which are ordered by their priorities as shown in Figure 3.2. In Firewall policy, r_1, r_2, and r_3 are mutually dependent. Thus, computing intra-module dependencies in Firewall policy requires considerable efforts. Computing dependencies in Route policy is relatively easy, since r_6 is a superset of r_7 and r_7 is not visible in the network. On the other hand, LB and Monitor policies do not have any intra-module dependencies.

2. *Inter-module dependency*: Computing inter-module dependencies is more tricky. In Figure 3.2, we can observe that r_1 is dependent with r_4 and r_5, r_2 is dependent with r_9, and r_3 is dependent with r_5. Therefore, Firewall policy is

dependent with all other modules' policies. But Route policy is only dependent with Firewall policy, since r_5 is dependent with r_1, r_2, and r_3. As discussed above, computing inter-module dependencies by *pair-wise* comparison requires considerable efforts.

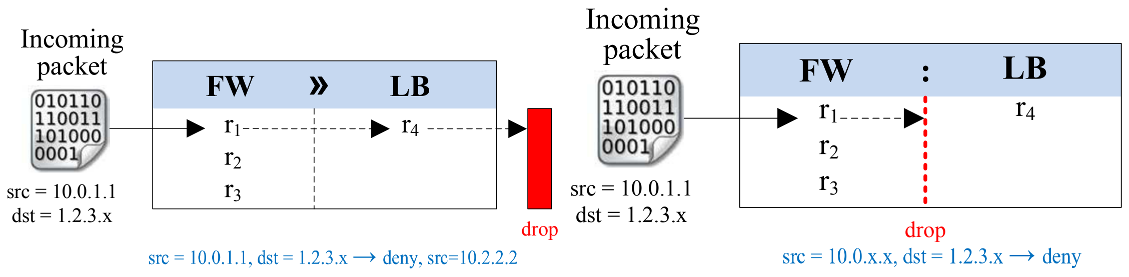
3. *Insecure composition caused by inter-module dependency*: We now explore problematic issues potentially caused by inter-module dependencies. We first assume that two modules are sequentially composed, $Firewall \gg LB$. Since r_1 and r_4 internally overlap with each other, the packets matching r_4 will be blocked by r_1. However, if we consider an opposite composition sequence, $LB \gg Firewall$, r_4 in LB modifies packets' source IP address to 10.2.2.2 and r_1 in Firewall cannot block these packets. In such a case, we argue that inaccurate composition sequence may cause security breaches in the network.

4. *Inefficient composition caused by intra-module and inter-module dependencies*: A programmer may want to compose two modules in parallel such as $Firewall \mid Route$. In Figure 3.2, we could observe that all rules in Firewall policy are dependent with r_5. Since r_1 has the highest priority, r_1 and r_5 are jointly combined and we can obtain the following rule: $src = 10.0.0.x, dst = 1.2.3.4 \longrightarrow deny, fwd(1)$. Indeed, r_5 is not necessary to compose with Firewall rules, since the Firewall rule, r_1, already blocks packets matching the rule pattern. In addition, composing r_6 and r_7 with Firewall policy is also invalid since r_7 should not appear in the network. Therefore, we argue that it is obviously inefficient to *always* compose the multiple policies and install them into the network switches like Pyretic [89] does.

As discussed above, we need to pay special attention on addressing challenges in this layer. First, treating every module equally sometimes evades security policies such as a firewall policy. Since a firewall policy is generally considered more important than policies produced by other modules, distinguishing security modules from non-security modules is vital for secure policy composition. Second, commodity SDN switches with limited ternary content addressable memory (TCAM) space typically support only a few thousands of rules [72, 117]. Thus, we should also strive to provide mechanisms with respect to an efficient policy composition.

Efficient Policy Generation

To efficiently compose policies generated by different modules of an SDN application, we group security modules and non-security modules separately and carefully examine the inter-module dependencies between security modules and non-security modules.



(a) Inefficient flow rule generation (b) Efficient flow rule generation

Figure 3.3: Comparison Between Inefficient and Efficient Flow Rule Generation.

Figure 3.3 shows an example comparing inefficient policy generation mechanism and our solution. In Figure 3.3a, Firewall and Load-balance (LB) modules are composed sequentially so that only legitimate packets can traverse the network. In this case, the generated packet processing rule contains an unnecessary rule, r_4, which modifies source IP address to 10.2.2.2. Such an extra operation will impose overhead to the production network. In contrast, our approach ignores subsequent policies generated by non-security modules if security modules deny matching packets. Figure 3.3b illustrates our approach where r_1 drops the matching packets immediately without the processing of r_4. We use a new notation (:) to indicate our policy generation operation.

Policy Segmentation

The major goal of our framework in this layer is to remove intra-module and inter-module dependencies for policy composition. In order to achieve such a goal, computing these dependencies is essential. Since SDN applications can obtain a global view of network, our segmentation algorithm also adopts global data structure which allows us to compute intra-module and inter-module dependencies at the same time. Our policy segmentation mechanism generates a set of disjoint segments.

Definition 1 ***(Segment).*** *A segment is a 4-tuple* $\{s_{id}, ms, R_a, OR\}$, *where* s_{id} *is a segment identifier,* ms *is a matching space of segment,* R_a *is a set of active rules, and* OR *is a set of overlapping rules. An element of* R_a *is selected from* OR *by finding out a rule having the highest priority in the overlapping rule set associated with a module. Therefore,* $R_a \subseteq OR$.

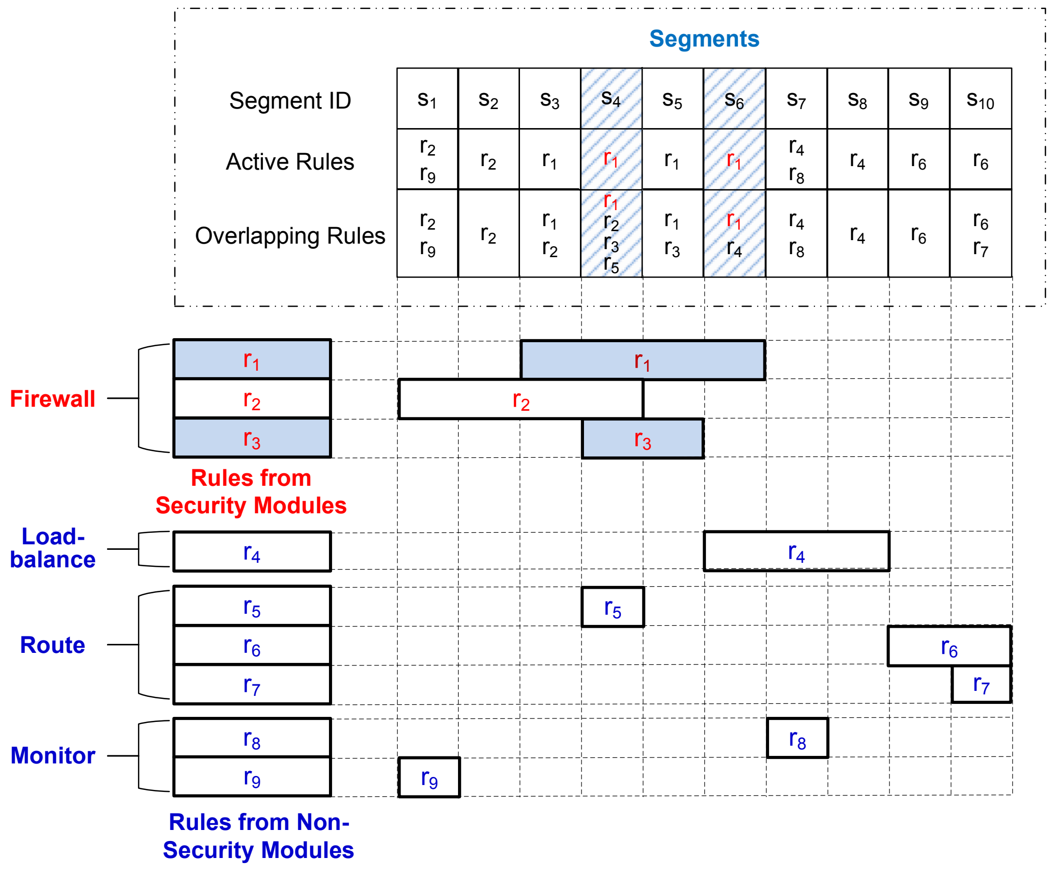
Our policy segmentation mechanism globally examines overlapping rules and computes distinct matching space in order to remove intra-module and inter-module dependencies. To compute dependencies, we record all overlapping rules in each segment. Overlapping rules indicate those rules are possibly have intra-module or inter-module dependencies so that we could remove all dependencies and obtain active rules. Active rules are completely applicable to process matching packets because dependencies in corresponding overlapping segments could be completely removed.

Algorithm 1 shows the pseudocode of our policy segmentation mechanism. We first insert security policies into a set of disjoint segments. As shown in lines 17-40

Algorithm 1: Policy Segmentation Algorithm.

```
Input: A security policy $P_s$ and non-security policies $P_{ns} = \{P_1, \cdots, P_k\}$.
Output: A set of segments $S$.
PolicySegmentation($R$)
$S.New()$;
/*Insert security module rules*/
$R \longleftarrow GetRule(P_s)$;
foreach $r \in R$ do
    $S \longleftarrow InsertRule(S, r)$;
end
/*Insert non-security module rules*/
foreach $P \in P_{ns}$ do
    $S.AppendModuleSeperator()$;
    $R \longleftarrow GetRule(P)$;
    foreach $r \in R$ do
        $S \longleftarrow InsertRule(S, r)$;
    end
end
/*Compute active/inactive rules in each module*/
foreach $s \in S$ do
    $s \longleftarrow ComputeActiveRule(s)$;
end
return $S$;
InsertRule($S, r$)
$s_n \longleftarrow MatchingSpace(r)$;
$s_n.OverlappingRules.Append(r)$;
foreach $s \in S$ do
    /* $s_n.ms$ is a subset of $s.ms$*/
    if $s_n.ms \subset s.ms$ then
        $s.ms \longleftarrow s.ms \setminus s_n.ms$;
        $s_n.OverlappingRules.Append(s.OverlappingRules)$;
        $S.Append(s_n)$;
        $break$;
    end
    /* $s_n.ms$ is a superset of $s.ms$*/
    else if $s_n.ms \supset s.ms$ then
        $s_n.ms \longleftarrow s_n.ms \setminus s.ms$;
        $s.OverlappingRules.Append(s_n.OverlappingRules)$;
    end
    /* $s_n.ms$ partially matches $s.ms$*/
    else if $s_n.ms \cap s.ms \neq \emptyset$ then
        $s_c$.New();
        $s_c.ms \longleftarrow s_n.ms \cap s.ms$;
        $s_c.OverlappingRules.Append(s_n.OverlappingRules \cup s.OverlappingRules)$;
        $S.Append(s_c)$;
        $s.ms \longleftarrow s.ms \setminus s_c.ms$;
        $s_n.ms \longleftarrow s_n.ms \setminus s_c.ms$;
    end
end
$S.Append(s_n)$;
return $S$;
ComputeActiveRule($s$)
if $s.OverlappingRules \cap P_s \neq \emptyset$ then
    $R \longleftarrow s.OverlappingRules \cap P_s$;
    $s.ActiveRules.Append(GetHighPriorityRule(R))$;
    if $s.RuleType = deny$ then
        return $s$;
    end
end
else
    foreach $P \in P_{ns}$ do
        if $s.OverlappingRules \cap P \neq \emptyset$ then
            $R \longleftarrow s.OverlappingRules \cap P$;
            $s.ActiveRules.Append(GetHighPriorityRule(R))$;
            $continue$;
        end
    end
end
return $s$;
```

in Algorithm 1, *InsertRule* function computes disjoint matching space of a segment and substitutes newly updated matching space to old one for resident segments. At the same time, each segment stacks up overlapping rules. A function called *Compute-ActiveRule* generates active rules of segments (lines 41-53). To achieve an efficient policy generation, this function compares denying rules first (line 45) so that it only obtains active denying rule without computing additional relations.



Segment ID	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}
Active Rules	r_2 r_9	r_2	r_1	r_1	r_1	r_1	r_4 r_8	r_4	r_6	r_6
Overlapping Rules	r_2 r_9	r_2	r_1 r_2	r_1 r_2 r_3 r_5	r_1 r_3	r_1 r_4	r_4 r_8	r_4	r_6	r_6 r_7

Figure 3.4: Policy Segmentation.

For example, assume that a programmer installs two rules r_6 and r_7. Since r_6 has no dependent rules, our algorithm will add a segment, s_9, directly. Regarding r_7, it examines if the relation between r_6 and r_7 satisfies one of the following relations: subset, superset, partial match, or disjoint. Since r_7 is a subset of r_6, we compute intersection between r_7 and s_9. Then, Algorithm 1 appends new segment s_{10} and updates overlapping rules of s_{10}. After appending a set of segments, Algorithm 1 calls *ComputeActiveRule* function to compute active/inactive rules. Since there doesn't exist denying rule, both segments, s_9 and s_{10}, choose r_6 as their active rules.

Figure 3.4 illustrates a detailed policy segmentation. The OR of segment s_1 are r_2 and r_9. R_a is the same as OR, since r_2 and r_9 are defined in different modules. However, R_a of segment s_{10} is r_6, since r_6 and r_7 are defined in the same module and the priority of r_6 is higher than the priority of r_7. Note that our algorithm applies different operations on segments s_4 and s_6. Since they have a denying rule r_1 as an active firewall rule, the rest of overlapping rules are ignored.

Policy Composition

Algorithm 2 presents how our policy composition mechanism leverages the results of policy segmentation to generate composed policies. This algorithm imports a set of segments S and composing definition D, which consists of two modules with the composition operator. Policy segmentation provides a set of disjoint segments with corresponding active rules in each segment. The algorithm only computes active rules in each segment (lines 5-20) and checks M_1 first. If any element of active rules overlaps with M_1, the algorithm keeps composing with M_2. Since the algorithm deals with two kinds of composition operators, it performs two different tasks, sequential composition and parallel composition.

3.2.3 Policy Management in SDN Control Plane

If different applications attempt to process the same flow, there may exist control plane conflicts caused by inter-application dependencies. We describe such conflicts in depth with a motivating example and provide corresponding resolution strategies.

Algorithm 2: Policy Composition

Input: A set of segments S and composing definition $D = \{M_1, M_2, type\}$, where $D.type = \{\gg, |\}$.

Output: A set of packet processing rule $PR = \{pr_1, pr_2, \cdots, pr_k\}$, such that $pr_i = \{ms, action\}$.

```
1  PolicyComposition($S, D$)
2  $PR.New()$;
3  /*Insert security module rules*/
4  foreach $s \in S$ do
5      if $s.ActiveRules \cap D.M_1 \neq \emptyset$ then
6          $pr.New()$;
7          $pr.ms \longleftarrow s.ms$;
8          $pr.action \longleftarrow (s.ActiveRules \cap D.M_1).action$;
9          if $D.type ='\gg'$ then
10             if $s.ActiveRules \cap D.M_2 \neq \emptyset$ then
11                 $pr.action \longleftarrow pr.action \cup (s.ActiveRules \cap D.M_2).action$;
12             end
13             else
14                 $temp.ms \longleftarrow ComputeMachingSpace(pr.ms, pr.action)$;
15                 if $IsDependent(temp.ms, D.M_2)$ then
16                     $pr.action \longleftarrow pr.action \cup ComputeNextAction(temp.ms, D.M_2)$;
17                 end
18             end
19         end
20         else if $D.type ='|'$ then
21             if $s.ActiveRules \cap D.M_2 \neq \emptyset$ then
22                 $pr.action \longleftarrow pr.action \cup (s.ActiveRules \cap D.M_2).action$;
23             end
24         end
25         $PR.Append(pr)$;
26     end
27     else if $s.ActiveRules \cap D.M_2 \neq \emptyset$ then
28         $pr.New()$;
29         $pr.ms \longleftarrow s.ms$;
30         $pr.action \longleftarrow (s.ActiveRules \cap D.M_2).action$;
31         $PR.Append(pr)$;
32     end
33 end
34 return $PR$;
```

Inter-application Dependency

In this layer, our framework deals with inter-application dependencies when different applications process the same flow. The root cause of those dependencies is that each application wants to enforce its own policy over other policies in order to process the same flow. Figure 3.5 illustrates those dependencies more precisely.

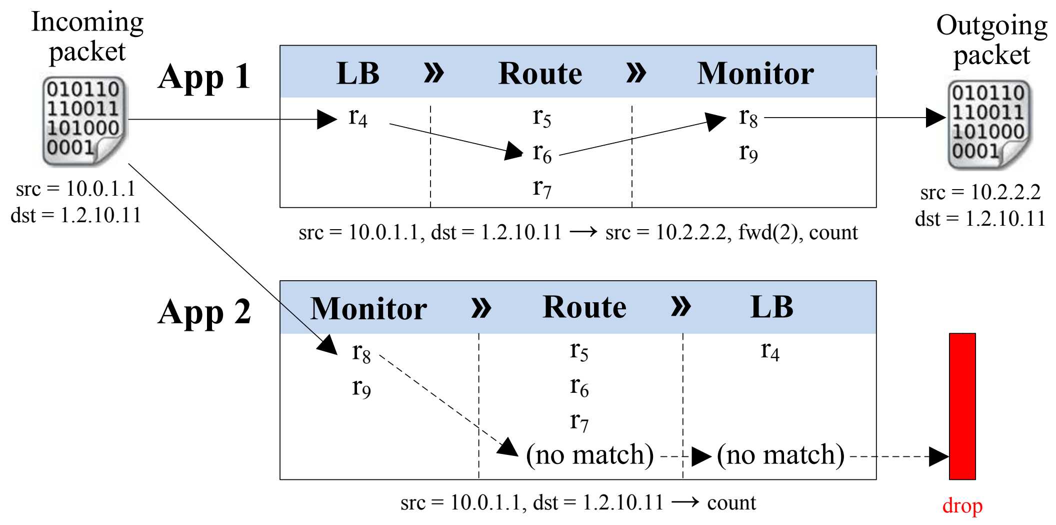


Figure 3.5: Different Applications Cause Control Plane Conflicts While Managing the Same Flow.

The `APP 1` composes Load-balance (LB), Route, and Monitor modules sequentially contrary to the `APP 2` which composes Monitor module first. Incoming packets matching source IP address 10.0.1.1 and destination IP address 1.2.10.11 is managed by two different applications since both r_4 in LB and r_8 in Monitor have been defined to handle these packets. The `APP 1` accepts a packet and modifies source IP address to 10.2.2.2 and forwards it to port 2 and counts it sequentially. On the other hand, the `APP 2` drops it after counting it because there exist no matching rules in Route and Monitor modules. Runtime will generate these flow rules at the same time so that there exist confusion on production network.

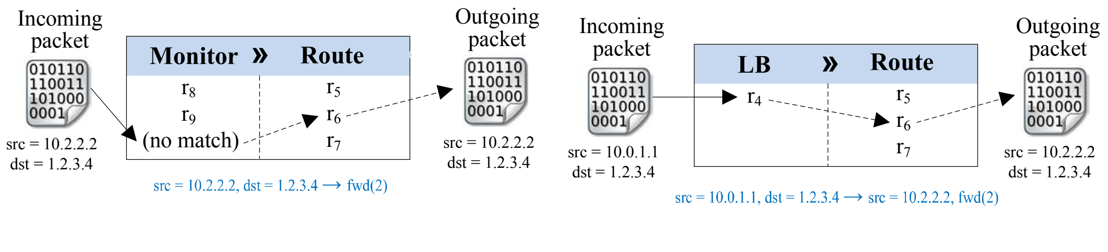
In order to identify these conflicts, we need to recall policy segmentation mechanism in order to extract R_a of each segment. As shown in Figure 3.4, policy segmentation indicates R_a of s_7 is r_4 and r_8. So we obtain the new overlapping module set, $OR_M = \{LB, Monitor\}$, since $r_4 \in LB$ and $r_8 \in Monitor$. And let us make another set A_1, and A_2 for the `APPs 1` and `2` to enumerate modules used by each application, $A_1 = \{LB, Route, Monitor\}$ and $A_2 = \{Monitor, Route, LB\}$. Since $A_1 \cap A_2 = \{LB, Route, Monitor\} \supseteq OR_M = \{LB, Monitor\}$, we conclude that there exist potential conflicts between two applications. It is easy to detect whether different applications have potential conflicts or not. If OR_M is a subset of $\cap_i M_i$, we conclude that these applications have potential conflicts.

Resolution Strategy

We consider two situations: one is that different applications are allowed to jointly combine each policy and another is that application are mutually exclusive. For the first case, we accept inter-application dependency so that we apply parallel composition operator to combine two policies. As an alternative, we remove inter-application dependency by assigning different priorities to conflicting applications. By assigning different priorities to conflicting applications, we enforce the policy defined by the application having the higher priority for the same flow. And we also enforce one of policies in a similar way by assigning different priorities to the applications. For example, an application that employs security module may have a upper priority so that this application takes the precedence over other normal applications. Different conflict resolution strategies proposed by our previous work [66] could be also applied to resolve inter-application dependencies caused by conflicting applications.

Intra-table Dependency

Different applications normally do not make any problematic issues when they manage two distinct flows. However, if the flow paths of the different flows overlap each other, intra-table dependency should be identified to check potential violations.



(a) Flow 1 processed by App 1 (b) Flow 2 processed by App 2

Figure 3.6: Two Applications Manage Two Different Flows.

For example, there exist two traffic flows processed by different applications as shown in Figure 3.6. An application, `App 1`, generates the flow rule which accepts the packet matching source IP address 10.2.2.2 and destination IP address 1.2.3.4 and forwards to the port 2. On the other hand, another application, `App 2`, priorities combines two network modules so that it obtains the flow rule which accepts different packet and modifies source IP address to 10.2.2.2 and forwards to the port 2. Even though incoming packets are different, *flow space* [108] of outgoing packets are overlapped so that one of applications might lose its control of the flow. In particular, if different flows have common visiting switch along with their flow paths, the flows can be affected by different priorities so that original flow paths would be changed.

Resolution Strategy

Our goal in this layer is to remove intra-table dependency through flow isolation. To achieve this goal, we consider two flow isolation methods, *flow tagging* and *flow rerouting*.

Inspired by the approach discussed in [54], which utilizes tags to distinguish packets belonging to different versions of policies for ensuring consistent network updates, we can use tags to delete the dependencies. In this mechanism, the new flow policy is preprocessed by adding a tag to differentiate the match pattern with other policies. The rule of the policy in the ingress switch will take additional action on the packets to stamp them with the same tag. As the packets leave the network, in the egress switch, the corresponding rule of the policy will strip the tag from the packets.

Using flow rerouting, when a flow path causes an intra-table dependency, we request the controller to find another routing path for the policy to avoid the dependency. During this process, we maintain a list called *switch evading list* that contains all switches associated with the intra-table dependency. We provide such a list to the controller, and then the controller will calculate a new routing path ignoring those switches in the list to break the dependency.

3.3 Implementation and Evaluation

We have implemented our framework on top of an open SDN controller, Floodlight [11]. Our proof-of-concept implementation consists of three major components: (1) policy generation; (2) policy segmentation; and (3) resolution strategy. The policy generation component imports policies created by applications to generate multi-functional policies, which are then used by the controller to generate corresponding flow rules and install them in OpenFlow-enabled switches. The policy segmentation component captures every flow rule to produce a set of segments, which are able to identify intra-module and inter-module dependencies. The resolution strategy component obtains a global view of network from the Floodlight controller using northbound APIs [11] and implements different resolution strategies as described in Section 5.3.2.

All of our experiments were performed with Floodlight v0.90 and Mininet v2.1.0 [22]. We obtained a *real-work* network configuration from *Stanford* backbone network [13], which has 26 switches with corresponding ACL rules, for our experiments. We removed redundant ACL rules and converted them to a firewall policy and in turn obtained $1,206$ firewall rules in total. At the same time, we generated $8,908$ network rules by parsing original network rules existing in *Stanford* network configuration to Floodlight-recognizable rules. Because these network rules contain routing rules as well as rewriting rules, we assume that these rules are generated by two modules, *Route* and *Load-balance* (LB) modules.

Our policy generation mechanism enables efficiently composing multiple modules and in turn generating a number of flow rules. To evaluate our policy generation mechanism, we performed experiments in two ways: (1) $Firewall : (Route|LB)$ [1] ; and (2) $Firewall >> (Route|LB)$. Suppose we only have one denying rule in firewall policy which blocks a number of network rules.

Without our policy generation mechanism, in case (2), the system generates exactly the same number of rules, $\mid Firewall \mid + \mid (Route|LB) \mid$, which is the worst case shown in Figure 3.7. Every single rule in $(Route|LB)$ may overlap with firewall policy. Thus, in the best case, our mechanism only generates firewall rules without $(Route|LB)$ rules. The average case means that half of $(Route|LB)$ rules overlap with firewall policy. Thus, the system installs the following number of rules, $\mid Firewall \mid + \frac{1}{2} \mid (Route|LB) \mid$. As shown in Figure 3.7, the best case constantly

[1]Note that our operator (:) means that it adopts our policy generation mechanism.

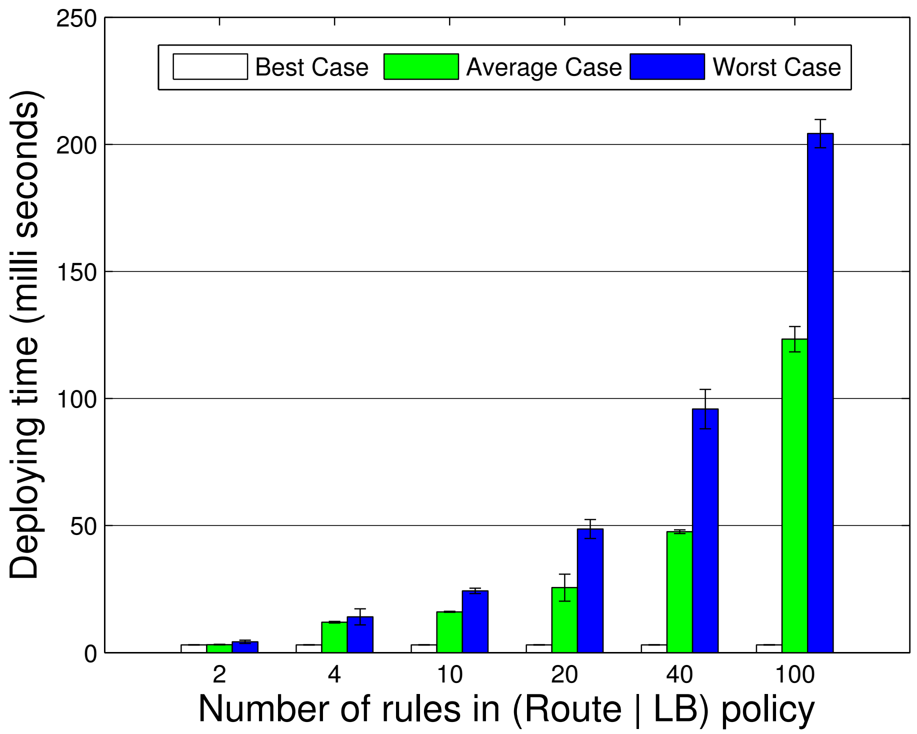


Figure 3.7: Comparison Between Inefficient VS. Efficient Policy Generation Mechanism.

takes only 3 milliseconds for both generating and installing flow rules to corresponding switches. On the other hand, the deployment time of the worst case increases dramatically in accordance with the growing number of rules in $(Route|LB)$.

To evaluate computing overheads of our policy segmentation mechanism, we installed all network rules into the network and measured the updating time of policy segments. As a result, 456 segments out of 688 firewall rules were produced by the policy segmentation mechanism, $8,273$ segments out of $8,908$ network rules were generated, and we got $8,729$ segments in total. Because there exist some redundant rules, the number of segments is less than the total number of rules, $\mid Firewall \mid + \mid (Route \mid LB) \mid$. 75% of updates have been finished within 0.2 milliseconds and most of cases (98%) have been computed less than 0.5 milliseconds.

We also evaluated the performance of two resolution strategies: assigning priorities

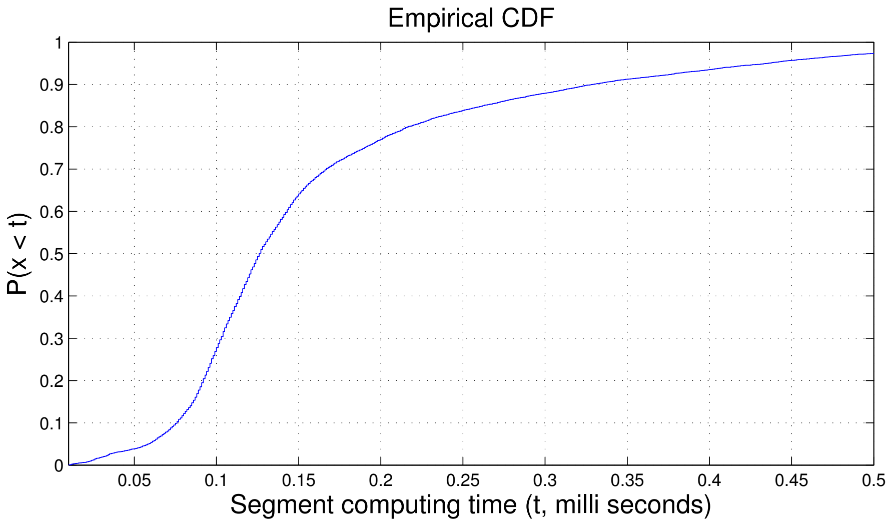


Figure 3.8: Cumulative Distribution Function for Segment Generation Time.

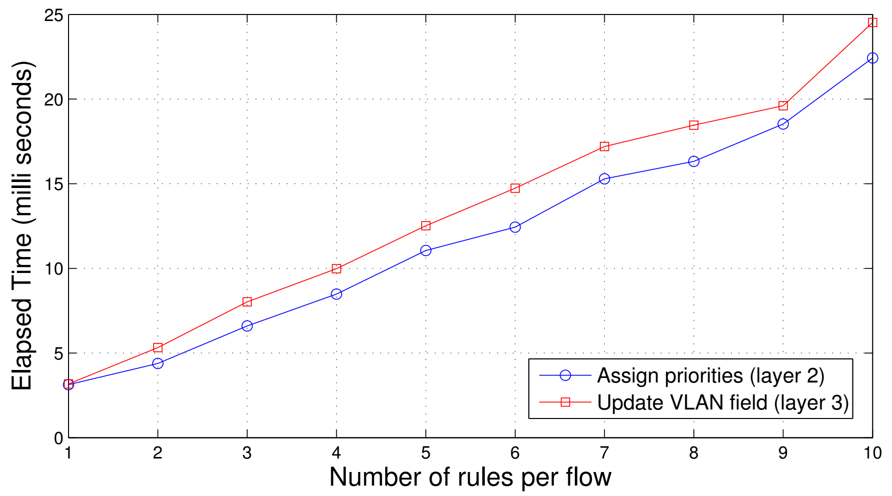


Figure 3.9: Evaluation of Two Resolution Strategies: Assigning Priorities for the SDN Control Plane and Updating VLAN Field for the SDN Data Plane.

for the SDN control plane and updating VLAN field for the SDN data plane. Both resolution strategies update a set of rules which define conflicting flows. First, we measured elapsed time for assigning priorities of rules. As shown in Figure 3.9, the elapsed time grows in accordance with the growing number of rules per flow. Similarly, we checked the elapsed time for updating VLAN field for isolating conflicting flows. The elapsed time increases with the growing number of rules per flow, but it generally took more time than the assigning priorities case, since the systems spent more time finding out ingress/egress switches of the flows to add and strip VLAN tags.

3.4 Related Work

Modular network programming has recently received considerable attention in SDN community [109, 57, 89, 77, 45]. However, this area is still immature and needs in-depth investigation in terms of security and effectiveness of module composition. Pyretic [89] enables a program to combine different policies generated by different modules together using policy composition operators. Then, combined policies can be propagated and enforced at SDN switches. However, lacking a policy dependency detection mechanism in Pyretic, it is obviously inefficient to *always* compose the multiple policies and install them into the network switches. Although FRESCO [109] deals with security application development framework using modular programming for SDN, it couldn't directly handle dependencies between modules in SDN applications. In contrast, our framework deals with various dependencies such as intra-module, inter-module, inter-application, and intra-table dependencies for robust SDN policy management.

Many research efforts have been recently devoted to the policy validation and enforcement mechanism in SDN due to programmability of SDN. Traditional policy checking mechanism such as binary decision diagrams (BDD) has been widely employed for checking anomalies of a firewall [66, 123, 52] which is not efficient for verifying real-time systems. Newly emerging network analysis tools such as NetPlumber [74] and VeriFlow [75] support real-time validation and verification of networks. Several middlebox approaches [97, 54] can deal with dynamic packet modification made by a number of rewriting flow rules in OpenFlow networks. Meanwhile, NICE [44] was proposed to adopt a model-checking framework which utilizes symbolic execution to automate testing process for OpenFlow applications. Even though our policy management framework imposes manageable overheads in SDN policy management, we will further study a more practical and efficient mechanism supporting analysis and enforcement of network policies in real time.

3.5 Conclusion

We investigated numerous problematic issues and security challenges in SDN policy management and proposed a novel framework to facilitate robust policy management for SDN with respect to three planes in the SDN architecture. In the SDN application plane, we introduced an efficient policy composition mechanism along with a policy segmentation technique to address the issue created by the situation where one application with multiple modules jointly precess the same flow. We also discussed approaches to address the inter-application dependency issue in the SDN control plane. Lastly, we provided flow isolation technique to address the intra-table dependency issue in the SDN data plane. Our experimental results showed that our solution is effective and only introduce manageable performance overheads to enable robust SDN policy management.

As our future work, we will extend our framework to support dynamic policy updates. In that case, a more sophisticated composition mechanism should be designed to consider both policy revocation and recomposition situations. In addition, since our current framework only deals with policy management challenges with respect to one SDN controller, we would like to expand our solution to support comprehensive SDN policy management in terms of heterogeneous controllers.

Chapter 4

FLOWGUARD: ENABLE STATELESS NETWORK POLICY MANAGEMENT

4.1 Introduction

Over the past few years, Software-Defined Networking (SDN) has evolved from purely an idea [47, 46, 60] to a new paradigm that various networking vendors are not only embracing, but also pursuing as their model for future enterprise network management. As the first standard for SDN, OpenFlow [86] essentially separates the control plane and the data plane of a network device, and enables the network control to become directly programmable as well as the underlying infrastructure to be abstracted for network applications. With OpenFlow, only the data plane exists in the network device, and all control decisions are communicated to the device through a logically-centralized controller.

One primary goal of SDN is to enable various network applications, which are basically network services, to run on the controller to manage the network directly by configuring packet-handling mechanisms in underlying devices. Consequently, when enterprises adopt OpenFlow for their networks, it is virtually inevitable that legacy security applications such as firewalls and intrusion detection and prevention systems (IDS/IPS) have to be migrated to OpenFlow-based networks by re-designing and implementing them as compatible security applications. In this chapter, we focus on the challenges of designing and implementing a *reliable* firewall application for OpenFlow-based networks.

Firewalls are the most widely deployed security mechanism in most businesses and institutions. A conventional firewall sits on the border between a private network and the public Internet, and examine all incoming and outgoing packets to defend against attacks and unauthorized access. However, one key assumption under this traditional model is that all insiders of the protected network are trusted, since internal traffic is not seen and cannot be filtered by the firewall [68]. That assumption has been invalid for a long time, because insiders could easily launch attacks on others inside the network by circumventing security mechanisms [104]. With OpenFlow, such a problem could be potentially alleviated, since OpenFlow offers a deeper level of control granularity via placing enforcement points in any entries of traffic flows in a network.

Unfortunately, OpenFlow also brings great challenges for designing firewall applications in emerging SDNs. First, OpenFlow allows various `Set-Field` actions, which can rewrite the values of respective header fields in packets. Such a feature can significantly increase the usefulness of an OpenFlow implementation. For example, a load balancer application may need to dynamically change flow paths and destinations. However, *malicious* OpenFlow applications could also leverage this feature to strategically enable flow rules that would evade security mechanisms [1]. Second, in an OpenFlow network, network states are dynamically updated and configurations are frequently changed. Thus, simply checking policy violations against *new* traffic flows in a firewall application is not effective since security violations induced by other changes of network states and configurations–such as updating flow entries and firewall rules–should be examined as well. Last but not least, when a security violation is detected, firewall applications cannot plainly reject the new flow rule(s) or remove resident flow rule(s) that causes the violation. In OpenFlow, multiple traffic flows may match the same rule. Also, OpenFlow allows using wildcard rules to define a flow. In these cases, if only *partial* packets matching a rule violate the firewall policy, eliminating the rule may drop legal traffic which in turn could encumber the

[1]We further articulate such scenarios in Section 4.3.2.

availability and utility of network services.

Therefore, we observe that OpenFlow not only presents tremendous opportunities to networking, but also changes the way of defining network security appliances, including firewalls. As a robust security solution, we claim that an OpenFlow-based firewall application should have following properties: (1) monitoring network insiders; (2) tracking dynamic packet modifications; (3) examining various network state and configuration changes; (4) providing fine-grained violation resolution; and (5) enabling real-time network protection.

An exemplar firewall application based on OpenFlow has been introduced in Floodlight [11], a popular open SDN controller, which enforces security rules against traffic flows by monitoring all packet-in behaviors in a network. Nevertheless, this preliminary implementation only inspects a traffic flow at its *ingress* switch and lacks a capability to actively monitor packet modifications. In other words, once a flow passes the ingress switch, modified packets cannot be further inspected by the firewall. Also, it can only examine violations when a *new* flow comes in the network, but cannot check any other network updates.

In this chapter, we propose FlowGuard, a new firewall application, which is designed to facilitate not only accurate detection but also effective resolution of firewall policy violations, and support network-wide access control in dynamic OpenFlow networks. FlowGuard detects violations by examining *flow path space* against *firewall authorization space*. The violation detection approach in FlowGuard is capable of tracking flow paths in the entire network and checking rule dependencies [74, 122] in both flow tables and firewall policies. Besides, FlowGuard can determine violations dynamically when network states or configurations are changed. In addition, we introduce a flexible violation resolution framework in FlowGuard to enable a fine-grained violation resolution with the help of four resolution strategies, namely *dependency breaking*, *update rejecting*, *flow removing*, and *packet blocking*, considering diverse update situations in both flow entries and firewall rules. In order to ensure real-time response in FlowGuard, we also address several optimization considerations in the FlowGuard design.

The major contributions of this chapter are summarized as follows:

- We present security challenges and design requirements in building a firewall application for OpenFlow networks with respect to both *packet modifications* and *rule dependencies* in flow tables and firewall policies.
- We propose a systematic solution for designing an OpenFlow-based firewall application that enables network-wide access control in dynamic OpenFlow networks. Our design addresses challenges created by the *inter-reaction* of flow path and firewall authorization space, and facilitates not only accurate detection but also effective resolution of firewall policy violations in OpenFlow networks.
- We provide a prototype implementation of FlowGuard in an open SDN controller. We evaluate FlowGuard using a real-world network topology and an emulated OpenFlow network. Our experimental results show that FlowGuard has low performance overhead to enable real-time violation detection and resolution.

This chapter is organized as follows. We overview related work in Section 6.8. Section 4.3 overviews the security challenges and design requirements in constructing an OpenFlow-based firewall application. Section 4.4 presents the design of FlowGuard in detail. We address the implementation and the evaluation of FlowGuard in Section 5.5. Section 6.9 describes several important issues and our future work. We conclude this chapter in Section 6.10.

4.2 Related Work

Several recent efforts have been devoted to address various security challenges, such as scanning attack prevention [69, 87], DDoS attack detection [42], vulnerability assessment [39, 78], and saturation attack mitigation [110], in SDNs. Differentiating from those work, our work focuses on exploring how to build reliable firewalls for SDNs.

Floodlight contains a firewall application [11] where each packet-in behavior triggered by the first packet of a traffic flow is matched against the set of existing firewall rules that allow or deny a flow at its ingress switch. However, such a primitive implementation of OpenFlow-based firewall application suffers from a couple of limitations as discussed before. Pyretic [89] was recently introduced as a higher-level language in the Frenetic Project [57] that allows SDN programmers to write modular network applications, including firewall application. Pyretic's sequential composition operators could potentially resolve *direct* policy conflicts by compiling conflicting policies into a prioritized rule set. However, lacking a flow tracking capability [54], Pyretic cannot discover and resolve *indirect* security violations caused by dynamic packet modifications. FortNOX [95] was proposed as a software extension aiming to provide security constraint enforcement for OpenFlow controllers, being able to identify *indirect* security violations. FortNOX was then used as a security enforcement kernel for FRESCO [109], an OpenFlow security application development framework. However, we cannot directly adopt FortNOX approach to design our firewall application by virtue of several reasons. On one hand, the rule conflict analysis algorithm provided by FortNOX records rule relations in alias sets, which are unable to accurately track all flows. In particular, the conflict detection algorithm in FortNOX only conducts *pairwise* conflict analysis between new flow rule(s) and each single security constraint without considering *rule dependencies* within flow tables [74, 75] and among security constraints (represented as a firewall policy in our approach) [63, 122]. On the other hand, when FortNOX detects a security violation caused by new rule(s) installed by a non-security application, it simply rejects the rule(s) without offering a fine-grained violation resolution.

A couple of verification tools [35, 74, 73, 76, 85] for checking network invariants and policy correctness in OpenFlow networks have been proposed. Anteater [85] detects violations of network invariants using a SAT solver through transferring the data-plane information to boolean expressions and converting network invariants into instances of SAT problem. FlowChecker [35] translates network policies into boolean expressions and uses Binary Decision Diagram (BDD) to model the network state for checking network invariants. However, both Anteater and FlowChecker are static in nature and could not scale well to dynamic changes in the network. VeriFlow [76] and NetPlumber [74] are capable of checking the compliance of network updates with specified invariants in real time. VeriFlow uses graph search techniques to verify network-wide invariants and deals with dynamic changes. NetPlumber utilizes Header Space Analysis (HSA) [73] in an incremental manner to ensure real-time response for checking network policies through building a dependency graph. Even though these tools can be potentially used to detect firewall policy violations, they could not provide automatic and effective violation resolution. Also, they ignore rule dependencies within security constraints, such as firewall policies, for compliance checking.

Policy verification tools discussed above are able to check network reachability and potentially utilized for tracking flow paths in OpenFlow networks. However, Anteater [85] and FlowChecker [35] are indeed offline systems and cannot be applied for real-time flow tracking. VeriFlow [75] can perform reachability checking in real time, but it does not support dynamic packet modifications. Another option for flow tracking would be FlowTags [54], which can additionally deal with dynamic transformations in the presence of legacy middleboxes (e.g., proxies). However, FlowTags needs to alter existing OpenFlow architecture. In this work, we leverage the mechanism introduced by NetPlumber [74] to track flow paths for firewall policy violation detection, since NetPlumber provides a couple of features that fit for our purpose, such as support for arbitrary header modifications, automatic rule dependency detection, and real-time response.

Numerous firewall algorithms and tools have been designed to assist system administrators in managing and analyzing firewall policies [36, 37, 38, 122]. Especially, some work has presented policy analysis tools with the goal of detecting firewall policy conflicts. Al-Shaer and Hamed [36] designed a tool called Firewall Policy Advisor to detect pairwise anomalies in firewall rules. Yuan et al. [122] presented FIREMAN, a toolkit to check for misconfigurations in firewall policies through static analysis. However, existing firewall policy analysis tools only detect policy conflicts *within* a firewall policy, but cannot be directly applied to deal with firewall policy violations against flow policies in dynamic OpenFlow networks.

4.3 Background Technologies and Challenges

Before introducing the design of FlowGuard, we first briefly introduce the concepts of flow policy and firewall policy in this section. We then review security challenges and design requirements that motivate the features of FlowGuard.

4.3.1 Overview of Flow Rules and Firewall Policies

Flow Policy: In an OpenFlow network, flow rules can be added into flow tables, both *reactively* (generating rules in response to the packets of new flows) and *proactively* (generating rules before packets arrive at the switches) [2]. In the first case, if a

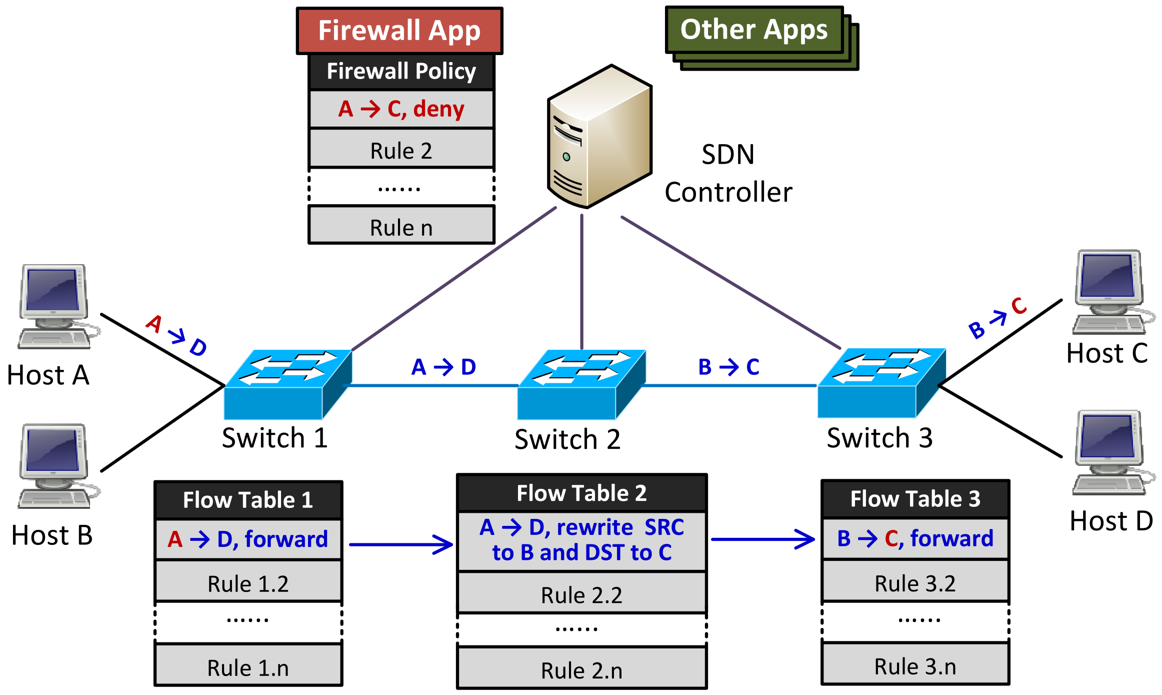


Figure 4.1: Firewall Is Bypassed by a Single Flow.

switch receives a packet for which no matching rule is found, it forwards the packet to the controller for the further inspection. The controller will determine whether that packet should be sent and can then install a new *flow policy*, which is a collection of rules to be installed at switches [2], for handling future packets of the same type. In the second case, the controller or applications are allowed to initiate some rules in the network devices before receiving flow packets.

Each flow rule specifies a *pattern* that matches on bits in the packet header, *actions* that are performed on matching packets to describe packet forwarding, packet modification or packet dropping, a *priority* that disambiguates among overlapping patterns, and *timeouts* that allow a switch to delete expired rules.

Firewall Policy: A firewall policy consists of a sequence of rules that define the actions performed on packets that satisfy certain conditions. The rules are specified

[2]In OpenFlow v1.0, each switch consists of one flow table. However, new versions of OpenFlow allow every switch contains multiple flow tables.

in the form of $\langle$*condition*, *action*$\rangle$. A *condition* in a rule is composed of a set of fields, which is typically specified in a 5-tuple format that contains *source IP*, *source port*, *destination IP*, *destination port*, and *protocol*, to identify a certain type of packets matched by this rule. The general *action* in a firewall rule is either "allow" or "deny".

In a firewall policy, multiple rules may overlap, which means one packet may match several rules. Moreover, multiple rules within one policy may conflict, implying that those rules, not only overlap each other but also yield different decisions. To resolve policy conflicts, a firewall typically implements a *first-match* resolution mechanism based on the order of rules. In this way, each packet processed by the firewall is mapped to the decision of the first rule that the packet matches.

4.3.2 Security Challenges

OpenFlow offers greater flexibility to networking. At the same time, its flexibility comes with security challenges. One such a challenge is introduced by the feature of *packet modification*, since OpenFlow permits various `Set-Field` actions that can dynamically change the packet headers. *Adversaries* could take advantage of this feature to circumvent network security mechanisms (e.g. firewalls). Another challenge may arise due to *rule dependencies* in flow tables and firewall policies. Flow rules may overlap each other in a flow table, indicating intra-table dependency of flow rules [74]. The rules in a firewall policy may overlap as well [63, 122]. These rule dependencies could be also leveraged by *malicious* OpenFlow applications and may cause severe network breaches. Next, we articulate two hypothetical scenarios to elaborate these challenges. To make our discussion concrete, we use an example network shown in Figures 4.1 and 4.2 with three switches, four hosts, and one SDN controller on which a simple firewall application (e.g. the Floodlight built-in firewall application) and several other applications are running.

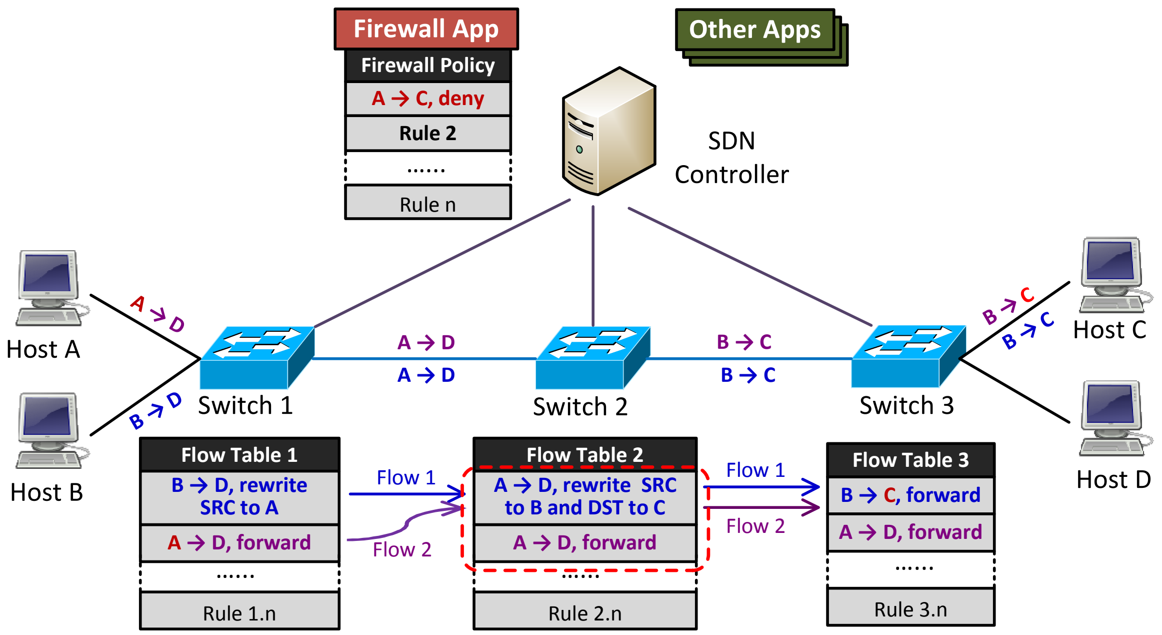


Figure 4.2: Firewall Is Bypassed Due to Rule Dependency.

Bypass Scenario 1: Figure 4.1 shows a simple bypass scenario, which is similar to the example given in [95]. The firewall application has a rule to deny network packets from *Host A* to *Host C*. [3] Suppose that another application running on the controller establishes a new flow by installing a flow policy, which contains three rules installed in flow tables of different switches, in the network. The first rule in the flow policy allows to simply forward packets from *Host A* to *Host D*. The second rule rewrites the source address (SRC) of a packet to *Host B* and the destination address (DST) of the packet to *Host C*. The last rule forwards packets from *Host B* to *Host C*. In this case, if *Host A* sends a packet to *Host D*, the packet will be finally delivered to *Host C*, which violates the firewall rule. However, if the firewall application only inspects the flow at its ingress switch (*Switch 1*) without tracking the entire flow in the network, such a violation cannot be observed by the firewall.

Bypass Scenario 2: A more complicated bypass scenario is illustrated in Figure 4.2.

[3]For brevity, we use the host name to represent the source and destination directly in the example rules.

The firewall application has the same policy. Assume that a policy for a flow (*Flow 1*) has been installed by an application in the network. Only the first rule in the flow policy is different from Scenario 1. That rule matches packets from *Host B* to *Host D* and modifies the source address of the matched packets to *Host A*. Since the *original* source of this flow is *Host B* and the *final* destination is *Host C*, the flow policy does not violate the firewall policy. Then, suppose another application installs a new policy, which contains three forwarding rules, for a flow (*Flow 2*) in the network. This policy is allowed by the firewall, since it directly forwards packets from *Host A* to *Host D* and does not violate the firewall policy either. We further assume that the policy for *Flow 2* are installed in the switches with a lower priority than the policy for *Flow 1* as shown in Figure 4.2. As we can notice, two rules belonging to different flow policies in *Switch 2* overlap each other, as they both match packets with *Host A* as the source address and *Host D* as the destination address. As a result, the packet belonging to *Flow 2* originally sent from *Host A* to *Host D* will be changed and eventually sent to *Host C*. This situation causes a violation of the firewall policy. Thus, even though individual policies defined for different flows (such as *Flow 1* and *Flow 2*) do not violate the firewall policy, the *dependency relations* among them may induce violation(s).

The rule dependencies in firewall policies may also affect the presence of violations. For example, if we add a new rule, saying "$A, B \rightarrow C, allow$", before the current rule in the firewall policy. The new rule will overlap with the existing rule, because the packets sent from *Host A* to *Host C* can match both of them. However, the first-matched rule will take precedence in the firewall. Thus, the new rule will shadow the existing rule [122]. As a result, no violation arises when applying such a new firewall policy to both Scenario 1 and Scenario 2.

The built-in firewall application in Floodlight and Pyretic's composition operators could not detect and resolve both bypass scenarios discussed above, since they are unable to monitor dynamic packet modifications. FortNOX has a limitation in identifying the violation introduced in the second scenario, as the rule conflict analysis algorithm in FortNOX ignores *rule dependencies* in flow tables and firewall policies.

4.3.3 Design Requirements

Our goal is to design a reliable firewall application that enables effective and efficient detection and resolution of firewall policy violations in dynamic OpenFlow networks. Consequently, to achieve our goal, we seek a solution that fulfills following design requirements to balance network protection and system performance.

1. *Accuracy.* The firewall application should precisely detect violations caused by traffic modifications, as well as rule dependencies in both flow tables and firewall policies. Also, the identified violations should be effectively resolved with respect to different violation situations, such as *partial* or *entire* violations. [4]
2. *Flexibility.* The firewall application should have the capability to inspect any network state and configuration updates, which may potentially incur firewall policy violations. In addition, flexible resolution strategies should be provided to deal with fine-grained violation resolutions.
3. *Efficiency.* The firewall application needs to continuously work in a timely fashion. Also, the state of an OpenFlow-based network generally evolves rapidly. Thus, it naturally requires that the response time of the firewall application should be fast enough and its performance overhead should not affect other network utilities.

[4] The detailed definitions are given in Section 18.

4.4 FlowGuard Design

In this section, we introduce our design of FlowGuard that is based on the proposed requirements. We focus on two main functions in FlowGuard: violation detection and violation resolution.

4.4.1 Violation Detection

In an OpenFlow network, the header fields of flow packets could be dynamically changed when the packets traverse the network. Thus, to support accurate violation detection, a firewall application needs to check violations at the ingress switch of each flow. It should also track the flow path and then clearly identify both the *original* source and *final* destination of each flow in the network. Next, we will first introduce flow path classifications and then articulate our violation detection method.

Flow Path Classification

A flow path is a forwarding path where one or multiple flows can pass through in the network. A flow path, which contains a sequence of (switch, rule) pairs, can be denoted by:

$$(s_1, r_1) \to \ldots \to (s_{n-1}, r_{n-1}) \to (s_n, r_n).$$

OpenFlow supports two kinds of flow rules, *microflow* rules and *wildcard* rules. A microflow rule is a rule in which all header fields that a packet can exactly match. In contrast, wildcard rules can match a large group of packets, such as all packets with source IP address matching the prefix 1.*.*.*, which represents an IP address range from 1.0.0.0 to 1.255.255.255. Therefore, we classify flow paths as two categories: *microflow path* and *wildcard-flow path.* In a microflow path, it contains at least one microflow rule, but the rules in a wildcard-flow path are all wildcard rules.

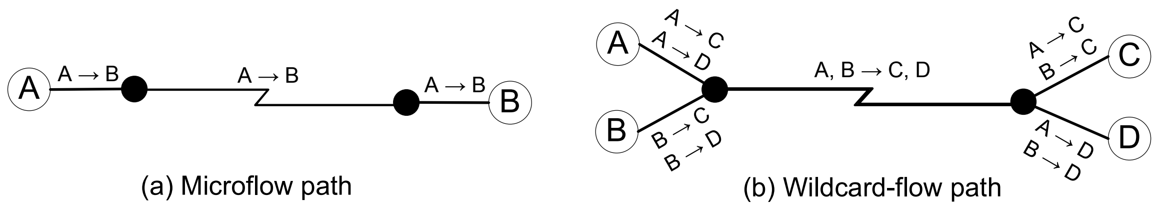


Figure 4.3: Examples of Microflow Path and Wildcard-flow Path.

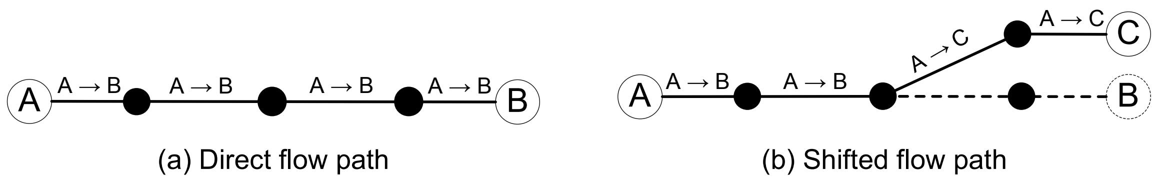


Figure 4.4: Examples of Direct Flow Path and Shifted Flow Path.

Figure 4.3 (a) shows an example of a microflow path where only packets from *Host A* to *Host B* can be sent. In Figure 4.3 (b), a wildcard-flow path is illustrated in such a way that *Host A* and *Host B* can send packets to *Host C* and *Host D*.

Since the packet headers of a flow may be modified when it passes through a path in the network, we further divide flow paths into two other categories: *direct flow path* and *shifted flow path*. In a direct flow path, all rules only perform "forward" action to the matched packets. In a shifted (or indirect) flow path, at least one rule enforces `Set-Field` action(s) to the matched packets. Figure 4.4 shows two examples for these paths. In the direct flow path shown in Figure 4.4 (a), *Host A* sends packets to *Host B* without any changes. However, in the shifted path depicted in Figure 4.4 (b), the destination of packets sent by *Host A* is changed from *Host B* to *Host C*.

Flow Path Space Analysis

Flow Tracking: To support network-wide access control in an OpenFlow network, a firewall application needs to figure out both the *original* source address and *final* destination address of each flow in the network through tracking its flow path. Accordingly, we need an effective flow tracking mechanism to identify flow paths. Several existing network invariant verification tools [74, 76] could check network reachability in real time and be potentially used to help find flow paths in OpenFlow networks. We currently leverage NetPlumber [74] as a baseline for building our flow tracking mechanism, because it offers a couple of features that can fulfill our design requirements: (1) Building on HSA [73], NetPlumber uses a geometric model (*Header Space*) of packet processing to provide a uniform and protocol-independent model of the network; (2) NetPlumber models networking boxes using a switch transfer function, which can transform a received header to a set of packet headers arbitrarily, supporting dynamic packets modifications; and (3) NetPlumber constructs a plumbing graph, which represents all next-hop dependencies and intra-table dependencies of rules. Through such a plumbing graph, all flow paths including both *direct* and *shifted* flow paths in the network can be automatically captured.

In a direct flow path, when the flow packets pass through it, the packet headers keep the same. For checking the firewall policy violations, it is not necessary to track this kind of direct flow path and violations can be simply identified in its ingress switch. Therefore, we introduce a concept of *Shifted Flow Path Graph* (SFPG), which is a sub-graph of the plumbing graph and contains all shifted flow paths and partial direct flow paths that have dependency relations with shifted flow paths. Therefore, our firewall application only needs to maintain and deal with an SFPG graph when monitoring an OpenFlow network, which could significantly reduce the overhead of flow tracking process.

Flow Path Space Calculation: We abstract fields, which are needed for checking firewall policy violations, from the pattern expression of a flow rule to represent the space of corresponding flow path. In addition, we reorganize these fields with a (*source address*, *destination address*) pair, denoted as $[P^s, P^d]$, to specify a *flow path space*.

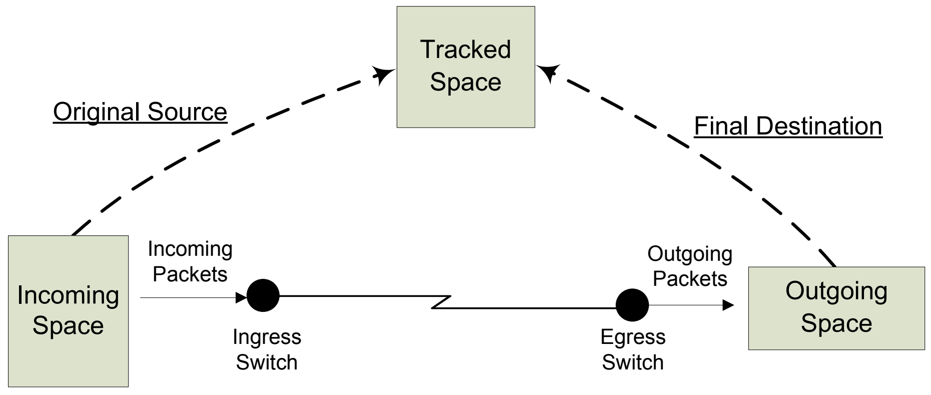
In the context of IP 5-tuple sense, the source address P^s consists of bit values from three fields, *source IP*, *source port*, and *protocol* of the flow rule. The destination address P^d contains bit values from two fields, *destination IP* and *destination port* of the flow rule. Then, we additionally define three kinds of spaces for representing a flow path space:

1. *Incoming Space* (S_i^P): It represents *original* header spaces of packets that can pass through the flow path, denoted as $[P_i^s, P_i^d]$.

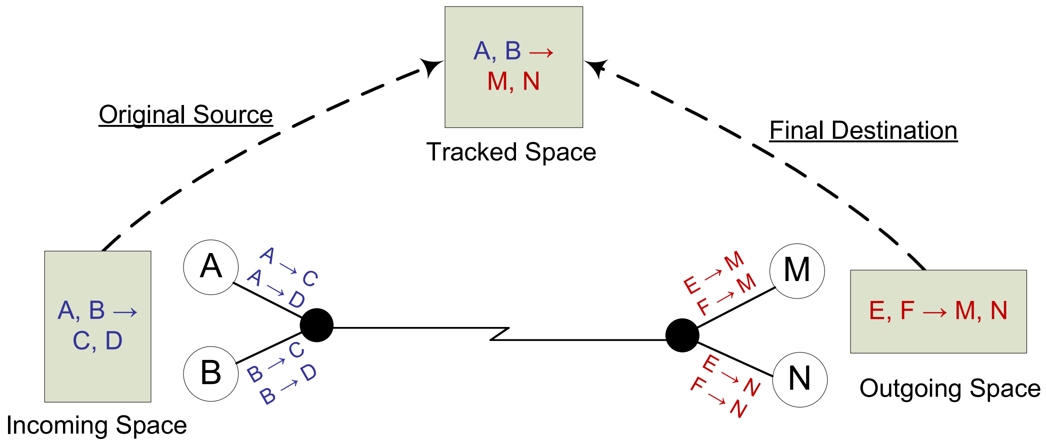
2. *Outgoing Space* (S_o^P): It represents *final* header spaces of packets after the packets pass through the flow path, denoted as $[P_o^s, P_o^d]$.

3. *Tracked Space* (S_t^P): This space represents *original source* address and *final destination* address of header spaces of packets that can pass through the flow path. Thus, it is a combination of the source address of the incoming space (P_i^s) and the destination address of outgoing space (P_o^d), denoted as $[P_i^s, P_o^d]$.

Figure 4.5 (a) depicts the relationships of three types of flow path space. As we can see, the *incoming space* of a flow path is calculated from the header spaces of incoming packets of the flow. The *outgoing space* of the flow path is computed from the header spaces of outgoing packets of the flow. Then, the *tracked space* of the flow path is derived from the source address of the incoming space and the destination address of outgoing space. An example is given in Figure 4.5 (b), which illustrates the space representation of a *wildcard shifted* flow path. The incoming space of this flow path, $[(A, B), (C, D)]$, indicates that *Host A* and *Host B* can send packets to *Host C* and *Host D* through this flow path, while the outgoing space of the flow path, $[(E, F), (M, N)]$, presents that *Host E* and *Host F* can send packets to *Host M* and *Host N*. That is, any packets sent from *Host A* and *Host B* through this flow path



(a) Flow path space relation



(b) Example of flow path space

Figure 4.5: Flow Path Space Classification.

will be finally delivered to *Host M* or *Host N*. Thus, the tracked space of this path is composed of the source address from its incoming space and the destination address from its outgoing space, represented as $[(A, B), (M, N)]$.

Firewall Authorization Space Partition

In many cases, a system administrator may intentionally introduce certain overlaps in firewall rules knowing that only the first rule is important. In reality, this is a commonly used technique to exclude specific parts from a certain action, and the proper use of this technique could result in a fewer number of *compact* rules [122].

Algorithm 3: Partitioning firewall authorization space

```
Input: A set of rules, $R$.
Output: A set of allowed spaces, $S_a^F$; A set of denied spaces, $S_d^F$.
foreach $r \in R$ do
    $s_r \longleftarrow HeaderSpace(r)$;
    if $Action(r) = allow$ then
        foreach $s \in S_d^F$ do
            /* $s_r$ is overlapping with $s$ */
            $s_r \longleftarrow s_r \setminus s$;
            $S_a^F.Append(s_r)$;
        end
    end
    if $Action(r) = deny$ then
        foreach $s^{'} \in S_a^F$ do
            /* $s_r$ is overlapping with $s^{'}$ */
            $s_r \longleftarrow s_r \setminus s^{'}$;
            $S_d^F.Append(s_r)$;
        end
    end
end
return $S_a^F$, $S_d^F$;
```

Hence, for the purpose of accurately detecting firewall policy violations in OpenFlow networks, the dependency relations between “allow” rules and “deny” rules in the firewall policy should be decoupled.

We first introduce a concept of *Firewall Authorization Space*, which represents a collection of all packets either allowed or denied by the firewall rules. Then, we introduce an approach, which represents rules with *header space* and performs various set operations on rules, to convert a list of firewall rules into two *disjoint* authorization

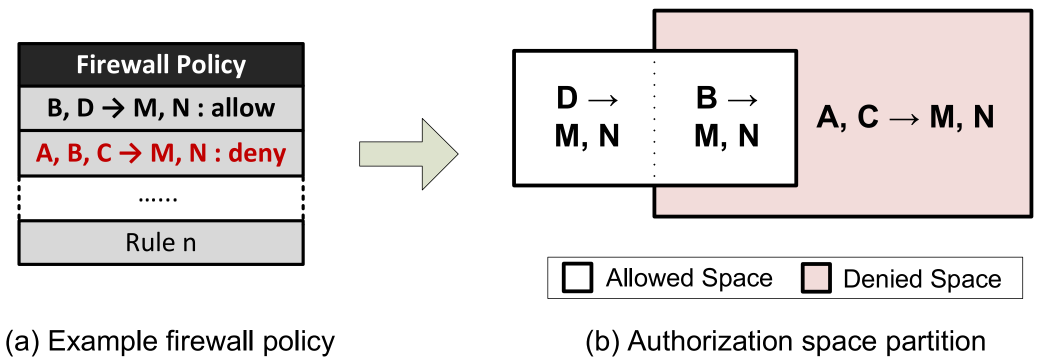


Figure 4.6: Example of Firewall Authorization Space.

sub-spaces, *denied authorization space* and *allowed authorization space*. Algorithm 3 shows the pseudocode of partitioning authorization space for a set of firewall rules R. This algorithm works by sequentially examining a header space s_r derived from a rule r and adding it to corresponding firewall authorization space sets, S_a^F or S_d^F, based on its type. For each r in R, if this rule is an "allow" rule, the header space s_r derived from this rule is compared with existing header spaces in the denied space set S_d^F. If the header space s_r is covered by any existing header spaces in S_d^F, the covered space(s) is removed from s_r and then the modified s_r is added into S_a^F. The similar process is applied to a "deny" rule. Therefore, one can utilize set operations to separate the overlapped spaces of a firewall policy into two disjoint authorization space sets S_a^F: $\{s_{a_1}^F, \dots, s_{a_{n-1}}^F, s_{a_n}^F\}$ and S_d^F: $\{s_{d_1}^F, \dots, s_{d_{m-1}}^F, s_{d_m}^F\}$. Formally, $s_{a_i}^F \cap s_{d_j}^F = \varnothing$, where $s_{a_i}^F \in S_a^F$, $s_{d_j}^F \in S_d^F$, $1 \leq i \leq n$, and $1 \leq j \leq m$. Note that it is unnecessary to eliminate overlapping header spaces within S_a^F and S_d^F, since those overlapping header spaces could not affect the results of violation detection and keeping them can potentially reduce the number of header spaces in each authorization space set.

An example of firewall authorization space partition is shown in Figure 4.6. For the purposes of brevity and understandability, we employ a two-dimensional geometric representation for each header space derived from firewall rules. Note that a firewall rule typically utilizes five fields to define the rule condition, thus a complete representation of header space should be multi-dimensional. In Figure 4.6 (b), we utilize colored rectangles to denote two kinds of authorization spaces: *allowed space* (white color) and *denied space* (pink color), respectively. In this example, there are an allowed space representing the first rule and a denied space depicting the second rule. Two spaces overlap when there are packets matching both rules (Figure 4.6 (b)). Applying Algorithm 3 to the example policy (Figure 4.6 (a)), the header space of the first rule is added into the allowed authorization space set. Then, the overlapped space is removed from the header space of the second rule, and the modified header space is added to the denied authorization space set.

Violation Discovery

Once the space of a flow path and the firewall authorization space of the firewall policy are calculated, we identify violations through checking the tracked space (S_t^P) of a flow path, which allows a flow to pass through the network, against the denied authorization space ($S_d^{F'}$) that is a union of all header spaces in the denied authorization space set (S_d^F) of the firewall policy. If these two spaces overlap each other, we call the overlapping space as the violated space ($S_v = S_t^P \cap S_d^{F'}$, denoted by $[P_v^s, P_v^d]$, where s and d denote source and destination addresses, respectively), which indicates a firewall policy violation. There are two kinds of violations.

- *Entire Violation*: If the denied authorization space $S_d^{F'}$ includes the whole tracked space S_t^P of the flow path, the violated space S_v indicates an entire violation. Formally, $S_t^P \subseteq S_d^{F'}$.
- *Partial Violation*: If the denied authorization space $S_d^{F'}$ partially includes the tracked space S_t^P of the flow path, the violated space S_v points out a partial violation. Formally, $S_t^P \nsubseteq S_d^{F'}$ and $S_t^P \cap S_d^{F'} \neq \varnothing$.

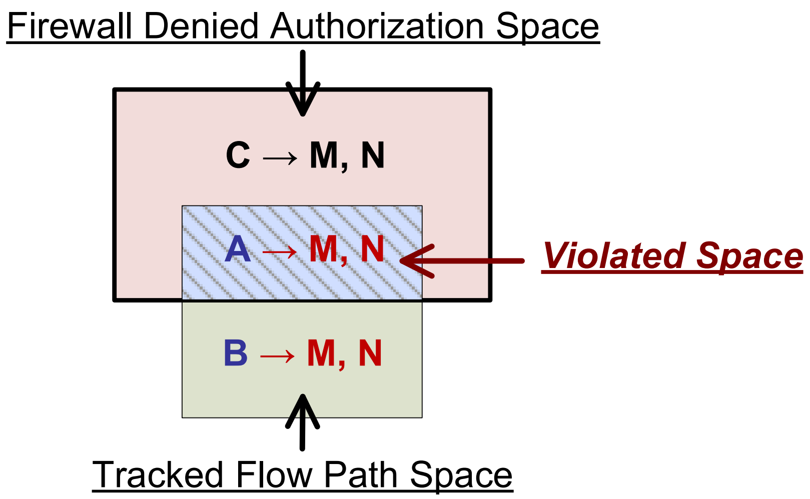


Figure 4.7: Violation Detection.

Figure 4.7 shows an example of our violation detection approach. The tracked space depicts that the original source of the flow is *Host A* and *Host B*, and the final destination of the flow is *Host M* and *Host N*. The firewall authorization space illustrates that all packets from *Host A* and *Host C* to *Host M* and *Host N* are denied. Thus, the violated space, which is depicted in inclined stripes in Figure 4.7, contains a partial tracked flow space represented with the original source of *Host A* and the final destinations of *Host M* and *Host N*. That is, all packets originally sent from *Host A* and finally arrived at *Host M* or *Host N* should be denied by the firewall.

4.4.2 Violation Resolution

An intuitive means for resolving a firewall policy violation is to simply disable the violated flow policy. That is, for a new flow policy, the request for installing this policy is rejected, if the firewall application detects this policy is in violation of the firewall policy. Regarding existing flow policies that violate the firewall policy, they are removed from the network devices directly. However, such a solution have several drawbacks. First, a flow policy may only *partially* violate the firewall policy as we discussed in Section 18. In this case, rejecting/removing the flow policy may affect

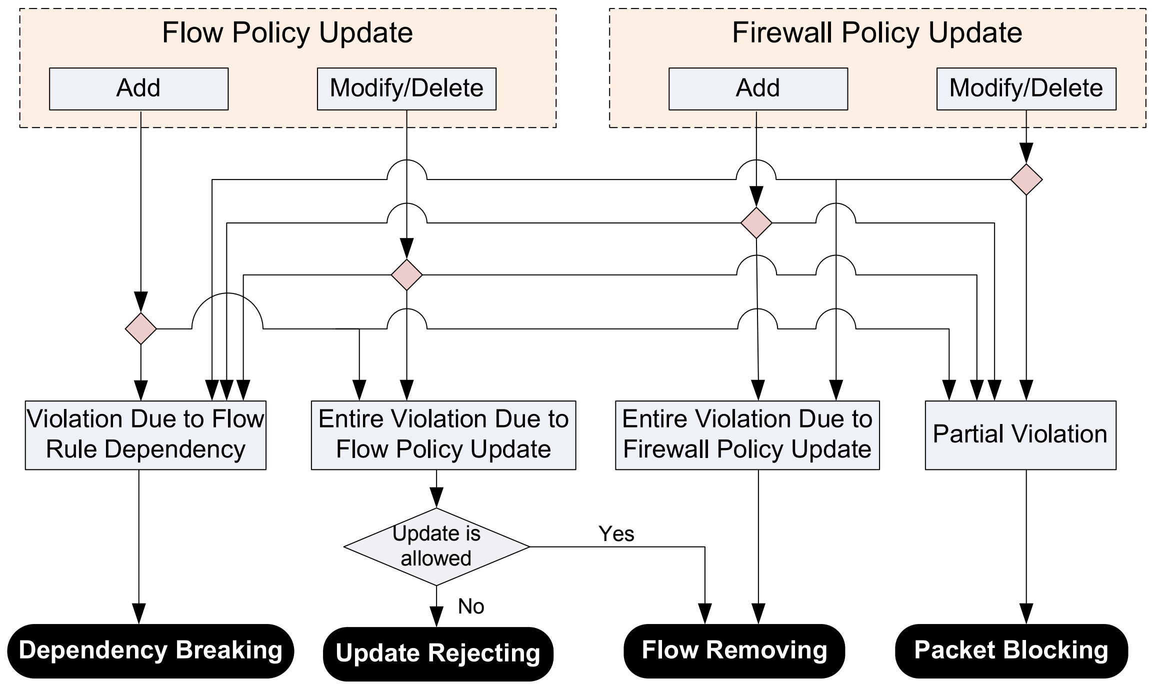


Figure 4.8: FlowGuard Violation Resolution Framework.

the utility of network services. Second, a rule in a flow policy may have dependency relations with the rules of other flow policies. Deleting a rule in a violated policy may impact other flow policies and even create new violation(s). Obviously, it is necessary to seek a systematic solution to enable a flexible and effective violation resolution. To this end, we introduce a violation resolution framework, as depicted in Figure 4.8, which demonstrates how FlowGuard adopts four violation resolution strategies to resolve different firewall policy violations in terms of various update operations on both flow policies and firewall policies in OpenFlow networks. We next discuss such a framework based on those resolution strategies.

Dependency Breaking

Situation: A new flow policy is being added into the network switches and this single flow policy does not violate the firewall policy. However, the rules in this new flow policy overlap with the rules of other flow policies when it is installed in the switches following the routing path calculated by the controller. Such a situation causes rule dependencies among flow policies. In addition, like the Scenario 2 demonstrated in Section 4.3.2, these rule dependencies cause new firewall policy violation(s). This kind of violation can also be incurred by other changes of network states, such as modifying flow entries and updating firewall rules.

Solution: Since rule dependencies among different flow policies could cause unexpected changes in packet headers of flows and also lead to new firewall policy violations, an approach for these issues is to break the dependencies among flow policies. Then, we can guarantee that when the packets of a flow traverse the network, they are precisely processed by the policy defining such a flow.

Inspired by the approach discussed in [101], which utilizes tags to distinguish packets belonging to different versions of policies for ensuring consistent network updates, we can use tags to break the rule dependencies as well. In this mechanism, the new flow policy is preprocessed by adding a tag to differentiate the match pattern with other policies. The rule of the policy in the ingress switch will take additional action on the packets to stamp them with the same tag. As the packets leave the network, in the egress switch, the corresponding rule of the policy will strip the tag from the packets.

Update Rejecting

Situation: There are three possible cases that can apply this strategy: (1) when adding a new flow policy, corresponding flow path is detected as a violation of the firewall policy and the violation is an *entire* violation; (2) changing a rule induces new *entire* violation(s); and (3) deleting a rule causes new *entire* violation(s), since some rules of other flows have dependency relations with this rule.

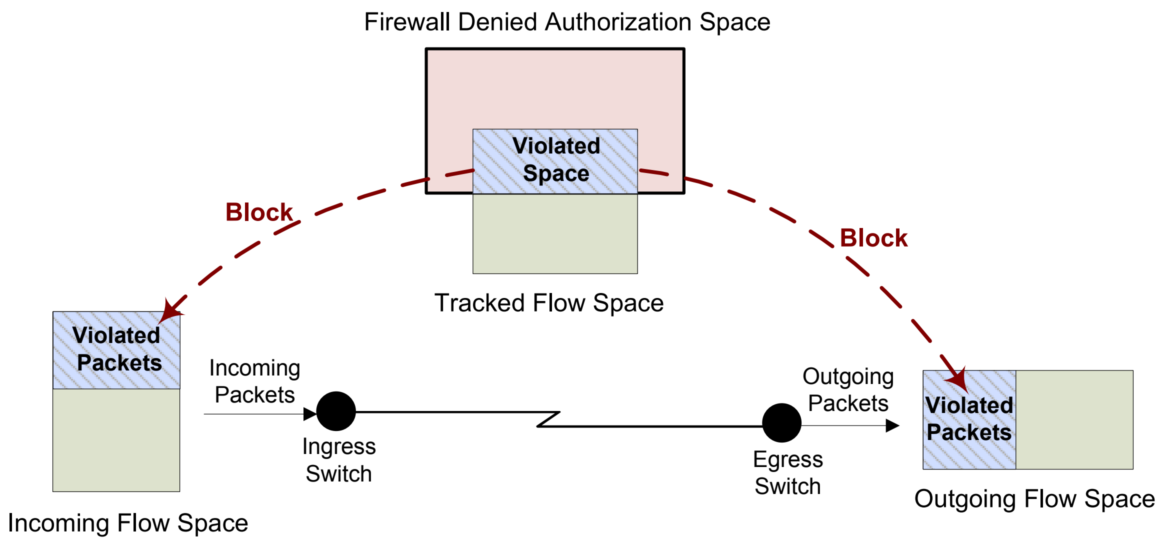


Figure 4.9: Violation Resolution Through Packet Blocking.

Solution: Applying this strategy, the update operation is rejected directly. Note that, this strategy may not be always applied for cases (2) and (3), since a change or delete operation on a rule may be mandatory depending on the privileges of the operator. [5]

Flow Removing

Situation: Two cases can apply this strategy: (1) when updating (adding, changing, or deleting) a rule(s) in the firewall policy, the firewall application examines the current network state applying the updated rule(s) and detect new entire violation(s); and (2) a change or delete operation on a rule is allowed, even though it causes entire violation(s).

Solution: Using this strategy, all rules associated with a flow path, which entirely violates the firewall policy, are removed from the network switches.

[5]A permission system for OpenFlow controller like the proposal discussed in [121] is required for deciding the operator's privileges.

Table 4.1: Detection and Resolution Elapsed Time in *ms* for Different Resolution Strategies.

Resolution Method	**Example Topology**		**Stanford w/o rules**		**Stanford w/ rules**	
	Detection	Resolution	Detection	Resolution	Detection	Resolution
Dependency Break	2.12	10.11	5.84	10.24	6.93	11.39
Update Rejecting	2.73	1.42	5.13	1.50	7.09	4.83
Flow Removing	2.62	2.61	5.86	2.80	7.85	16.27
Packet Blocking	2.36	5.34	5.74	5.21	6.34	5.25

Packet Blocking

Situation: For any *partial* violation detected by the firewall application, this strategy can be applied.

Solution: There may exist two ways to block packets of a flow: (1) if the flow is a new flow, the firewall application only needs to block it in the ingress switch of the flow; and (2) if the flow is an old flow, the firewall application needs to block the packets in both ingress and egress switches. In such a case, blocking packets in the ingress switch can prevent any new packets of the violated flow entering the network, while blocking packets in the egress switch can prevent any *inflight* packets of the violated flow from going through the network.

In order to block packets, the firewall application needs to install new blocking rules in the ingress and/or egress switches. As shown in Figure 4.9, the blocking rules can be derived from the violated space (S_v: $[P_v^s, P_v^d]$). The header space of the blocking rule for the ingress switch is a combination of the source address of the violated space (S_v) and the destination address of the incoming space (S_i^P), denoted as $[P_v^s, P_i^d]$. The header space of the blocking rule for the egress switch is combined from the source address of the outgoing space (S_o^P) and the destination address of the violated space (S_v), denoted as $[P_o^s, P_v^d]$.

4.4.3 Optimization Considerations

Since an OpenFlow firewall application must perform the violation detection and resolution in real time, several optimization mechanisms should be considered.

Incremental Checking: Building on NetPlumber, the flow track mechanism in FlowGuard is capable of performing incremental checks when updating flow policy. In addition, FlowGuard enables incremental checks for the firewall policy as well. Instead of recomputing an entire firewall authorization space each time whenever the firewall policy changes, FlowGuard only incrementally calculates the header spaces that are affected by these changes.

Maintaining Partial Flow Graph: As we have discussed in Section 4.4.1, FlowGuard only needs to maintain an SFPG graph that is a sub-graph of the plumbing graph. In addition, FlowGuard can check the source address in the incoming space of each shifted flow path against the source address of head spaces in its denied space. If these two source addresses do not overlap each other, FlowGuard can guarantee that the shifted flow path will not violate its policy without tracking the flow path. Thus, that shifted flow path can be removed from the SFPG graph.

4.5 Implementation and Evaluation

We have implemented our FlowGuard application on top of Floodlight. FlowGuard adopts NetPlumber data structure [13] for building header objects and computing intra-table dependencies. Thus, FlowGuard adds several new classes, such as *HeaderObject* and *RuleNode*, to implement bit-level representation of packet headers and rule dependency checking features. In addition, FlowGuard adds *listeners* to monitor two modules, *Static Flow Pusher* and *Memory Storage Source*, supported by Floodlight controller to retrieve network topology information and flow rules in real time. FlowGuard retrieves flow rules using the *Static Flow Pusher* module to build/modify *RuleNodes*, so that FlowGuard sorts them by their priorities and computes intra-table dependencies. At the same time, FlowGuard obtains the information of network devices including attached switch ID and corresponding port number using the *Memory Storage Source* module, and utilizes it to understand the physical topology of a network.

FlowGuard combines topology information with flow rules installed in the network to build the flow graph for tracking flow paths. If the tracked spaces of flow paths overlap with firewall denied authorization space, FlowGuard analyzes the root cause of each violation and leverages a corresponding resolution strategy to resolve the identified violation as illustrated in Figure 4.8. At the same time, FlowGuard maintains updated flow rules and network topology information, so that it is able to re-propagate header objects at any associated switches to update flow paths. In addition, FlowGuard utilizes the Floodlight built-in firewall to generate new blocking rules and the *Static Flow Pusher* module to add/modify/delete flow rules for violation resolution with respect to different violation resolution strategies.

4.5.1 Experiment Design

All of our experiments were performed in Ubuntu 12.04 virtual machines, each of which has four processors and 8GB memory. We ran Mininet 2.0 [22] in one virtual machine to simulate the network topologies and used another virtual machine to run FlowGuard on top of Floodlight v0.90.

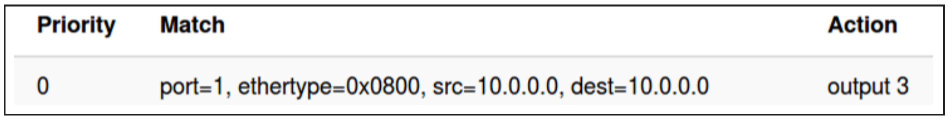
The experiments were carried out on two network topologies, one of which is the network demonstrated in Section 4.3.2. We instantiated the IP addresses of *Host A* to *Host D* from 10.0.0.1 to 10.0.0.4. The other topology is constructed based on the Stanford backbone network [74] that includes 14 operational zone Cisco routers, 10

Ethernet switches, and 2 backbone Cisco routers. By using this *real-world* network, we attempted to demonstrate the scalability of FlowGuard. The entire configuration of the Stanford backbone network was retrieved from [13]. We developed two parsers to convert the file formats in Stanford dataset to Floodlight's languages. The first parser reads Cisco routing table information and outputs Floodlight-acceptable flow table information. Overall, there are 8,908 flow entries in the network. The second parser accepts Cisco access control list (ACL) files as input and generates corresponding firewall rules for FlowGuard. There are the total of 1,206 *realistic* firewall rules used in the network.

4.5.2 FlowGuard *Violation Detection and Resolution*

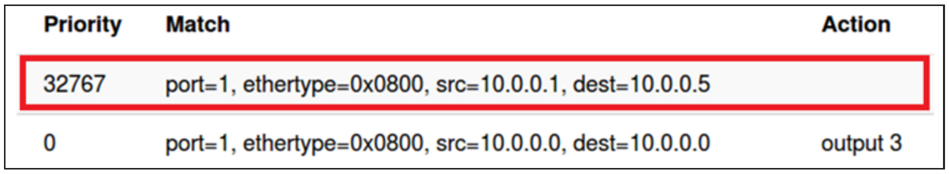
Dependency Breaking: To test the effectiveness of our dependency breaking strategy, we used the same topology as shown in *Bypass Scenario 2*. The flow tables of Switches 1, 2, and 3 before applying the resolution technique are shown in Figure 4.11 (a), (c), and (e). There was only one firewall rule specified in FlowGuard, which denied the communications from 10.0.0.1 to 10.0.0.3. As we discussed in Section 4.4, *Flow 1* and *Flow 2* in Figure 4.2 does not violate the firewall rule directly. However, the intra-table dependency occurred in Switch 2 introduces a potential firewall rule violation. In order to resolve this issue, FlowGuard removed the intra-table dependency by isolating *Flow 1* from *Flow 2* using a virtual LAN (VLAN) field. As depicted in Figures 4.11 (b), (d), and (f), the flow tables were updated by FlowGuard respectively. The action field of the first flow entry in Switch 1 has been updated to add a VLAN ID 100, and the corresponding flow entry in Switch 2 was modified to match the same VLAN ID. And finally Switch 3, which is the egress switch in this case, removed the VLAN ID to restore the original packet.

We carried out the same experiment again in the Stanford backbone network. As



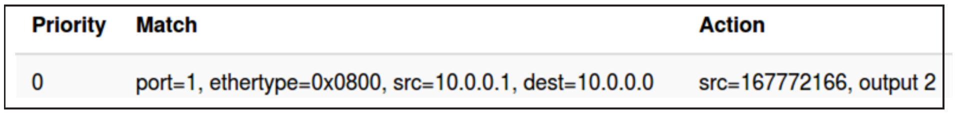
Priority	Match	Action
0	port=1, ethertype=0x0800, src=10.0.0.0, dest=10.0.0.0	output 3

(a) Switch 1 Flow Table before Resolution



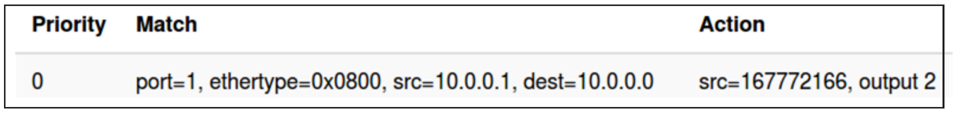
Priority	Match	Action
32767	port=1, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.5	
0	port=1, ethertype=0x0800, src=10.0.0.0, dest=10.0.0.0	output 3

(b) Switch 1 Flow Table after Resolution



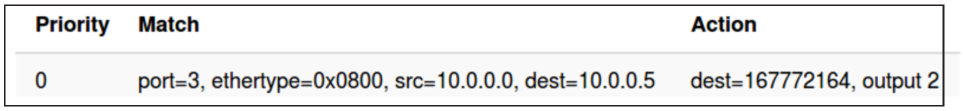
Priority	Match	Action
0	port=1, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.0	src=167772166, output 2

(c) Switch 2 Flow Table before Resolution



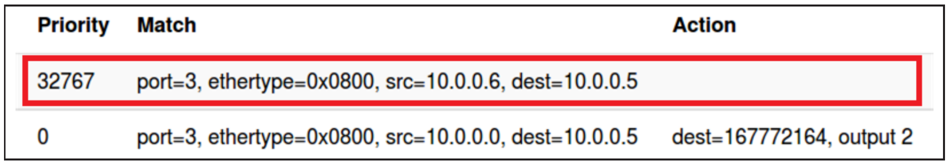
Priority	Match	Action
0	port=1, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.0	src=167772166, output 2

(d) Switch 2 Flow Table after Resolution



Priority	Match	Action
0	port=3, ethertype=0x0800, src=10.0.0.0, dest=10.0.0.5	dest=167772164, output 2

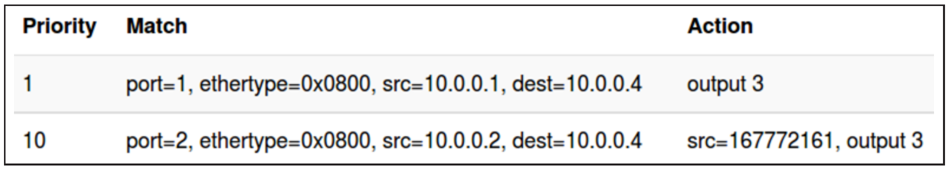
(e) Switch 3 Flow Table before Resolution



Priority	Match	Action
32767	port=3, ethertype=0x0800, src=10.0.0.6, dest=10.0.0.5	
0	port=3, ethertype=0x0800, src=10.0.0.0, dest=10.0.0.5	dest=167772164, output 2

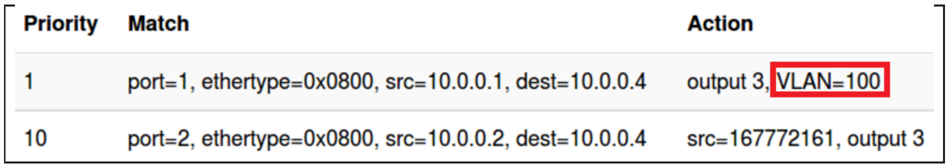
(f) Switch 3 Flow Table after Resolution

Figure 4.10: Flow Tables Before/After FlowGuard's Packet Blocking Strategy. Every Subfigure Is a Screen Shot Taken from the Floodlight Controller GUI.



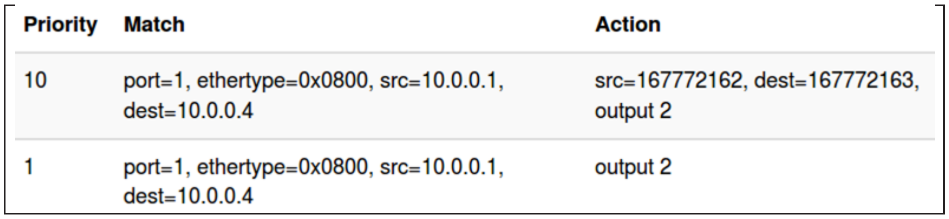
Priority	Match	Action
1	port=1, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.4	output 3
10	port=2, ethertype=0x0800, src=10.0.0.2, dest=10.0.0.4	src=167772161, output 3

(a) Switch 1 Flow Table before Resolution



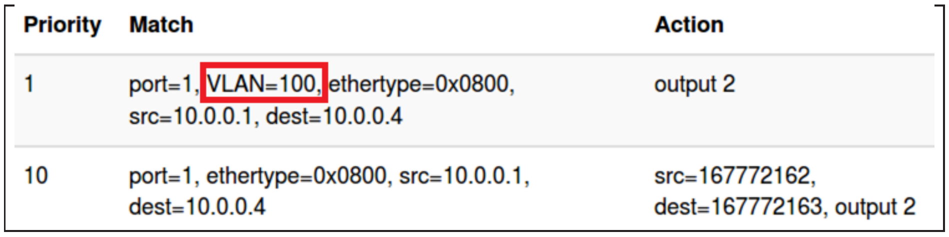
Priority	Match	Action
1	port=1, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.4	output 3, VLAN=100
10	port=2, ethertype=0x0800, src=10.0.0.2, dest=10.0.0.4	src=167772161, output 3

(b) Switch 1 Flow Table after Resolution



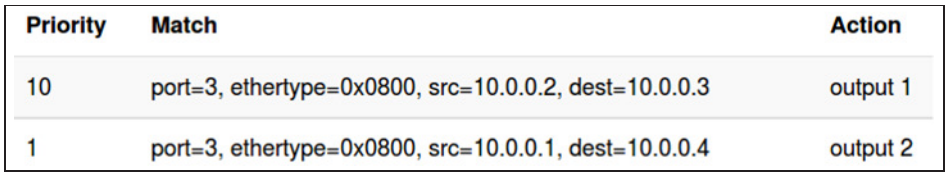
Priority	Match	Action
10	port=1, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.4	src=167772162, dest=167772163, output 2
1	port=1, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.4	output 2

(c) Switch 2 Flow Table before Resolution



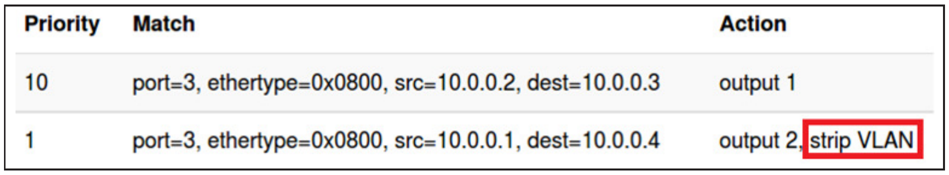
Priority	Match	Action
1	port=1, VLAN=100, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.4	output 2
10	port=1, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.4	src=167772162, dest=167772163, output 2

(d) Switch 2 Flow Table after Resolution



Priority	Match	Action
10	port=3, ethertype=0x0800, src=10.0.0.2, dest=10.0.0.3	output 1
1	port=3, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.4	output 2

(e) Switch 3 Flow Table before Resolution



Priority	Match	Action
10	port=3, ethertype=0x0800, src=10.0.0.2, dest=10.0.0.3	output 1
1	port=3, ethertype=0x0800, src=10.0.0.1, dest=10.0.0.4	output 2, strip VLAN

(f) Switch 3 Flow Table after Resolution

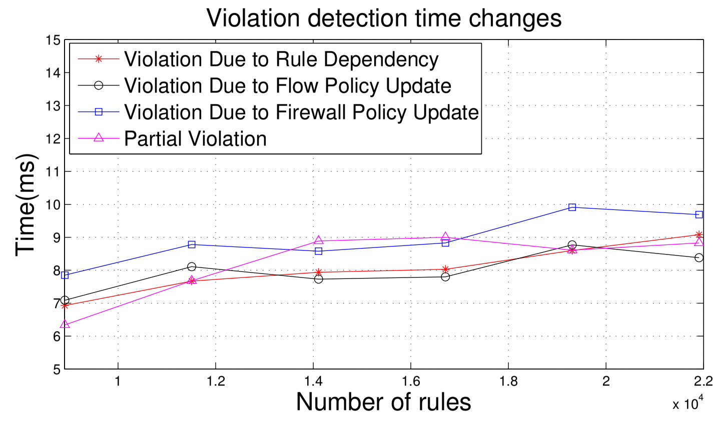
Figure 4.11: Flow Tables Before/After FlowGuard's Dependency Breaking Strategy. Every Subfigure Is a Screenshot Taken from the Floodlight Controller GUI.

shown in Table 4.1, the detection and resolution time with the example topology were 2.12 and 10.11 milliseconds (*ms*), respectively. And the resolution time with the Stanford topology was almost unchanged but the detection time has been slightly increased.

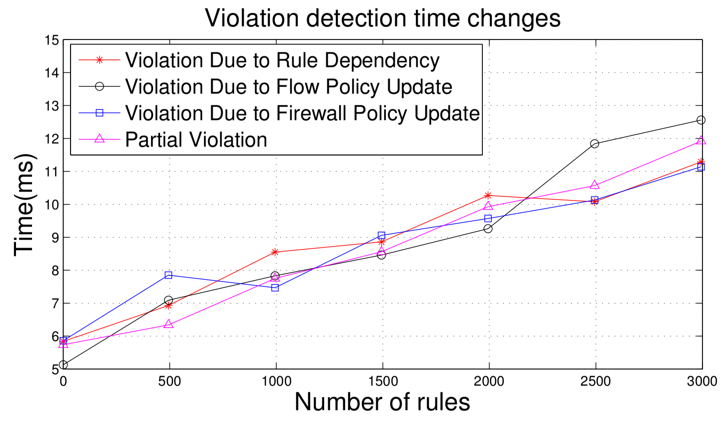
Update Rejecting: We used the scenario shown in *Bypass Scenario 1* to test this strategy. We first installed the flow entries in Switch 1 and Switch 3 as shown in Figure 4.1. As the green boxes in Figure 4.13 indicate, the installations were successful. Then we tried to add the flow entry shown in Flow Table 2 in Figure 4.1 to Switch 2. As the red box showing in Figure 4.13, FlowGuard decided to reject this flow table update request since it brought a firewall rule violation. As shown in Table 4.1, FlowGuard spent 2.73 *ms* to detect this violation and 1.42 *ms* for resolution with example topology. For the Stanford network with flow entries, FlowGuard took 7.09 *ms* for detection but only spent 4.83 *ms* for resolution since FlowGuard simply rejected to insert new rule.

Flow Removing: We utilized *Bypass Scenario 1* again to evaluate this strategy. We first set up all the flow entries shown in Figure 4.1 without running FlowGuard. As expected, the built-in Floodlight firewall could not identify any violation. Then, we enabled FlowGuard and specified the firewall rule as shown in Figure 4.1. As a result, FlowGuard detected the violation and removed corresponding flow entries that caused this violation in the three switches. As Table 4.1 shows, the detection took 2.62 $\sim$ 7.85 *ms* (slightly increasing) but the Stanford topology with flow entries took 16.27 *ms* to resolve the violation since FlowGuard inspected all flow entries to find out any missing flow entries which are associated with the removed flow.

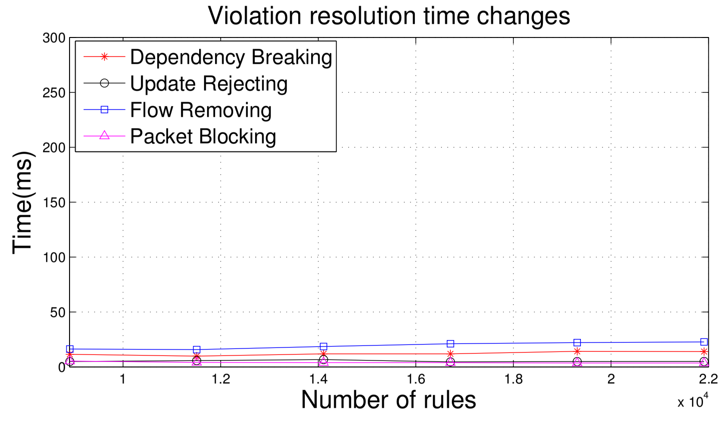
Packet Blocking: To set up a partial violation, we first installed a flow policy as shown in Figure 4.10. The flow entry in Switch 1 forwards every packet whose source



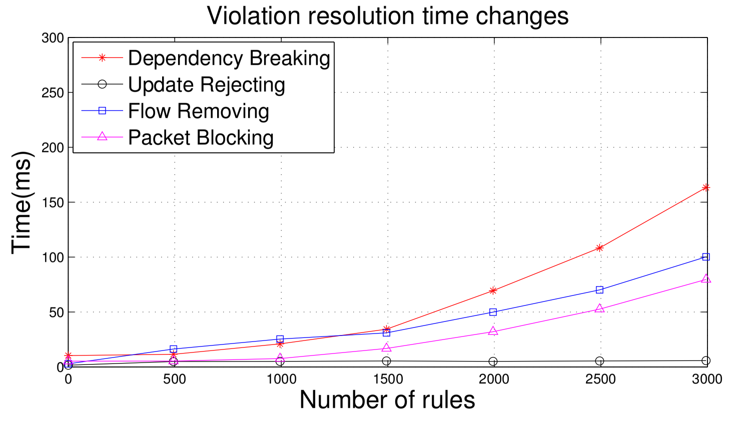
(a) Detection time changes in the first scenario



(b) Detection time changes in the second scenario

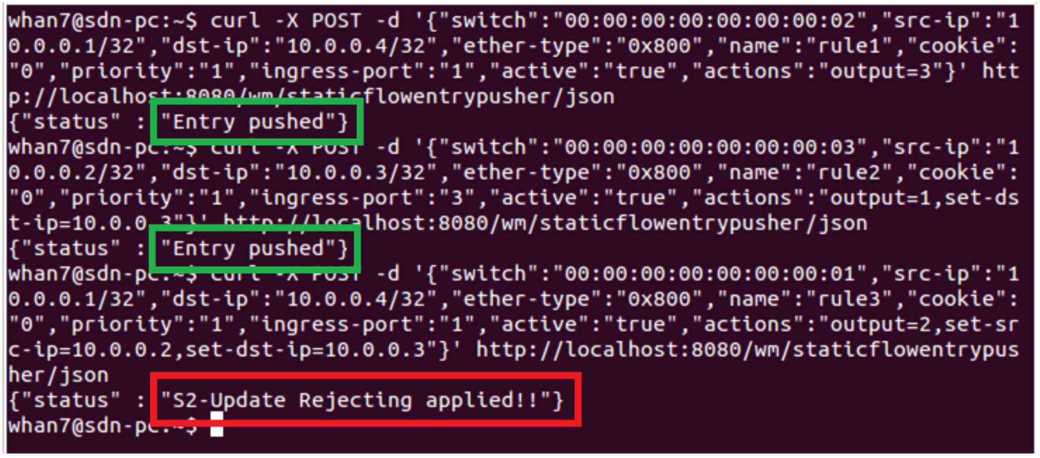


(c) Resolution time changes in the first scenario



(d) Resolution time changes in the second scenario

Figure 4.12: Detection and Resolution Time Changes Under Two Different Scenarios.



```
whan7@sdn-pc:~$ curl -X POST -d '{"switch":"00:00:00:00:00:00:00:02","src-ip":"1
0.0.0.1/32","dst-ip":"10.0.0.4/32","ether-type":"0x800","name":"rule1","cookie":
"0","priority":"1","ingress-port":"1","active":"true","actions":"output=3"}' htt
p://localhost:8080/wm/staticflowentrypusher/json
{"status" : "Entry pushed"}
whan7@sdn-pc:~$ curl -X POST -d '{"switch":"00:00:00:00:00:00:00:03","src-ip":"1
0.0.0.2/32","dst-ip":"10.0.0.3/32","ether-type":"0x800","name":"rule2","cookie":
"0","priority":"1","ingress-port":"3","active":"true","actions":"output=1,set-ds
t-ip=10.0.0.3"}' http://localhost:8080/wm/staticflowentrypusher/json
{"status" : "Entry pushed"}
whan7@sdn-pc:~$ curl -X POST -d '{"switch":"00:00:00:00:00:00:00:01","src-ip":"1
0.0.0.1/32","dst-ip":"10.0.0.4/32","ether-type":"0x800","name":"rule3","cookie":
"0","priority":"1","ingress-port":"1","active":"true","actions":"output=2,set-sr
c-ip=10.0.0.2,set-dst-ip=10.0.0.3"}' http://localhost:8080/wm/staticflowentrypus
her/json
{"status" : "S2-Update Rejecting applied!!"}
whan7@sdn-pc:~$
```

Figure 4.13: Screen Shot for Update Rejecting Strategy.

IP is in 10.0.0.0/24 and destination IP is in 10.0.0.0/24. The flow entry in Switch 2 modifies source IP to 10.0.0.6/32 when the incoming packet has source IP 10.0.0.1/32 and destination IP in 10.0.0.0/24. The flow entry in Switch 3 rewrites destination IP to 10.0.0.4/32 whenever the source IP is in 10.0.0.0/24 and the destination IP is 10.0.0.5/32. Then, we enabled FlowGuard and installed a firewall rule that blocks packets from 10.0.0.1 to 10.0.0.4. Since the installed flow policy would allow the connection between 10.0.0.1 to 10.0.0.4, the firewall rule is violated. FlowGuard detected such a partial violation, generated two additional flow entries, and installed them in the ingress switch and egress switch, respectively.

As shown in Figure 4.10 (b), FlowGuard installed a new flow entry in Switch 1 that blocked packets with source IP 10.0.0.1/32 and destination IP 10.0.0.5/32. Since the flow entry in Switch 2 rewrote the source IP of packets from 10.0.0.1/32 to 10.0.0.6/32, FlowGuard also installed a new flow entry in Switch 3 that dropped packets with source IP 10.0.0.6/32 and destination IP 10.0.0.5/32. In addition, FlowGuard installed a new firewall rule that denied the packets from 10.0.0.1 to 10.0.0.5. As Table 4.1 shows, the overhead of detection is similar to the other strategies, and it took 5.21 $\sim$ 5.34 ms for resolution.

4.5.3 Scalability Analysis of FlowGuard

Although we have performed our evaluation under two different topologies, we still want to examine the scalability of FlowGuard with respect to different sizes of flow rules. Therefore, we increased the number of flow rules based on the Stanford network topology to evaluate the scalability of FlowGuard. We checked detection and resolution time changes under two different scenarios. In the first scenario, by inserting $100 \sim 500$ additional flow rules in each switch, the Stanford network have $11k \sim 22k$ rules in total since it contains 26 switches. In the second scenario, we only increased the number of flow rules at the switches associated with the violated flow paths by adding $500 \sim 2,500$ flow rules at each relevant switch. As shown in Figure 4.12 (a) and Figure 4.12 (b), the violation detection time was increased linearly in accordance with the growing numbers of flow rules. The resolution time changes in the second scenario, depicted in Figures 4.12 (d), tell us that the dependency breaking strategy is the most heavy resolution mechanism among four resolution strategies. However, as shown in Figures 4.12 (c), FlowGuard spent less than 25 ms to resolve each violation in networks with large sizes of flow rules.

4.5.4 Performance Comparison with Floodlight Built-in Firewall

We also compared the performance of FlowGuard (FG) with the performance of the Floodlight built-in firewall (FW). We first measured the time FG and FW spent to initialize themselves in different network configurations. For an empty network where there is no network node, FG took 0.88 ms to finish initialization, while FW took 0.87 ms. For the Stanford network without any flow entries, FG took 3.21 ms, while FW spent 1.02 ms. For the Stanford network with all forwarding entries and ACL rules installed, FG spent 740.08 ms, while FW took 0.97 ms. This is because

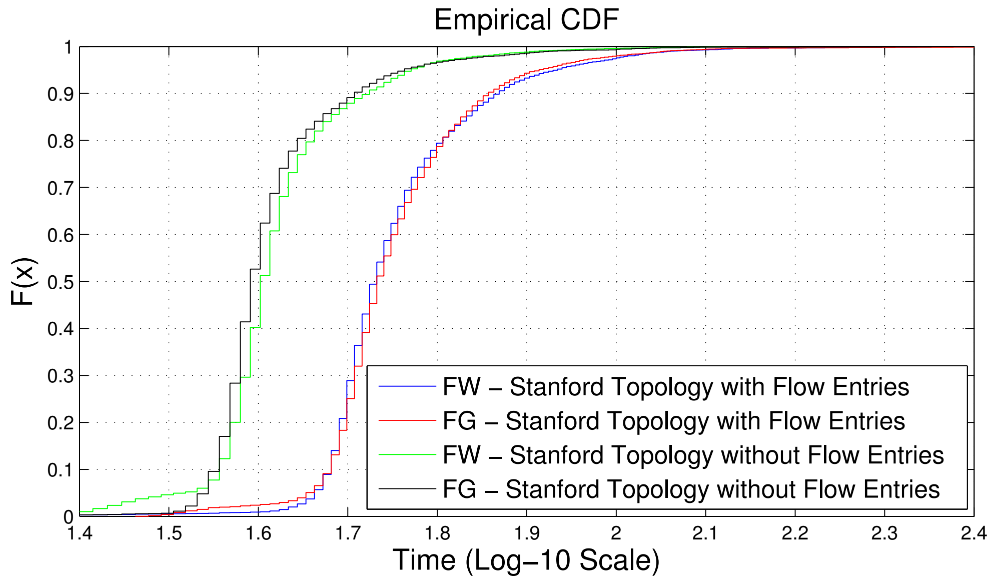


Figure 4.14: Firewall Rule Update Time in Microsecond.

FG needs to analyze many flow entries as well as firewall rules. As we can observe, even with a real world network, the initialization of FlowGuard is fast (less than one second).

We are also interested in how long it takes for FG and FW to update new firewall rules. To test this, we updated FG and FW with around 700 rules for each of which we recorded the time FG and FW used for processing. We draw an empirical cumulative distribution function (CDF) graph based on the results as shown in Figure 4.14. As the figure suggests, FG has almost the same update time as FW does, under the same network conditions. Most rules could be updated in less than 63 microseconds in the Stanford topology.

We also generated around 5K testing packets to measure per-packet inspection time for FG and FW. We used the Stanford topology with all ACL rules installed. As shown in Figure 4.15, the inspection time of 90% packets with flow entries was slower than 0.074 milliseconds, while without flow entries FG and FW spent less than 0.051 milliseconds to inspect 90% packets. As the results suggest, even though it took

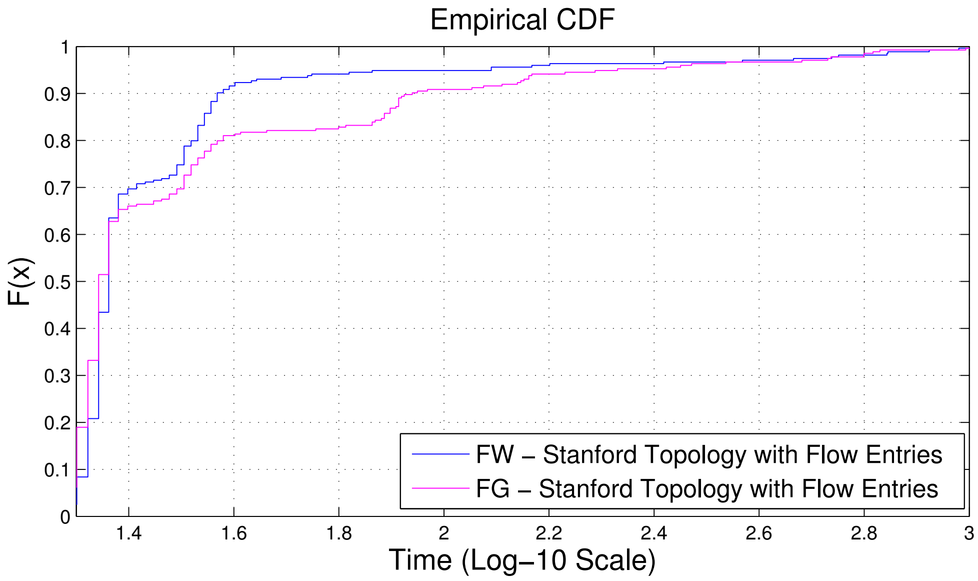


Figure 4.15: Per Packet Inspection Time in Microsecond.

longer for FG to inspect packets than FW, the processing speed was still very fast.

4.6 Discussion and Future Work

Security Enforcement Kernel: The design goal of FortNOX is to provide a security enforcement kernel (SEK) that can be integrated into OpenFlow controllers. Then, other OpenFlow-based security applications can rely on such an SEK to detect and resolve rule conflicts that may be introduced by non-security applications. FortNOX has been utilized to support FRESCO [109], an OpenFlow security application development framework. However, as we have discussed above, FortNOX has several limitations in rule conflict detection and resolution. In contrast, FlowGuard provides a new design that facilitates not only accurate conflict detection but also flexible and effective conflict resolution. Thus, we believe the solution provided by FlowGuard could be potentially utilized for building a more robust SEK for OpenFlow controllers.

Stateful Monitoring: Currently, OpenFlow only provides very limited access to packet-level information in the controller [112]. In addition, the OpenFlow forwarding plane is almost stateless and incapable to actively monitor flow status without the involvement of the controller [113]. Therefore, as our first step for designing an OpenFlow-based firewall application, we only implemented FlowGuard as a stateless firewall application, which could not perform stateful packet inspection in OpenFlow networks. However, we would explore how FlowGuard can be extended to support stateful packet inspection.

Flow Tracking: The flow tracking mechanism used in FlowGuard builds on NetPlumber, which has limitations for dealing with middleboxes with dynamic state [74]. FlowTags [54] was recently proposed to handle dynamic traffic modification in the presence of legacy middleboxes. However, FlowTags needs to extend current OpenFlow architecture to support flow tracking features. As our further work, we would like to study a more effective flow tracking solution for FlowGuard implementation.

Network Programming Language: The current design of FlowGuard is built on top of OpenFlow, which is defined at a low level of network abstraction. The Frenetic Project [57] introduces a family of languages providing reusable and high level abstractions for programming SDNs. In particular, Pyretic [89], which is one member of the Frenetic family, enables a program to combine multiple policies together using policy composition operators, potentially resolving partial policy conflicts including *direct* firewall policy violations. However, lacking a policy conflict detection mechanism in Pyretic, it is obviously inefficient to *always* compose the firewall policy with flow policies and install them into the network switches due to several reasons. First, a firewall policy may consist of over thousands of rules, but commodity SDN switches with limited TCAM space typically support only a few thousands of rules [117]. Second, if flow policies *entirely* violates the firewall policy, it is unnecessary to install those violated flow policies into the network switches. Therefore, we would study solutions that facilitate more secure and effective policy compositions in high level abstractions for building security applications in SDNs.

4.7 Conclusion

In this chapter, we have presented the design and implementation of a new OpenFlow-based firewall application, FlowGuard, for software-defined networks. FlowGuard provides an effective approach to detect firewall policy violations through examining the flow path space against the firewall authorization space. In addition, FlowGuard supports a flexible and fine-grained conflict resolution with respect to different update scenarios in flow entries and firewall rules. Our experimental results show that FlowGuard has the manageable performance overhead to enable real-time monitoring of software-defined networks.

Chapter 5

STATEMON: ENABLE STATEFUL NETWORK POLICY MANAGEMENT

5.1 Introduction

Over the past few years, Software-Defined Networks (SDNs) have evolved from purely an idea [47, 46, 60] to a new paradigm that several networking vendors are not only embracing, but also pursuing as their model for future enterprise network management. According to a recent report from Google, SDN-based network management helped them run their WAN at close to 100% utilization compared to other state-of-the-art network environments with about 30% to 40% network utilization [70].

As the first widely adopted standard for SDNs, OpenFlow [86] essentially separates the control plane and the data plane of a network device and enables the network control to become directly programmable as well as the underlying infrastructure to be abstracted for network applications. With OpenFlow, only the data plane exists in the network device, and all control decisions are conveyed to the device through a logically-centralized controller. In this way, OpenFlow can tremendously help administrators access and update configurations of network devices in a timely and convenient manner and provide this ease of control to SDN applications as well.

While the abstraction of a logically centralized controller, which is a core principle of SDNs is powerful, a fundamental limitation of OpenFlow is *the lack of capability to enable the maintenance of network connection states inside both the controller and switches.* First, OpenFlow-enabled switches only forward the first packet of a new flow to the controller so that the controller can make a centralized routing decision. Because the controller is unaware of subsequent packets of the flow, including those that change the state of a network connection (e.g., TCP FIN), the controller has no knowledge of the state of the connections in its network. Second, OpenFlow-enabled switches are incapable of monitoring network connection states as well. The "match-action" abstraction of OpenFlow heavily relies on L2/L3 fields (e.g., src_ip and dst_ip) and the limited L4 fields (only src_port and dst_port), yet essential information for identifying and maintaining the state of connections is contained in other L4 fields, such as TCP flags and TCP sequence and acknowledgment numbers.

The lack of knowledge of network connection states in SDNs brings significant challenges in building *state-aware* access control management schemes [90]. In particular, some critical security services, such as stateful network firewalls that perform network-wide access control, cannot be realized in SDNs. A stateful network firewall, which is a key network access control service in a traditional network environment [59, 64, 102] and requires state-awareness, keeps track of the states of connections in the network and makes a decision for its access (e.g., ALLOW or DENY) according to the states of connections in networks. However, it is impossibly hard to realize them in current SDNs due to the inherent limitations of OpenFlow.

Some recent research efforts [88, 90, 54, 120, 41, 8, 40, 124] extended the OpenFlow data plane abstraction to support stateful network applications. They attempted to let individual switches, rather than the controller, track the state of connections. We believe that, not only does this design go against the spirit of SDN (because it brings the control plane back to switches and makes switches manipulate connection states and performs complex actions beyond a simple forwarding operation), these existing approaches are only applicable for designing applications that need only *local states* on a single switch [40]. However, such solutions force SDN applications individually access every single switch to collect entire network states, consequently network-wide monitoring to detect abnormalities and enforcing network-wide access control of flows become extremely difficult.

To overcome the limitations of existing approaches, we argue that utilizing the SDN controller for *global* tracking connections is more advantageous than existing solutions in terms of its state visibility across SDN applications that is crucial to some security applications such as a stateful network firewall. To bring such a *state-aware* network access management in SDNs, we propose a novel state tracking framework called StateMon. StateMon models active connections in SDNs and monitors *global* connection states in the controller with the help of both a *global state table* that records the current state of each active connection and a *state management table* that governs the state transition of new and existing connections. StateMon also introduces a lightweight extension to OpenFlow, called *OpenConnection*, that programs the data plane to forward the state-changing packets to the controller. At the same time, it retains the simple "match-action" programmable feature of OpenFlow and avoids scalability problems over the communication channel between the controller and switches. In essence, StateMon follows the general SDN principle of logical-to-physical decoupling and avoids embedding complicated control logic in the physical devices, therefore, keeping the SDN data plane as simple as possible.

In addition, to demonstrate the practicality and feasibility of StateMon and state-aware network access management applications in SDNs, we design a stateful network firewall based on the APIs provided by StateMon. Our firewall application provides more in-depth access control than a stateless SDN firewall [67]. It detects and resolves connection disruptions and unauthorized access attempts targeting active connections in SDNs. To demonstrate the generality of StateMon, we reimplement a prior work (port knocking) based on StateMon (Section 5.5.2). Our experimental results show that StateMon and network access management applications (stateful firewall and port knocking) introduce manageable performance overhead to manage network access control.

Contributions: The contributions are summarized as follows:

- We propose a connection tracking framework called StateMon that enables SDN to support state-aware access control schemes by leveraging global network states. StateMon keeps the data plane as simple as possible, thus being compliant with the spirit of SDN's design principle.
- We propose the *OpenConnection* protocol, which is a lightweight extension to OpenFlow and retains the simple "match-action" programmable feature of OpenFlow to enable a stateful SDN data plane.
- We implement a prototype of StateMon using Floodlight [11] and Open vSwitch. Our experiments demonstrate that StateMon introduces a minimal increase of communication messages with manageable performance overhead (3.27% throughput degradation).
- We design a stateful network firewall application, using the APIs provided by StateMon. Our experiments show that the stateful firewall provides more control than existing stateless firewalls and it can effectively detect and mitigate certain connection-related attacks (e.g., connection disruptions and unauthorized access) in SDNs.

This chapter is organized as follows. We overview the motivating problems in Section 6.2. Section 6.3 presents the design of state-aware StateMon. Section 5.4 describes the design of stateful network firewall supported by StateMon, and the implementation and evaluation details are in Section 5.5. Section 6.8 discusses the

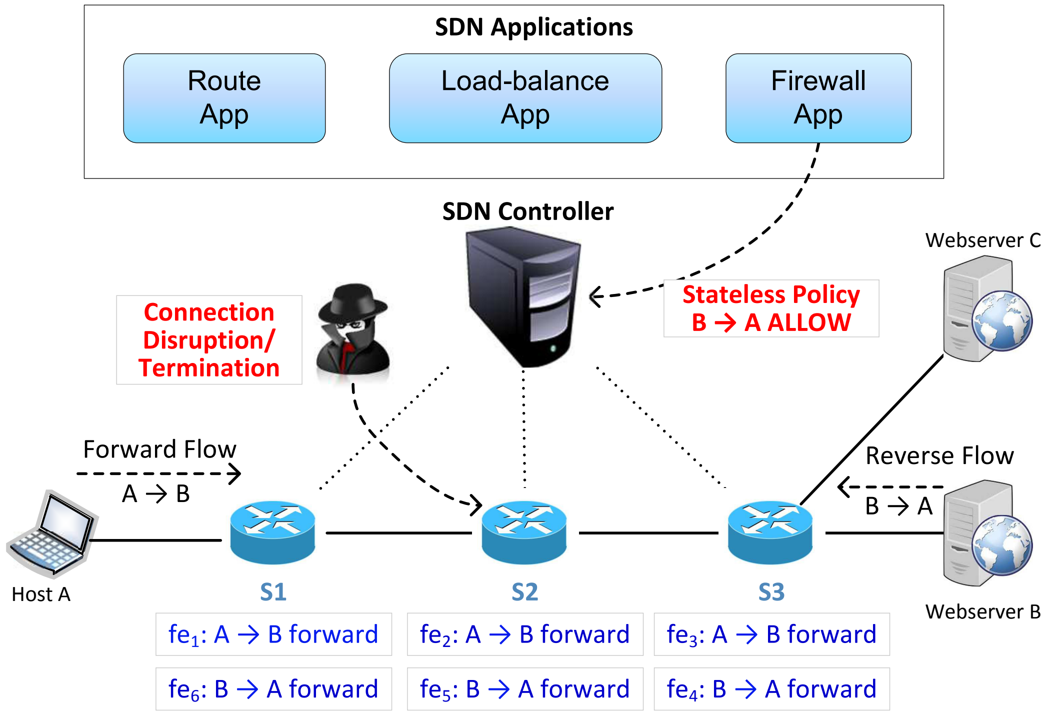


Figure 5.1: Standard OpenFlow Operation and Its Stateless Property.

related work of this chapter, and Section 6.9 describes several important issues. In Section 6.10, we conclude this chapter.

5.2 Background and Problem Statement

To understand our proposed solution to adding state-awareness to SDNs, we provide an overview of the current OpenFlow operation. When an OpenFlow-enabled switch receives a packet, it first checks its *flow tables* to find matching rules. If no such rules exist, this means it is the first packet of a new flow. The switch then forwards the packet to the controller, and it is the controller's job to decide how to handle the flow and to install flow table rules in the appropriate switches. Specifically, the packet is encapsulated in an `OFPT_PACKET_IN` message sent to the controller, and the controller then installs corresponding rules called *flow entries* into the switches along the controller's intended path for the flow. Once these flow entries are installed, all

Table 5.1: Existing Stateful Inspection and Management Methodologies for SDNs. (D = Data Plane, C = Control Plane, A = Application Plane)

Solution	Inspection			Storage			Implementation
	D	C	A	D	C	A	
App-aware [88]	✓			✓			firewall, load-balancer
UMON [120]	✓			✓			software switch
Conntrack [8]	✓			✓			software switch
OpenState [40]	✓			✓			software switch
FAST [90]	✓			✓			firewall
SDPA [124]	✓			✓			firewall, hardware switch
FlowTags [54, 55]	✓		✓			✓	proxy cache
OpenNF [58]	✓	✓			✓		intrusion detection system, network monitor
P4 [41]	✓				✓	✓	(no implementation)
StateMon	✓	✓			✓		-

subsequent packets of this flow are automatically forwarded by the switches, without sending the packet to the controller.

For example, in Figure 5.1, host A wants to initiate a TCP connection with web server B. The first packet (TCP SYN) sent by host A is checked by the ingress switch S1 and forwarded to the controller because S1 has no flow table entry for the packet. The controller allows the flow from host A to server B by installing flow entries fe_1, fe_2, and fe_3, into switches S1, S2, and S3, respectively. The flow from host A to server B is called a *forward flow.* Using the same process, the response packet (TCP SYNACK) generated by server B will trigger the controller to install fe_4, fe_5, and fe_6 into S3, S2, and S1, respectively. The flow from server B to host A is called a *reverse flow.*

As can be seen from Figure 5.1, neither the OpenFlow-enabled switch nor the controller has the ability to track and maintain connection states, which makes it impossible to directly develop stateful access control based on OpenFlow in SDNs. As a result, existing SDN controllers (e.g., Floodlight) only have a stateless firewall application that enforces ACL (Access Control List) rules to monitor all `OFPT_PACKET_IN` behaviors.

Using Figure 5.1 as an example, these stateless firewall applications can only specify simple rules, such as "packets from server B to host A are allowed." In contrast, a *stateful* firewall is a critical component in traditional systems and networks which provides more control over whether a packet is allowed or denied based on *connection state information*. For example, a stateful firewall rule could specify "packets from server B to host A are allowed, if and only if host A initiates the connection to server B." These stateful rules are incredibly useful for security purposes, for instance to specify that a web server should be able to accept incoming connections but never initiate an outgoing connection. However, despite the great security benefit of these stateful policies, it is challenging to build a stateful firewall in SDNs without the full support of stateful packet inspection [67], which is critical to provide effective network access control management.

In addition to the development of a stateful firewall application, the knowledge of connection states in SDNs can also help maintain the network's availability. The SDN controller and applications can install, update, or delete flow entries for their own purposes. However, *these actions may interrupt established connections*, which may consequently damage the availability of services in the network. Consider the case of a load balancer application, which switches flows between two web servers (Servers B and C in Figure 5.1). If the flows are changed while a network connection is still in progress, the availability of the service would be affected. Also, attackers, who are able to perform a man-in-the-middle attack on OpenFlow-enabled switches [39], can also disrupt existing connections in the network by intentionally updating flow entries.

The root cause of these issues is that the controller and the SDN applications have no knowledge of the connection states, which results in creating potential chances of unauthorized access into existing connections by attackers. We argue that a critical functionality of OpenFlow or any other SDN implementation is that the controller should be able to identify the conflicts between active connections and any pending flow entry update and provide network administrators with an early warning before a conflicting flow entry takes effect. Existing verification tools [74, 73, 75, 85] cannot detect and address such conflicts, *because they are unaware of connection states in the network.* By tracking global connection states in the network, the controller will be able to deal with such conflicts and help maintain the availability of the services in the network.

We summarize existing solutions in Table 5.1 that are mostly applicable only for designing applications that need states locally. App-aware approach [88] adds customized OpenFlow table in the data plane to be able to check packets and redirect them to different network applications such as a firewall and a load balancer. However, this application-aware table is statically defined at the compile time and maintains network states **locally**. Consequently, network states are to be distributed whereas StateMon centrally collects all network state and records them in SDN controller. UMON [120] and Conntrack [8] put customized tables in the middle of OpenFlow pipelines in the data plane to perform anomaly detection and stateful packet inspection, respectively. OpenState [40] performs state checking using the state table in conjunction with an extended finite state machine that is directly programmable by the controller. FAST [90] compiles the state machine of a specific network protocol such as TCP in the control plane and installs corresponding state-based tables (state transition table and action table) into SDN switches. SDPA [124] inserts the forwarding processor into OpenFlow-based processing pipeline to enable stateful forwarding scheme including hardware-based design. FlowTags [54, 55] attaches TAG information at the end of TCP header of in-flight packets to record middleboxes' state-based information rather than checking state using SDN switches or the controller. However, none of these approaches is able to maintain global network states centrally and helps SDN applications access the information, allowing them to implement their business logic based on global network states. Only OpenNF [58] and P4 [41] attempt to utilize the control plane of SDNs for state checking and consolidating network states. OpenNF focuses on collecting states of network middleboxes (e.g., IDS, Net-Monitor) to support dynamic middlebox migration, and P4 is a proposal for next generation of OpenFlow to support state inspections. However, the former is not applicable for collecting generic network states (e.g., connection state), and the latter does not include a workable implementation. Thus, we argue that a global connection monitoring framework, which can be aggregated by the controller, is imperative for network-wide connection monitoring and access management. Such a global connection awareness not only enables stateful firewall applications to detect *indirect* policy violations considering *dynamic packet modification* in SDNs, but also helps identify connection disruptions and unauthorized access occurred in existing connections.

5.3 StateMon Design

In this section, we first present the key design goals of our StateMon framework. Then, we illustrate the overall architecture and working modules of StateMon and further show how they meet our design goals.

5.3.1 Design Goals

To enable stateful access management applications and overcome the limitations of existing approaches, we propose a novel state-aware connection tracking framework called StateMon to support building stateful network firewall for SDNs. StateMon is designed with the following goals in mind:

- **Centralization**: StateMon should, in adhering to the principles of SDN, manage a global view of all network connection states in a centralized manner at the control plane.
- **Generalization**: StateMon should support any state-based protocols and provide state information to SDN applications.
- **High Scalability**: StateMon should minimize message exchanges between the controller and switches so that the control channel will not be the performance bottleneck when monitoring all network connection states.

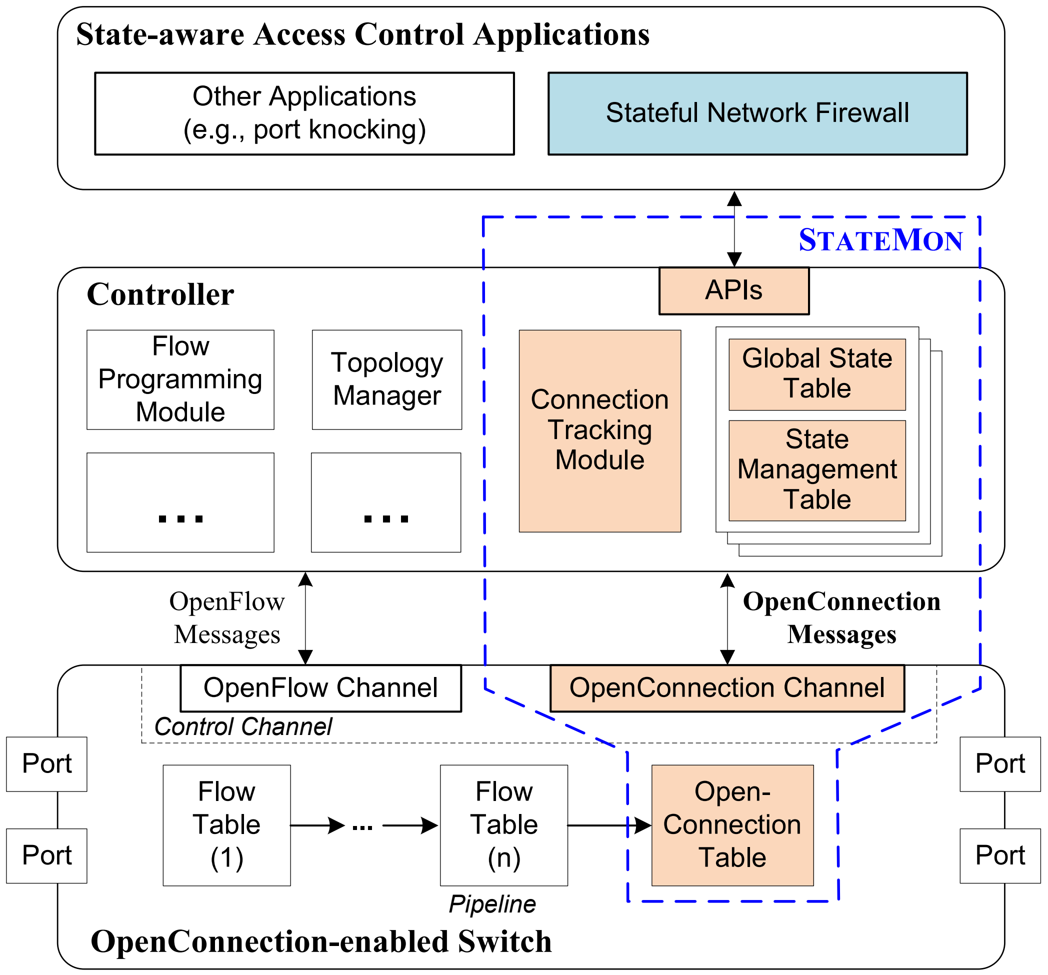


Figure 5.2: StateMon Architecture Overview

5.3.2 StateMon *Architecture Overview*

Figure 5.2 shows an overview of the StateMon architecture, which adds new modules in both the control plane (controller) and the data plane (switches) of the OpenFlow system architecture.

To achieve the centralization goal, StateMon modules in switches use only the match-action abstraction to perform packet lookups, forwarding, and other actions based on the *OpenConnection* table (Section 5.3.3), whereas modules in the controller track a global view of states (Section 5.3.4). A controller uses the *OpenConnection* protocol to program *OpenConnection* tables, which are added to the Open-

Flow processing pipeline by introducing a “Goto OpenConnection Table” instruction (`Goto-OCT`) in OpenFlow action set.

To achieve the generality goal, StateMon maintains a pair of global state table and state management table for each state-aware application. A state-aware application initializes those tables and registers callback functions using the APIs provided by StateMon. The global state table records network-wide connection state information. Each entry in this table represents an active connection by specifying the flow entries that govern the active connection (e.g., $fe_1, \cdots, fe_6$ in Figure 5.1) and its connection state (e.g., ESTABLISHED in TCP). The state management table keeps state transition rules and actions that should be performed on each state (e.g., send an OpenConnection message to the controller).

StateMon uses three methods to minimize the communication overhead between the controller and switches to meet the high scalability design goal. First, StateMon leverages existing OpenFlow protocols such as `OFPT_PACKET_IN` message for monitoring connection states. For example, the first packet of a new flow delivered by `OFPT_PACKET_IN` message would not trigger a separate OpenConnection message. Second, StateMon identifies ingress and egress switches for each connection and only installs necessary OpenConnection entries into those switches to perform a state-based inspection. Thus, StateMon minimizes the increase of additional table entries and avoids the potential overhead that can be generated by other intermediate switches on the path. Third, the OpenConnection protocol sends only expected state-changing packets from switches to the controller.

5.3.3 OpenConnection Protocol

On receipt of a packet, an OpenConnection-enabled switch starts with the OpenFlow-based packet process. For any new flow, the first packet of this flow is forwarded to the

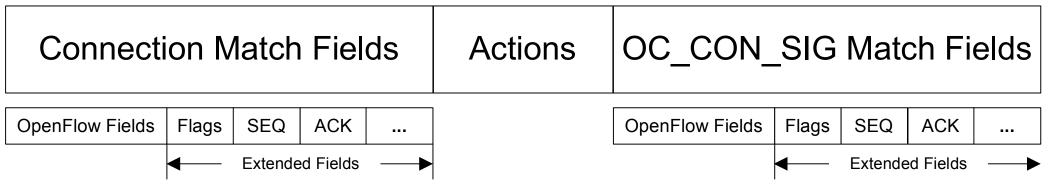


Figure 5.3: Structure of an Entry in an OpenConnection Table.

controller via an `OFPT_ PACKET_IN` message. Then, the controller determines whether that packet should be sent. If so, the controller will install new flow entries into corresponding switches to handle future packets of the same flow. StateMon also listens to the `OFPT_PACKET_IN` message. If this message carries a packet that any state-aware application wants to monitor (Section 5.3.5), StateMon will install OpenConnection entries in OpenConnection tables (Section 5.3.3) of corresponding switches using OpenConnection messages (Section 5.3.3) and add a `Goto-OCT` instruction in the flow entries to start OpenConnection processing pipeline.

OpenConnection Table

Before illustrating how OpenConnection-enabled switches process packets, we first explain the structure of the OpenConnection table. An OpenConnection entry, which is shown in Figure 5.3, has (1) connection match fields, (2) actions for a decision of forward, drop, and update fields, etc., and (3) `OC_CON_SIG` match fields that triggers switches to send `OC_CON_SIG` message when matched. To achieve generality, both connection and `OC_CON_SIG` match fields are directly programmable by state-aware SDN applications (Section 5.3.5).

If and only if a packet matches connection match fields, the packet will be processed by both the OpenFlow and OpenConnection pipeline as shown in Figure 5.4. In case the packet also matches the `OC_CON_SIG` match fields, which means the packet is a state changing packet, such as FIN in TCP, it will be encapsulated in an `OC_CON_SIG` message and forwarded to the connection tracking module of StateMon in the

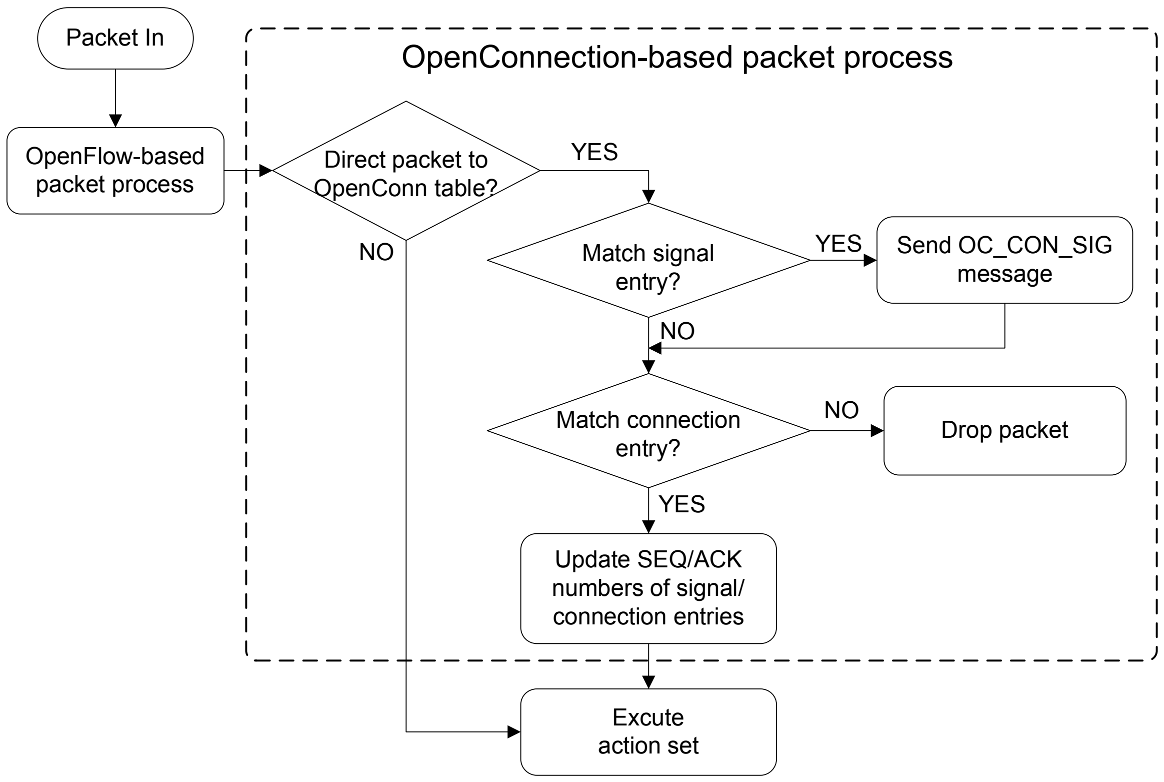

Packet In
OpenFlow-based packet process
OpenConnection-based packet process
Direct packet to OpenConn table?
YES
NO
Match signal entry?
YES
Send OC_CON_SIG message
NO
Match connection entry?
NO
Drop packet
YES
Update SEQ/ACK numbers of signal/ connection entries
Excute action set

Figure 5.4: Flowchart for OpenConnection Packet Processing.

controller. The connection tracking module will maintain the state and manage associated switches accordingly. Upon completion of these OpenConnection-based packet process, the action set that includes the rest of the OpenFlow actions will be executed.

The design of the OpenConnection table is aligned in spirit to the design of the flow table, so that the data plane can process packets using the simple "match-action" paradigm. However, OpenConnection tables are more scalable than OpenFlow tables, because OpenConnection table entries are only installed in the OpenConnection tables of the two *endpoint* switches that directly connect the initiating host and the receiving host of a connection. In contrast, using OpenFlow for each new flow, corresponding flow entries must be installed in *all* flow tables of switches that the flow traverses.

Table 5.2: OpenConnection Messages (C: Controller, S: Switch)

Message Name	Direction	Description
`OC_CON_SIG`	S→C	Encapsulate entire packet (including payload) and forward it to connection tracking module
`OC_ADD`	C→S	Install a new entry in an OpenConnection table
`OC_UPDATE`	C→S	Update an OpenConnection entry
`OC_REMOVE`	C→S	Remove an OpenConnection entry

OpenConnection Message Exchanging Format

We define four OpenConnection messages to enable state-based connection monitoring. OpenConnection messages help the connection tracking module of StateMon monitor the overall process of connection establishment and tear-down behaviors occurring in the data plane. Table 5.2 summarizes the four OpenConnection messages with a brief description of each.

The `OC_CON_SIG` message is used to encapsulate the state-changing packet and conveying it to the controller (switch-to-controller direction). The main difference from OpenFlow `OFPT_PACKET_IN` is that the `OC_ CON_SIG` message is only for StateMon (so that it will not be effective to other SDN applications), and it also contains a randomly generated unique identifier for the connection to distinguish the affiliation of the message. The other messages are sent from the controller to the switches to program an OpenConnection table. The connection tracking module generates a `OC_ADD` message to install a new entry in an OpenConnection table. For instance, to monitor a TCP connection, it installs an entry to match TCP ACK packet at its ingress switch of the flow path. `OC_UPDATE` is used for updating an OpenConnection table entry. If a connection is terminated (or by timeout mechanism), the connection tracking module sends an `OC_REMOVE` message to remove all associated entries. Compared with OpenFlow, which exchanges messages between the controller and multiple switches, OpenConnection introduces only a constant number of message exchanges between the controller and two endpoint switches for handling a specific state-based connection. Using TCP as an example, OpenConnection uses eight messages in total for a TCP connection (see Table 5.5): (1) three `OC_CON_SIG` messages, (2) two `OC_ADD` messages, (3) one `OC_UPDATE` message, and (4) two `OC_REMOVE` messages.

5.3.4 Tracking Connection States

For generality, StateMon maintains a pair of global state table and state management table for each state-aware application. The connection tracking module listens to `OFPT_PACKET_IN` messages to initialize an entry in the global state table for a connection and listens to `OC_CON_SIG` messages to update the states of the connection based on state transition rules in the *state management table* provided by the application.

Global State Table

The global state table records network-wide connection state information. However, simply extracting a connection's state from a specific switch is not sufficient to account for the overall global state of a connection. Because OpenFlow-enabled switches are able to rewrite packets' headers at any point using the Set-Field action, a packet's header may look different at its ingress and egress switches. This poses a challenge for the controller to identify which packets belong to the same connection. To solve this problem, StateMon bonds a connection's state (e.g., ESTABLISHED) with its associated network rules (i.e.,the forward and reverse flow entries) to effectively monitor and track an active connection.

We design the entry in the global state table as 5-tuple denoted $\langle C_I, C_E, \sigma_F, \sigma_R, S_a \rangle$. Connection information at the ingress switch (C_I) contains a set of packet header fields along with its incoming physical switch port, p_i. Connection information at the egress switch (C_E) contains the same elements, except p_o which refers to the out-

Table 5.3: State Management Table Example for TCP connection. (A or B Refers a Pair of $\langle$IP, port$\rangle$.)

State	Transition Conditions			Next State	OpenConnection Events			Timeout
	Message Type	Source	Match Fields		Message Type	Destination	`OC_CON_SIG` Match Fields	
INIT	`OFPT_PACKET_IN`	Ingress	A→B, TCP, Flag=SYN	SYN_SENT	`OC_ADD`	Ingress	A→B, TCP, Flag=ACK	∞
SYN_SENT	`OFPT_PACKET_IN`	Egress	B→A, TCP, Flag=SYNACK	SYNACK_SENT	`OC_ADD`	Egress	B→A, TCP, Flag=FIN	5
SYNACK_SENT	`OC_CON_SIG`	Ingress	A→B, TCP, Flag=ACK	ESTABLISHED	`OC_UPDATE`	Ingress	A→B, TCP, Flag=FIN	5
ESTABLISHED	`OC_CON_SIG`	Ingress	A→B, TCP, Flag=FIN	FIN_WAIT				1800
		Egress	B→A, TCP, Flag=FIN					
FIN_WAIT	`OC_CON_SIG`	Egress	B→A, TCP, Flag=FIN	CLOSED				60
		Ingress	A→B, TCP, Flag=FIN					
CLOSED	-	-	-	INIT	`OC_REMOVE`	Ingress		0
					`OC_REMOVE`	Egress		

going physical switch port. For instance, C_I for a TCP connection can be defined as $\langle$src_ip, src_port, dst_ip, dst_port, network_protocol, $p_i\rangle$. Note that some fields in C_I and C_E (e.g., $src\_ip, src\_port, dst\_ip, dst\_port$) might not be identical due to dynamic packet modification (Set-Field action) in SDNs. σ_F is a series of identifiers of flow entries that enable the forward flow, and σ_R is also a series of identifiers for the reverse flow. For example, the forward flow and the reverse flow in Figure 5.1 would be $\sigma_F = \langle fe_1, fe_2, fe_3\rangle$ and $\sigma_R = \langle fe_4, fe_5, fe_6\rangle$, respectively. The last element, S_a, denotes the state of a connection and it will be further elaborated in Section 5.3.4.

The elements in a global state table entry have several properties. The relation between C_I and C_E is to be determined by σ_F or σ_R such that $C_I \xrightarrow{\sigma_F} C_E$ and $C_E^{-1} \xrightarrow{\sigma_R} C_I^{-1}$. C_I^{-1} and C_E^{-1} are directly derived from C_I and C_E by replacing the source with the destination and changing the incoming port (p_i) to the outgoing port (p_o). For example, if C_I =$\langle$src_ip: 10.0.0.1, src_port: 3333, dst_ip: 10.0.0.2, dst_port: 80, network_protocol: tcp, p_i: 2$\rangle$ then C_I^{-1} =$\langle$src_ip: 10.0.0.2, src_port: 80, dst_ip: 10.0.0.1, dst_port: 3333, network_protocol: tcp, p_o: 2$\rangle$.

State Management Table

An entry in the state management table is a 5-tuple denoted as ⟨*State*, *Transition Conditions*, *Next State*, *OpenConnection Events*, *Timeout*⟩. When an `OFPT_PACKET_IN` or `OC_CON_SIG` message is received, the connection tracking module compares its originated location and header of the encapsulated packet with the *Source* and *Match Fields* of the current state in the state management table. If the packet meets the *Transition Conditions* of the current state, the state will be updated to the *Next State* and *OpenConnection Events* will be triggered. OpenConnection events instruct the connection tracking module to send `OC_ADD`, `OC_UPDATE`, or `OC_REMOVE` to corresponding switches. The *Match Fields* in *OpenConnection Events* will configure the OpenConnection table entries in corresponding switches to initialize connection and `OC_ CON_SIG` match fields. *Timeout* allows StateMon to automatically close a connection.

Table 5.3 shows how a state-aware application can use the state management table for the TCP state transitions. A TCP connection starts with INIT state that transitions to SYN_SENT when it receives an `OFPT_PACKET_IN` message that contains a TCP SYN flag. StateMon identifies the location of the ingress switch (I) from the message, and it sends an `OC_ADD` message back to I with its match fields. StateMon locates the egress switch (E) as well by listening for the second `OFPT_PACKET_IN` message. `OC_CON_SIG` messages collected from I or E are then used to update the connection states. `CLOSED` is a temporary state only used for sending `OC_REMOVE` messages and removing the associated entries. Note that one state can transition to multiple `Next States` based on matching conditions and generate a variety of actions as defined by SDN applications.

Table 5.4: StateMon APIs

Category	API Name	Key Parameters	Description
`Type I`	InitGST()	Match fields in C_I and/or C_E	Initialize the global state table
	InitSMT()	5-tuple of state management table	Initialize the state management table
	SetInterest()	Range of match fields with wildcard	
`Type II`	SearchEntry()	Raw packet or ConnectionID	Search an associated global state entry
	GetConnState()	ConnectionID	Obtain current state of a connection
	DeleteEntry()	ConnectionID	Delete a connection
`Type III`	ConnAttempt()	Type of message and raw packet	Return one of actions (`allow` or `drop`)
	StateChange()	ConnectionID and next state	

5.3.5 StateMon APIs

StateMon provides three types of application programming interfaces (APIs) for SDN applications so that the applications only need to implement their business logic. The APIs can be used (1) to configure both the global state table and the state management table (`Type I`), (2) to retrieve state information from the global state table (`Type II`), and (3) to register callback functions in StateMon to subscribe specific state-based events (`Type III`). The APIs are summarized as follows:

- `Type I` is used to configure the two state-specific tables in StateMon: the global state table and the state management table. To customize the global state table, SDN applications can specify match fields for C_I or C_E (e.g., IP and port number) to distinguish one connection from another. Applications can also define a state set for the connection along with its transition rules for the state management table.

- `Type II` APIs are built for sending queries (applications to StateMon) to retrieve network states, which SDN applications are interested in. Because all connection information is recorded in the global state table, those queries are directly conveyed to the global state table.

- `Type III` APIs are used to register callback functions in StateMon. For example, when a global state entry is updated, StateMon can call this function to subscribing applications to allow them to execute their own business logic.

5.4 Stateful Firewall Design

In this section, to demonstrate the practicality and feasibility of StateMon and state-aware network access management applications in SDNs, we illustrate how a stateful firewall can take advantage of StateMon to implement its state-aware access control logic in SDNs.

The stateful firewall application first calls `Type I` APIs to initialize its global state table and state management table. We focus on TCP connections as a state-based protocol for this application. To enforce a stateful firewall policy such as "host B can communicate with host A if and only if host A initiates the connection," our firewall uses the state management table shown in Table 5.3. Then, StateMon calls the registered callback function (`Type III`) when a state changing event occurs. The application only needs to implement the logic in the callback function: (1) a packet (or flow) heading from host B to host A should be denied when its state is in INIT or SYNACK_SENT and (2) a packet (or flow) heading from host B to host A should be allowed when its state is in SYN_SENT or ESTABLISHED. Thus, the connection attempt (e.g., TCP SYN) initiated from host B cannot be made whereas the attempt from host A will pass.

To show some benefits of our stateful firewall, we focus on following features: (1)

Table 5.5: Additional State Management Table Entries for Unauthorized Access Prevention

State	Transition Conditions			Next State	OpenConnection Events			Timeout
	Message Type	Source	Match Fields		Message Type	Destination	`OC_CON_SIG` Match Fields	
SYNACK_SENT	`OC_CON_SIG`	Ingress	A→B, TCP, Flag=ACK	ESTABLISHED	`OC_ADD`	Egress	A→B, TCP, Flag=FIN	5
SYNACK_SENT	`OC_CON_SIG`	Ingress	A→B, TCP, Flag=ACK	ESTABLISHED	`OC_ADD`	Ingress	B→A, TCP, Flag=FIN	5
ESTABLISHED	`OC_CON_SIG`	Egress	A→B, TCP, Flag=FIN	**DETECTED**				1800
ESTABLISHED	`OC_CON_SIG`	Ingress	B→A, TCP, Flag=FIN	**DETECTED**				1800
DETECTED	-	-	-	ESTABLISHED				0

state-aware firewall policy enforcement, (2) connection disruption prevention, and (3) unauthorized access prevention against active connections.

5.4.1 State-aware Firewall Policy Enforcement

Since StateMon provides global network states to the firewall, our firewall application utilizes the state information for the following scenarios: (1) a host attempts to establish a new connection, (2) the state of an active connection has been updated, and (3) the firewall application updates the firewall policy.

First, when host A attempts to open a new connection to host B, both host A and host B exchange initiating signal packets to establish the connection. As soon as StateMon receives these attempts, the firewall would get relevant information via the `Type III` callback function defined when it called ConnAttempt(). If this attempt violates the pre-defined stateful firewall policy, the initiating packet is immediately denied and the firewall stops the controller from executing the rest of the `OFPT_PACKET_IN` handling process so that no flow entry is sent to the switches.

Second, if a global state entry is updated, the stateful firewall will also be notified via `Type III` callback function, StateChange(). Our firewall application performs pair-wise comparison, the current state of the connection against existing stateful firewall policies. The firewall searches the associated global state entry by calling SearchEntry() and acquires the connection information from the entry.

Algorithm 4: Obtaining Affected Entry Set (AES)

```
Input: New (or Updated) flow entry ($nf$) and existing flow entries ($FE = \{e_1, e_2, ...\}$) at the same switch.
Output: Affected entry set $AES = \{a_1, a_2, ...\}$ such that $a_i \in FE$.
/* First, append the new flow entry ($nf$) to AES */
$AES.append(nf)$;
/* $FE_t$: a set of flow entries installed in table $t$ */
$FE_t \longleftarrow retrieveEntries(nf.getSwitchID, nf.getTableID)$;
foreach $e \in FE_t$ do
    /* Check if $nf$ has higher priority than $e$ and is dependent with $e$ */
    if $nf.priority \geq e.priority$ and $nf.match \cap e.match \neq \emptyset$ then
        $AES$.append($e$);
        /* Recursively perform identical operation if $e$ has Goto-OCT instruction */
        if $e.getInstruction$ contains $GotoTable$ then
            $temp\_e$.match $\longleftarrow e.applyActions()$;
            $temp\_e.setTableID(e.getInstruction.getTableID)$;
            $AES\_child = self.(temp\_e, E)$;
            $AES.append(AES\_child)$;
        end
    end
end
return $AES$;
```

To consider `Set-Field` actions, it retrieves *tracked* space denoted $T(I, E)$, getting $\langle src\_ip, src\_port \rangle$ from I and $\langle dst\_ip, dst\_port \rangle$ from E. By putting them together, we obtain $T(I, E) = \langle I.src\_ip, I.src\_port, E.dst\_ip, E.dst\_port \rangle$. Using the combination of $T(I, E)$ and its current state, the firewall checks for rule compliance with firewall policies. If the update of the state is not allowed by the policy, the application raises an alarm to network administrators and the update is denied by setting the return value of StateChange() to `drop`. In case the stateful firewall application wants to remove the connection, it may invoke DeleteEntry() function to remove the associated entries from the OpenConnection and flow tables.

The final scenario deals with the case of updating firewall policies. When the firewall application updates a stateful rule in its policy set, all active connections are examined against the new rule to identify potential violations. Because each firewall policy has a priority, computing dependency relations of firewall rules after the updates are vital for identifying overlaps between rules. All violating connections are to be deleted from the network by calling the API DeleteEntry(). As a result, the associated OpenConnection and flow entries will be flushed from the OpenConnection tables and flow tables.

5.4.2 Connection Disruption Prevention

A malicious SDN application can manipulate existing flow entries or install new flow entries to disrupt active connections that consequently damage the availability of services in the network. To prevent this type of attack, detecting these attempts before they take effect in the network is mandatory, so our firewall application proactively analyzes the expected impact of updates on active connections. To this end, the application computes the Affected Entry Set (AES) as described in Algorithm 4. When a new flow entry is to be inserted into the network or an existing flow entry is about to be updated, the application computes its dependencies with existing flow entries in the same switch. To this end, it first retrieves all flow entries FE from a specific switch and computes affected flow entries by new (or updated) flow entry nf. The application next selects the exact flow table affected by nf and builds FE_t which is a subset of FE. Then, it compares the priority and matching conditions between e and nf, to decide whether e is affected. If nf is dependent on e and has higher priority than e, the application adds e into AES. If e has a goto instruction, the application further visits the specified flow table to find AES_{child}. Considering Set-Field actions e may have, the actions will be applied first in advance before pipelining to another flow table. The firewall makes use of AES to detect the connection disruption attacks.

Detection of connection disruption attacks: Newly installed (or updated) flow entry nf triggers the application to compute AES and check AES against active connections obtained from StateMon. The application then compares AES with σ_F and σ_R of each of active connections and invokes the connection tracking module to re-calculate σ'_F and σ'_R. The updated σ'_F may change the relation between C_I and C_E i.e., $C_I \xrightarrow{\sigma'_F} C'_E$. If $C_E \neq C'_E$, the firewall concludes that the candidate flow entry nf will disrupt an active connection. nf may also disrupt the reverse flow of the connection. If $C_E^{-1} \xrightarrow{\sigma'_R} C_I'^{-1}$ and $C_I^{-1} \neq C_I'^{-1}$, the firewall also concludes nf will disrupt an active connection.

Countermeasure: When the controller receives the request of installation of a new flow entry nf which may cause a connection disruption or interruption, StateMon treats it as a candidate flow entry and holds it until StateMon evaluates its impact on the network. Upon completion of computing AES and σ'_F (or σ'_R), if the firewall detects any error such as $C_E \neq C'_E$ or $C_I^{-1} \neq C_I'^{-1}$, it raises an alarm to the administrator about the attempt. The administrator can decide whether it is legitimate and an intended request. If it turns out nf is valid, StateMon allows it to be installed in the network. Otherwise, the firewall rejects the installation of nf.

5.4.3 Unauthorized Access Prevention

An attacker can attempt unauthorized access into an active connection by performing a man-in-the-middle attack such as TCP sequence inference attack to spoof packets. TCP protocol is inherently vulnerable to sequence inference attacks [99, 98]. We do not fundamentally solve these known vulnerabilities but can partially prevent specific types of unauthorized access to an active connection (e.g., TCP termination attacks). If an attacker successively infers the sequence number of the next packet, he/she will be able to create a spoofed termination packet by setting the TCP flags with FIN (i.e., man-in-the middle attack [39]). Our firewall can leverage StateMon to detect such an attack by customizing the state management table and adding OpenConnection entries.

Detection of connection termination attacks: The key idea of the detection mechanism is to add additional checking logic in the egress switch for the forward flow (or the ingress switch for the reverse flow) by installing new OpenConnection entries. In addition to the state management table described in Table 5.3, the firewall adds additional transition rules (Table 5.5) to install OpenConnection entries and detect connection termination attacks. The firewall first creates a new *OpenConnection Events* (the first line in Table 5.5) for the SYNACK_SENT state that instructs the egress switch to install a new OpenConnection entry that matches the forward flow. `OC_CON_SIG` match fields of this entry will match the TCP FIN packet that belongs to the forward flow. Benign TCP FIN requests sent from the initiating host will be checked at its ingress switch by Table 5.3, so StateMon transitions the state of the connection to the ESTABLISHED state. Hence, `OC_CON_SIG` fields of the third entry in Table 5.5 will not match the packet. However, attacking packet which is forged by an attacker in the middle of the flow path will match the `OC_CON_SIG` conditions of the third entry at the egress switch which results the state to be DETECTED. DETECTED state defined in the fifth line in Table 5.5 is a temporary state that is used to inform the existence of a TCP termination attack to the firewall. In the case of the reverse flow, the firewall leverages the second and the fourth entry for detecting connection termination attacks. In such a way, the firewall can capture this type of attack with the help of StateMon.

Countermeasure: To protect the network from the aforementioned unauthorized access (e.g., TCP termination attack), the firewall can take two countermeasures: (1) return *actions* in the `Type III` callback function with `drop` to drop the spoofed packet and (2) rollback the connection state (DETECTED to ESTABLISHED) to maintain the connectivity between end hosts. In addition, the firewall may add complementary business logic in a `Type III` callback function to implement post processing behaviors such as sending warning messages to the network administrator.

5.5 Implementation and Evaluation

5.5.1 Implementation

To implement StateMon, we chose a widely used controller, Floodlight, and a reference OpenFlow software switch implementation, Open vSwitch (ovswitch). The routing module and link discovery modules in Floodlight are used to provide network topology information to the connection tracking module. To track existing flow entries in the network and build its reachability graph, we used header space analysis [73] which translates each flow entry into a transition function that consists of a set of binaries, `0`, `1`, and `x` (for wildcard), to represent its matching conditions and actions. We also added `OFPT_PACKET_IN` listener within the controller along with an OpenConnection message handler to receive the state changing packets and program OpenConnection tables. Each global state entry has a unique identifier to distinguish it from other entries for ease of maintenance. The connection tracking module leverages the `OFPT_FLOW_MOD` OpenFlow message to construct controller-to-switch OpenConnection messages.

In the data plane, we implemented the OpenConnection table along with OpenConnection message handler. Because current versions of ovswitch can only support OpenFlow up to version 1.3.0, which cannot inspect TCP flags and sequence/acknowledgment numbers, we implemented a parsing module to additionally retrieve TCP flags and sequence/acknowledgment numbers. Then, we modified the legacy

OpenFlow pipelining logic to enable OpenConnection-based packet processing. In total, less than 500 lines of C code were added to the ovswitch code base.

To implement the stateful firewall we leveraged a built-in firewall application in Floodlight to add a stateful checking module. A stateful checking module in the firewall is able to access the global state table by using StateMon APIs for checking and enforcing its stateful firewall policy. We added the `state` parameter to REST interface methods provided by the built-in firewall so that users can define a stateful policy using REST requests. To prevent connection disruption and unauthorized access, we added a listener in the *Static Flow Pusher* module in Floodlight, so the application is able to intercept potentially malicious or accidentally harmful flow entry update requests and analyze their impacts on active connections before they become effective.

5.5.2 Evaluation

To manage the state of a connection, existing solutions add the transition logic of the connection in the data plane (Table 5.3). The fundamental question, therefore, is how many additional messages and/or performance overhead are introduced to achieve the same goal in StateMon. To this end, we conducted experiments using three virtual machines, each of which had a quad-core CPU and 8GB memory and ran a Linux operating system (Ubuntu). One virtual machine was used to run the Floodlight controller and each of another ran Mininet [22] to simulate two networks. After we built two separated networks, we connected them using a GRE tunnel to flexibly add new hosts and links in one network without impacting the other network. We also modified the size of the network by changing the number of intermediaries (i.e., network switches).

Evaluation of StateMon

To measure the worst-case performance of StateMon, it was configured to monitor every connection in the network. However, in a real-world deployment, StateMon only needs to monitor connections specified by state-aware applications, which will only improve the performance.

We first conducted experiments on an OpenConnection-enabled switch to test the overhead created by StateMon in the data plane. OpenConnection enabled-switch spent less than 1μ for checking the affiliation of incoming traffic in an OpenConnection table when the table is set to have 100 entries. Creating and updating the corresponding entries in the OpenConnection table have been completed within $2\mu s$ on average.

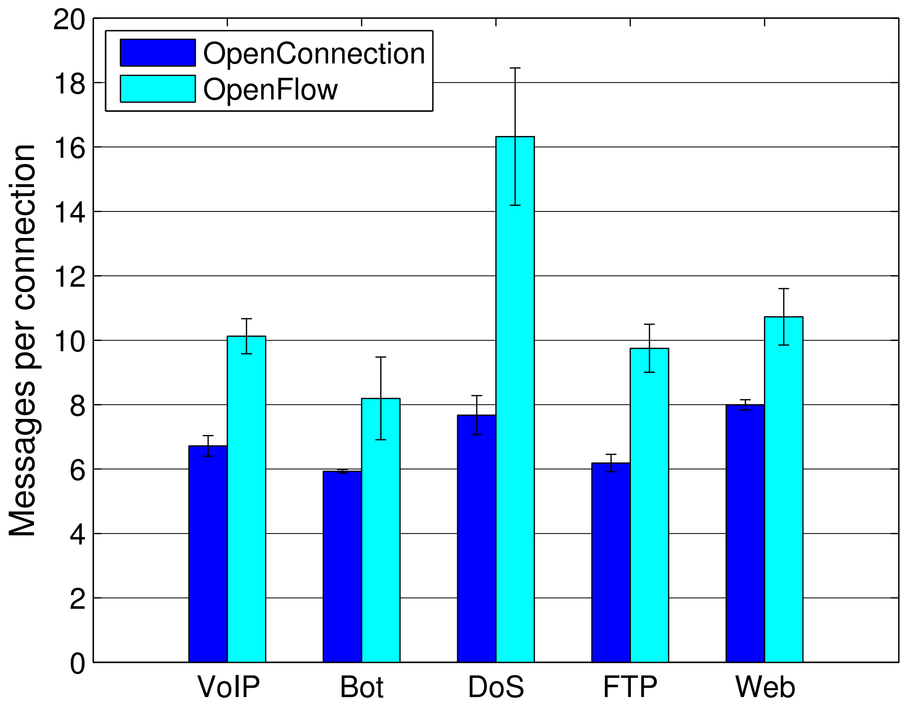
In the controller side, the connection tracking module is in charge of installing/deleting an entry in the global state table and computing next state using the state management table. This module spent less than $3\mu s$ on average to complete those two tasks when there exist 100 connections in the network. To evaluate how much of the delay can be attributed to network latency, we compared the numbers of message exchanges generated by both OpenFlow protocol and OpenConnection protocol. We collected real network traffics (five PCAP files) from different sources (available at [26, 20]) to generate real network traffic. Our testing framework (1) automatically identifies source and destination IP addresses of each packet in a PCAP file, (2) dynamically generates hosts for those IP addresses in a network, and (3) sends the packet through their network interfaces. Figure 5.5a shows the number of message exchanges. The first traffic is collected from VoIP traffic and consists of 32 connection attempts and 29 successful establishments. Network traffic generated by this file caused the controller to generate 324 OpenFlow messages along with 215 Open-

Connection messages, which mean 10 OpenFlow messages and 7 OpenConnection Messages per connection on average. For counting OpenFlow messages, we excluded unrelated messages, such as `OFPT_HELLO`, `OFPT_ECHO_REQUEST`, and `FEATURE_ REPLY`, and filtered out unrelated `OFPT_PACKET_IN` messages used to handle connectionless packets, such as LLDP, ARP, and DNS. Therefore, OpenConnection protocol actually generated much fewer messages than OpenFlow protocol. To account for theoretical number of OpenFlow messages, we develop the equation (5.1). For one way flow, we need one `OFPT_PACKET_IN` message and n number of `OFPT_ FLOW_MOD` messages where n is the number of switches on the path. Because a connection requires bi-directional flows, it is computed by $2 * (1 + n)$.

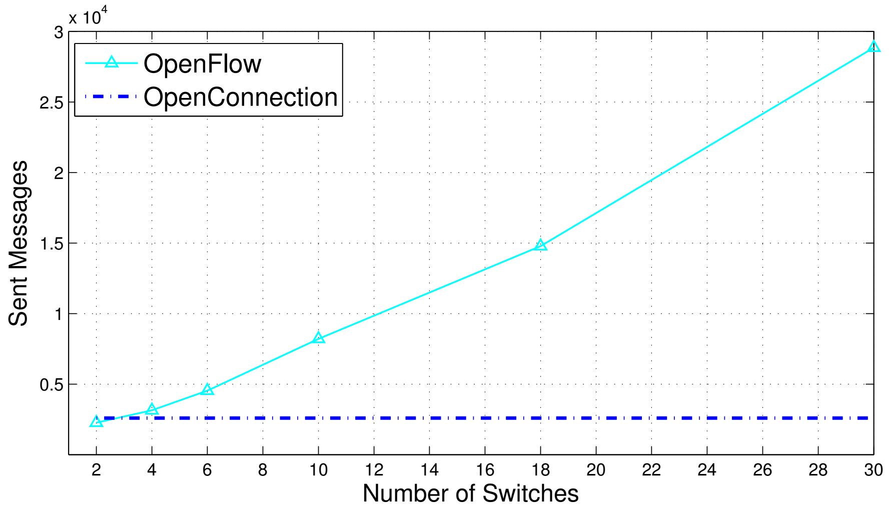
$$B_{OF}(n) = 2 * (1 + n) \tag{5.1}$$

However, the number of OpenConnection messages does not depend on n. Because StateMon requires eight messages for monitoring a connection, every PCAP type in Figure 5.5a creates ≤ 8 OpenConnection messages per connection. Considering the third traffic that contains DoS attacks, it has generated a large number of OpenFlow messages due to substantial connection attempts, while the count of OpenConnection messages remained unchanged. This results clearly show StateMon creates minimal message exchanges under any circumstances. Figure 5.5b shows how StateMon scales with respect to increasing the number of switches in the network. To stress an overhead, we maintained 300 connections when measuring Figure 5.5b. As expected, OpenFlow message count was linearly increased in accordance with the growing number of switches while StateMon maintains a constant number of message exchanges no matter how many switches exist in the network.

To discover overall overhead of StateMon including network latency, we first measured the time for establishing a connection using a TCP handshake with and



(a) Messages per connection of each PCAP file.



(b) Message exchanges with different number of switches.

Figure 5.5: Message Exchanges in StateMon

without StateMon. As defined in Table 5.3, StateMon exchanges 4 messages to monitor a TCP handshake. While a TCP handshake took 3.356ms on average without

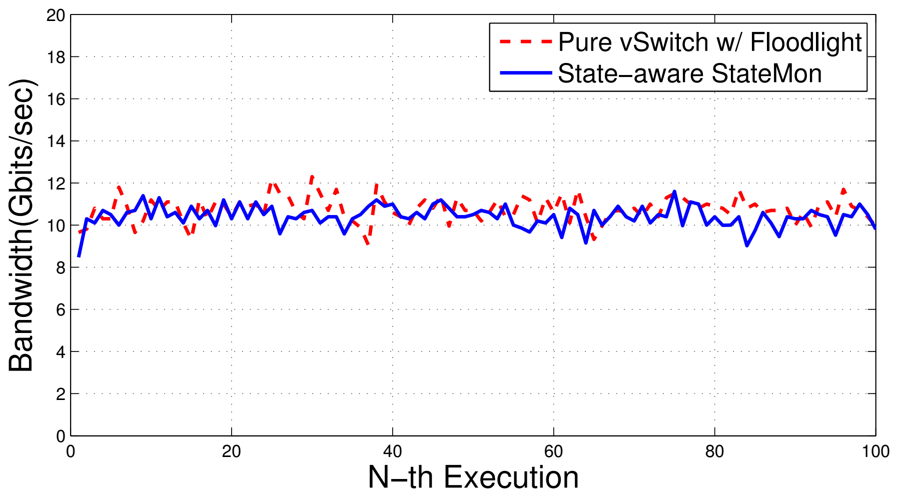


Figure 5.6: Throughput Between End Hosts

StateMon, it took 3.651ms on average with StateMon. This means StateMon only introduced a 0.295ms delay, which is 8.79% overhead for a TCP handshake. To evaluate the overall performance degradation caused by StateMon, we used the throughput between hosts as another metric. We used Iperf [1] for this experiment. Iperf client (host in network A) initiated a new connection with Iperf server (host in network B) and exchanged a set of packets to measure the throughput. In an Open vSwitch and Floodlight setting without StateMon, the throughput scored an average of 10.74 Gbits/sec (100 runs). With StateMon enabled, the throughput scored 10.40 Gbits/sec on average, with only 3.27% throughput degradation.

Evaluation of Stateful Network Firewall

We configured the number of firewall policies to be 1k and fixed the size of global state entries with 10k to measure the overhead of our stateful firewall.

For performing state-aware firewall policy enforcement, the firewall spent 1.02ms on average. When a host attempts to establish a new connection, it took 0.83ms to complete the searches with existing firewall policies, and the attempt was immediately denied in real-time ($0.01ms$). Whenever a global state entry is updated, the firewall performed a pair-wise comparison of the update with existing state-based rules within $1.16ms$, and it took $0.26ms$ to delete the violating connection from the network. In case of firewall policy updates, the firewall finished its dependency checking mostly within $0.5ms$, and spent a similar time ($0.31ms$) for deleting the conflicting connection from the network.

Preventing connection disruptions in the network is another key feature in our firewall. To this end, the firewall computes the Affected Entry Set (AES), and generating AES took less than $0.35ms$ on average. In addition to AES, the firewall computes updated flow entries, namely σ'_F or σ'_R, to further compute $C^{'}_E$ and $C^{'}_I$, respectively. By comparing the relation the old C_E and the updated $C^{'}_E$, the firewall draw a conclusion of potential connection disruption iff $C_E \neq C'_E$. All these tasks were completed in $0.49ms$ on average.

To detect/prevent unauthorized access into active connections, the firewall manipulates the state management table as described in Section 5.4.3. As shown in Table 5.5, the firewall proactively installs necessary rules in the state management table. Once a connection has successively been established between two end hosts, the firewall asks StateMon to install an additional OpenConnection entry to monitor the terminating packet at its egress switch. Since the firewall will be directly notified by StateMon when a connection termination attack is detected, the firewall only implements a logic to drop the attack packet. The firewall drops this packet and recovers the connection's state to its previous one, ESTABLISHED. Duration time for handling this type of unauthorized access took around $0.44ms$ in total.

We also checked the scalability of the stateful firewall application by measuring the duration time for completing three types of strategies. We gradually increased the number of existing connections from 20k to 100k. As shown in Figure 5.7, state-

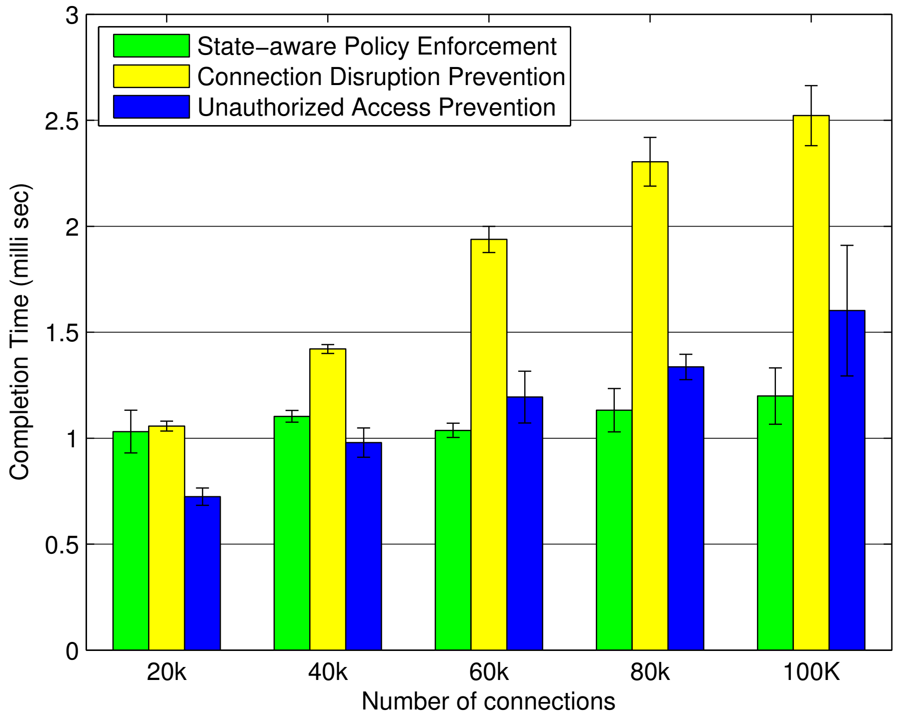


Figure 5.7: Scalability Analysis of Stateful Firewall

aware policy enforcement took almost constant time ($\approx 1ms$) no matter how many connections exist in the network. The firewall spent more time in preventing connection disruptions than that of unauthorized access prevention due to the computation overhead incurred by Algorithm 4. However, overall duration time for both cases linearly increased with respect to increasing number of connections and took less than 3 milliseconds at 100k connections, which is manageable.

Evaluation of Other Application: Port Knocking

Even though we mainly focused on TCP connection in this chapter, a key design goal is that StateMon can support different state-based protocols, such as port knocking. Port knocking is a method to open a *closed* port by checking a unique *knock sequence*, a series of connection attempts destined to different ports [80]. Thus, we developed this application to demonstrate how other network access management schemes can be also implemented using StateMon in SDNs.

For example, an application may want to allow a connection iff a series of requests matches a specific *port* order of A, B, C, and D. By modifying the state management table in StateMon, the application can receive state-changing packets by listening `OFPT_ PACKET_IN` messages. In other words, the initial state can transition to the first *knock* state (e.g., PORT_KNOCK1) when the packet is destined to port A, waiting for the subsequent knocking sequence (port B). Such a way, the application *opens* the *closed* port of a server if the state becomes the OPEN state.

To evaluate the overhead incurred by StateMon-based application, we reimplemented the port knocking that has been demonstrated in prior work [80], which performs the same functions but locally maintains the state in the switch. We installed the state transition rules for the port knocking in the switch. To complete the knocking sequence, it took 104.96ms without StateMon, and StateMon-based application spent 113.83ms in total (8.45% overhead).

5.6 Related Work

As explained in Table 5.3, majority of existing solutions are focused on performing stateful inspection in the data plane [88, 90, 54, 120, 41, 8, 40, 124]. There is some debate as to whether this design goes against the spirit of SDN's control and data plane separation. In addition, none of these approaches give much attention on how to leverage the *logically centralized controller* for providing a *global* state visibility of the network to applications. In contrast, the unique contribution of StateMon comes from its consolidated state checking mechanism enabled by OpenConnection protocol and the connection tracking module. Specifically, StateMon can provide *global* state-based connection information to SDN applications along with several APIs that allows them to define application-specific states. Even though OpenNF [58] attempts to achieve a similar state sharing, it mainly collects a state of middleboxes (e.g., firewall, proxy, and load-balancer), not generic network states.

A number of verification tools [95, 67, 85, 75, 74, 73] for checking network invariants and policy correctness in SDNs have been recently proposed. FortNOX [95] was proposed as a software extension to provide security constraint enforcement for OpenFlow-based controllers. However, the conflict detection algorithm provided by FortNOX is incapable of analyzing *stateful* security policies. FlowGuard [67] was recently introduced to facilitate not only an accurate detection but also a flexible resolution of firewall policy violations in dynamic OpenFlow-based networks. However, the design of FlowGuard fully relies on flow-based rules in the data plane and is only capable of building a *stateless* firewall application for SDNs. Anteater [85] is indeed an offline system and cannot be applied for a real-time flow tracking. VeriFlow [75] and NetPlumber [74] are able to check the compliance of network updates with specified invariants in real time. VeriFlow uses graph search techniques to verify network-wide invariants and deals with dynamic changes. NetPlumber utilizes Header Space Analysis [73] in an incremental manner to ensure real-time response for checking network policies through building a dependency graph. Nevertheless, none of those tools are capable of checking *stateful* network properties in SDNs.

5.7 Discussions

The OpenFlow protocol is evolving continuously, and the latest version (v1.5.0) has been recently released [30]. The newest version of OpenFlow attempts to add *TCP flags* for the extended matching criteria to address the problem of insufficient L4 header inspection capability as we have discussed.

However, the newest version of OpenFlow could not answer critical questions related to the maintenance and manipulation of network connection states. Especially, it does not articulate how to leverage *TCP flags* to monitor states in both the switch and controller. We expect that our design of OpenConnection in StateMon could provide an inspirational solution for OpenFlow to build and enable its future stateful inspection scheme.

While we took great efforts to realize state-aware applications for SDNs, the deployment of StateMon to real-world production networks requires additional considerations in terms of network security. For example, defense mechanisms against DDoS attacks discussed in [110] may need to be considered in StateMon. In addition, the current design and implementation of StateMon utilize OpenFlow-based controller and switch modules, hence it only works in the context of an OpenFlow-based environment. However, the main idea of StateMon, which is to provide state tracking framework for various network applications, can be also realized in other network paradigms, such as Network Function Virtualization (NFV) [3, 61] .

5.8 Conclusion

In this chapter, we have articulated network access control issues in SDNs and presented a state-aware connection tracking framework called StateMon that facilitates the control and data planes of SDN to enable stateful inspection schemes. In the control plane, we have designed a novel connection tracking mechanism using a global state table and a state management table to track active connections. To enable a state-aware data plane, we have introduced a new OpenConnection protocol, which defines four message formats and a state-aware OpenConnection table. We have implemented StateMon using Floodlight and Open vSwitch along with two access management applications (i.e., a stateful network firewall application and a port knocking application) for SDNs, to demonstrate the flexibility of StateMon.

Our experimental results have demonstrated that StateMon and two state-aware network access management applications showed manageable performance overhead to enable critical state-aware protection of SDNs.

Chapter 6

HONEYPROXY: DESIGN AND IMPLEMENTATION OF NEXT-GENERATION HONEYNET VIA SDN

6.1 Introduction

Today's internet system has evolved rapidly, leading to more and more business transactions (e.g., e-commerce and banking) carried out exclusively over the network. These sensitive transactions call for a need to provide a secure and reliable system that can effectively prevent security breaches, to preserve the integrity of the data and protect customers [81]. In addition to securing the integrity of the data, network infrastructure is faced with new exploits, automated scanning tools, and bots, which results in a great loss of business assets and units [96, 32].

It is difficult, if not impossible, to protect the network if we do not have knowledge of the attacker's techniques. In this way, network defenders can remain up-to-date on the latest exploits, tools, and bots. We require software tools and techniques to study behaviors of attackers and exploits. Some research and studies carried out in this direction provide robust mechanisms for early detection and prevention of such attacks thereby securing the network infrastructure. One such attempt is the advent of *honeypots*. A honeypot is a system that is designed to intentionally let attackers probe, scrutinize and ultimately exploit the system by exposing a set of vulnerable services [116, 32, 106]. The primary purpose of a honeypot is to closely monitor the emulated system to learn attackers' behaviors and collect malicious data during and after the exploitation of the honeypot. Honeypots are under active attack by real adversaries, therefore they are often isolated from the real operating system, services, or network. Therefore, honeypots' collected data and attackers' log can provide early warnings of new attacks and exploitation, to help administrators protect the real systems and networks from real adversaries.

Honeypots are generally categorized into two types: a low-interaction honeypot and a high-interaction honeypot. The main difference between them lies on their complexity and the level of interaction they provide to the attacker. Low-interaction honeypots emulate operating systems and other services, therefore low-interaction honeypots do not provide attackers much control. The main advantage stems from their simplicity (i.e., easy deployment and maintenance) and the low risk factor because they are not real production system. High-interaction honeypots are typically actual systems (i.e., not emulated systems), and therefore elicit more interactive information from attackers than that of low-interaction honeypots. However, the high level or interactivity has a downside in considerable maintenance and deployment cost with a high risk factor.

As an example, kippo [21] is one of the well-known low-interaction honeypots that emulates the ssh service to record brute force attacks and log all shell activities performed during the active session with attackers. However, shortcomings of kippo include a risk of being easily fingerprinted due to its nature of an insufficient level of interactions. To emulate the real ssh service, kippo mimics several functionalities using hard-coded strings, which make it vulnerable to fingerprinting attacks [62, 6].

A *honeynet* is an evolution of a honeypot, and it consists of a collection of honeypots. However, this collection poses the same weakness. In addition, the first honeynet architecture (Gen-I [115]) has first been proposed in the year of 2002, and the latest architecture, Gen-III [33], was built in 2004. Due to the outdated honeynet architecture, existing honeynet suffers from insufficient *data control* mechanisms and *data capture* capability. As described in Table 6.1, honeypots kept evolving (see

Table 6.1: Emerging Honeypots and Their Last Updates (As of September 2016).

Honeypot	***Interaction Level***	***Emulated Services***	***Last Commit***
Glastopf [12]	Low	HTTP	Aug 16
HIHAT [14, 91]	High	HTTP	Apr 16
Honeyd [16, 96]	Low	Network	Dec 13
Dionaea [10]	Low	HTTP, FTP, SMB	Jun 2014
Honeytrap [18]	Low	TCP	Jun 2016
Kippo [21]	Low	SSH	Oct 2015
Conpot [9]	Low	ICS (SCADA)	Aug 2016

Last Commit in Table 6.1) to accommodate an emerging services (e.g., industrial control system, or ICS) and catch up the enormous growth in well-crafted attacks and exploits. However, the latest architecture (Gen-III) remained unchanged for more than a decade—it is consequently losing its momentum. For example, inbound-/outbound traffic control mechanisms in Gen-III architecture cannot prevent internal propagation of malware within a honeynet because access control rules are mainly enforced by a custom gateway called *honeywall* [51, 84]. It is also incapable to support the transition between a low-interaction honeypot and a high-interaction honeypot. Low-interaction honeypots are beneficial to collect high level information about attackers (e.g., username and password pair) whereas high-interaction honeypots focus on collecting low level details [15]. However, existing honeynet architecture does not provide a practical way to fully utilize the advantages of both low-interaction and high-interaction honeypots.

In order to solve aforementioned problems, we argue that the architecture of current honeynet should be redesigned to provide more flexibility in terms of its network access management. We observe that such flexibility and network access controls can be satisfied by taking advantage of Software-defined Networking (SDN [23]). SDN basically provides a centralized network management platform by decoupling the control plane (e.g., exchanging network rules) from the data plane (e.g., network switches). SDN can centrally configure routing policies of connected devices via the SDN controller and can provide a global view of the network to SDN applications to help them easily build network-wide business logic. These strengths of SDN have high potentials to address the limitations of existing honeypots and honeynet architecture.

Thus, we propose a novel honeynet architecture to overcome the limitations of existing honeypots and honeynet architecture by leveraging SDN technology. HoneyProxy consists of the proxy module and corresponding SDN application. It takes the form of a reverse proxy to provide improved control over incoming and outgoing traffic while obtaining network configuration via the SDN controller. Malicious traffic from attackers is redistributed to all associated honeypots, and HoneyProxy selects one response from the response queue that does not contain fingerprinting indicator(s). To prevent internal malware propagation, HoneyProxy cooperates with the SDN controller to detect any anomalies within the network. Supporting a dynamic transition between a low-interaction honeypot and a high-interaction honeypot is realized by enabling three types of operating modes.

The contributions we make in this chapter are summarized as follows:

- We propose an SDN-based honeynet architecture called HoneyProxy that consists of a reverse proxy module and of a corresponding SDN application. HoneyProxy tackles important problems in existing honeypots and honeynet architecture: (1) fingerprinting attacks targeting honeypots, (2) internal malware propagation in honeynet, and (3) lack of honeypot transition.

- We propose a connection management engine that supports three operating modes: (1) Transparent Mode, (2) Multicast Mode, and (3) Relay Mode. Based on the decision of HoneyProxy-enabled controller, malicious traffic is processed differently so as to meet our design goals (Section 6.3). To the best of our knowledge, this is the first attempted to introduce flexibility to honeynets.

- We implement a prototype of HoneyProxy that is written in both Python and C. Our experimental results show that the TCP throughput of HoneyProxy achieves the line rate TCP throughput (942 Mbps) using multiple worker processes. The latency incurred by HoneyProxy is in the range of $0.5 - 1.2$ milliseconds on average.

The structure of this chapter is as follows. Section 6.2 discusses the problems we tackle in this chapter. Section 6.3 describes our design strategies along with the architecture and building blocks of our system. In Section 6.4, we introduce the core mechanism of the connection management engine. Section 6.5 focuses on flow programming modules. Implementation details are discussed in Section 6.6 followed by our experimental results (Section 6.7). Section 6.8 introduces related works, and Section 6.9 discusses several important issues. In Section 6.10, we conclude this chapter.

6.2 Problem Statement

Existing honeypots suffer from fingerprinting attacks, and current honeynet architecture suffers from internal malware propagation and a lack of honeypot transition mechanisms.

Vulnerable to fingerprinting attacks. A fundamental drawback of existing honeypots is that they can be easily fingerprinted by attackers. The essential objective of honeypots is to collect as many attacks as possible to learn attacker's behaviors and to discover new types of attacks and malware to provide early warnings to network administrators. However, lack of exposed functionalities and insufficient interactions of honeypots (especially low-interaction honeypots) hinder attackers from probing and exploiting the system. or example, existing ssh honeypots such as kippo [21] can be easily fingerprinted by Linux commands such as `uname -a`. Because kippo does not implement the entire functionality of the `uname` command, it prints out the hard-coded timestamp "Wed Nov 4 20:45:37 UTC 2009" (see Table 6.2). In this way, attackers can instantly identify the presence of honeypots, which reduces the effectiveness of collecting attacker behavior. Existing honeynet *data control* mechanisms do not take into account the fingerprinting attach, thus, removing honeypots themselves must remove emulation artifacts. In other words, neither honeypots nor honeynet architecture is capable to prevent fingerprinting attempts.

Internal propagation of malware. Current honeynet architecture cannot monitor internal traffic because access control mechanisms are enforced at the custom gateway called the *honeywall*. The honeywall monitors incoming and outgoing traffic at a fixed location, and the honeywall acts as a transitional network firewall. Due to the fixed location of a honeywall, monitoring and preventing internal propagation of malware in the honeynet is difficult. In general, honeypots are not to be trusted because attackers are encouraged to actively exploit the honeypots. Therefore, if a honeypot is compromised, it can easily infect other honeypots coexisting in the same network. To prevent these incidents, administrators may want to add host-based protection mechanisms within a machine (e.g., anti-virus, iptables, or sandbox), however host-based solutions are not feasible because the attacker, who is taking control of the honeypot, can circumvent these countermeasures. This is why existing honeynet architecture should be redesigned to better provide network-level protection.

Dynamic transition between low interaction honeypot and high interaction honeypot. Based on the level of interaction with attackers, honeypots are generally categorized into two types: low-interaction honeypots and high-interaction honeypots. Low-interaction honeypots emulate a set of real functionalities and expose (fake) vulnerable and (fake) exploitable services to attackers. The advantages of using them include the ease of deployment and a low possibility of compromise, however they can be easily fingerprinted. In particular, low-interaction honeypots are widely used in the early stage of attacks to collect information on scanning attacks and login attempts (i.e., username/password pairs). High-interaction honeypots implement the majority of the real service (e.g., ssh and http), along with exploitable and/or emulated vulnerabilities. While high-interaction honeypots provide deeper and realistic interactions to attackers, they require sophisticated configurations, high maintenance cost, and high possibility of compromise. Consequently, current honeynet mechanisms totally rely on the capability of each honeypot, resulting in the loss of potential opportunities for maximizing the advantages of both low-interaction honeypots and high-interaction honeypots. For example, we could consider to activate low-interaction honeypots for massive attacks or the login phase of an attack, while high-interaction honeypots provide more interactive attacker actions after a successful login event. Honeybrid [83, 15] strives to facilitate the use of both honeypots by supporting transition mechanisms between a low-interaction honeypot and a high-interaction honeypot. However, this approach does not provide a flexible way to configure when and how to migrate the establish connection from the one to another.

6.3 HoneyProxy: Design and Architecture

HoneyProxy is a novel next-generation honeynet architecture, which leverages Software-Defined Networking. In this section, we describe the key design goals of our

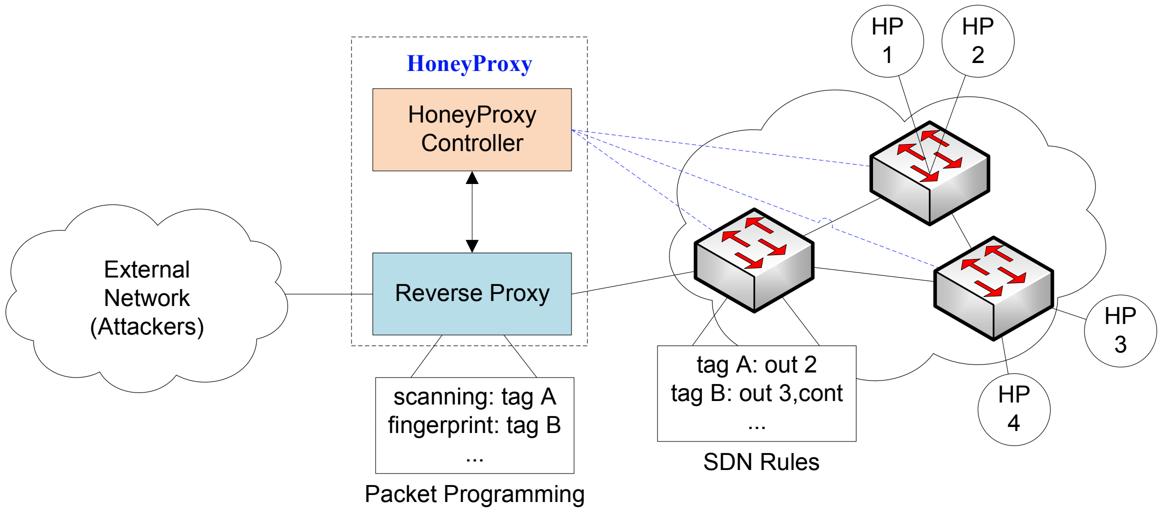


Figure 6.1: Overview of HONEYPROXY.

approach, and we illustrate the architecture of HONEYPROXY along with the detailed building blocks.

6.3.1 Design Goals

To overcome the limitations of existing honeypots and current honeynet architecture, we present an innovative SDN-based honeynet architecture called HONEYPROXY. We define the following design goals that any next-generation honeynet architecture should support:

- **Globalization.** The approach should *globally* monitor all internal traffic to prevent compromised honeypots from propagating malware within the network. Globalization should also include centralized network monitoring and network-wide policy enforcement.
- **Flexibility.** The honeynet architecture must support a smooth transition from a low-interaction honeypot to a high-interaction honeypot or vice versa. This transaction should also be flexible and configurable.
- **Stealthiness.** The approach must be covert, in that it has to hide the existence

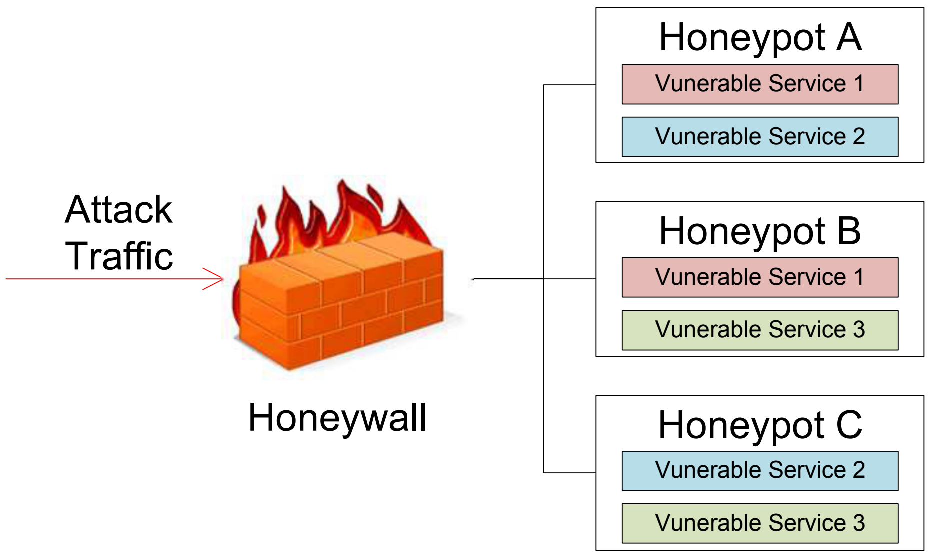
of itself and minimize the exposure of residing honeypots as much as possible. From this point of view, the approach should not incur noticeable delay in conducting the given tasks, as the delay can result in the detection of the honeynet.

- **Generalization.** The approach should be applicable—regardless of the type of residing honeypots or running services. The key question here is related to how the approach can address the redundant services in the network offered by different honeypots.

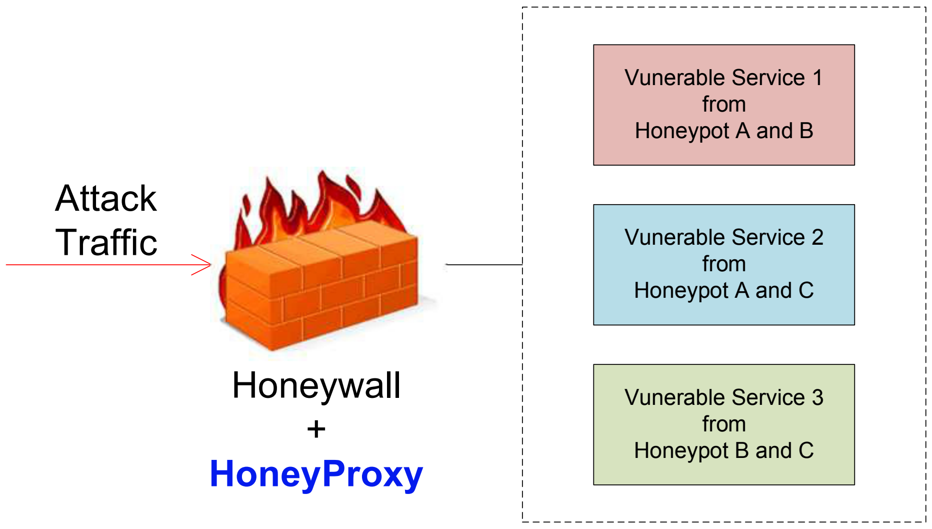
6.3.2 HoneyProxy *Overview*

At a high level, HoneyProxy consists of a proxy module and a HoneyProxy-enabled SDN controller (see Figure 6.1). Multiple honeypots are connected to different switches, and those switches are centrally managed by the HoneyProxy controller. SDN allows network administrators to have centralized control over the entire network by separating the data plane from the control plane. Based on the residing honeypots, the HoneyProxy controller centrally installs the necessary network rules and enforces security rules within the network. On the proxy side, the request sent by the attackers passes through a series of modules in the proxy and is transmitted to a set of relevant honeypots (based on the request).

As shown in Figure 6.1, the proxy pushes a specific type of tagging information inside the packet headers. HoneyProxy controller then creates SDN rules that check the tagging information at the SDN switches to enforce network policies. The proxy module has three operational modes. Based on the decision made by the HoneyProxy controller, the operating mode of the proxy would be reconfigured when necessary (see Section 6.4). The proxy module inspects the payloads of response



(a) Every honeypot runs separately and is managed by itself.


Vunerable Service 1 from Honeypot A and B
Attack Traffic
Vunerable Service 2 from Honeypot A and C
Honeywall + HoneyProxy
Vunerable Service 3 from Honeypot B and C

(b) Honeypots are grouped by vulnerable services using HoneyProxy.

Figure 6.2: HoneyProxy Reshapes the Landscape of Honeynet Architecture Toward One '*BIG*' Honeypot.

to see if it includes any fingerprinting indicators that may expose the presence of honeypots and/or honeynet. Upon discovering an indicator, the proxy module signals to the HoneyProxy controller to take appropriate action, such as changing the proxy mode or updating network configurations, accordingly. Section 6.3.3 provides detailed architecture and building blocks of HoneyProxy.

Figure 6.2 illustrates how HoneyProxy changes the landscape of honeynet architecture. Traditional honeynet architecture runs multiple honeypots behind the custom firewall (honeywall), shown in Figure 6.2a. This architecture allows residing honeypots to run their emulated services, which are possibly redundant with other honeypots as shown in Table 6.1. One consequence of this architecture is that only one vulnerable services is accessible to an attacker at any given time, while the rest are inactive. Because of these reasons, lots of manual configurations of honeypots are necessary to avoid duplicating services, and it can be easily detected since the configurations remain unchanged.

HoneyProxy, shown in Figure 6.2b, allows us to consider the honeypots as one large honeypot running many vulnerable services, which are the union of individual honeypots. It does not require the extra burden of running redundant services across diverse honeypots, because the proxy module distributes requests and selects the best response. In other words, from the attackers' perspective, our honeynet would appear as one large honeypot. To effectively generate multicast messages, the proxy module in HoneyProxy is internally conducting network address translation (NAT [119]) and deep packet inspection (DPI [43]) to interconnect the same services across honeypots.

Our approach has several strengths compared to existing honeynet architecture: (1) It allows attackers to easily access a variety of vulnerable services, which allows greater elicitation of behaviors; (2) Fingerprinting attacks can be mitigated by dynamically selecting the most appropriate response, which reduces fingerprinting and allows for collecting more attack data and learning attacker behavior; (3) SDN controller can globally monitor the entire honeypot network to detect abnormal behaviors of the honeypots (e.g., connection attempts between honeypots) so that internal propagation of malware can easily be prevented at first hand.

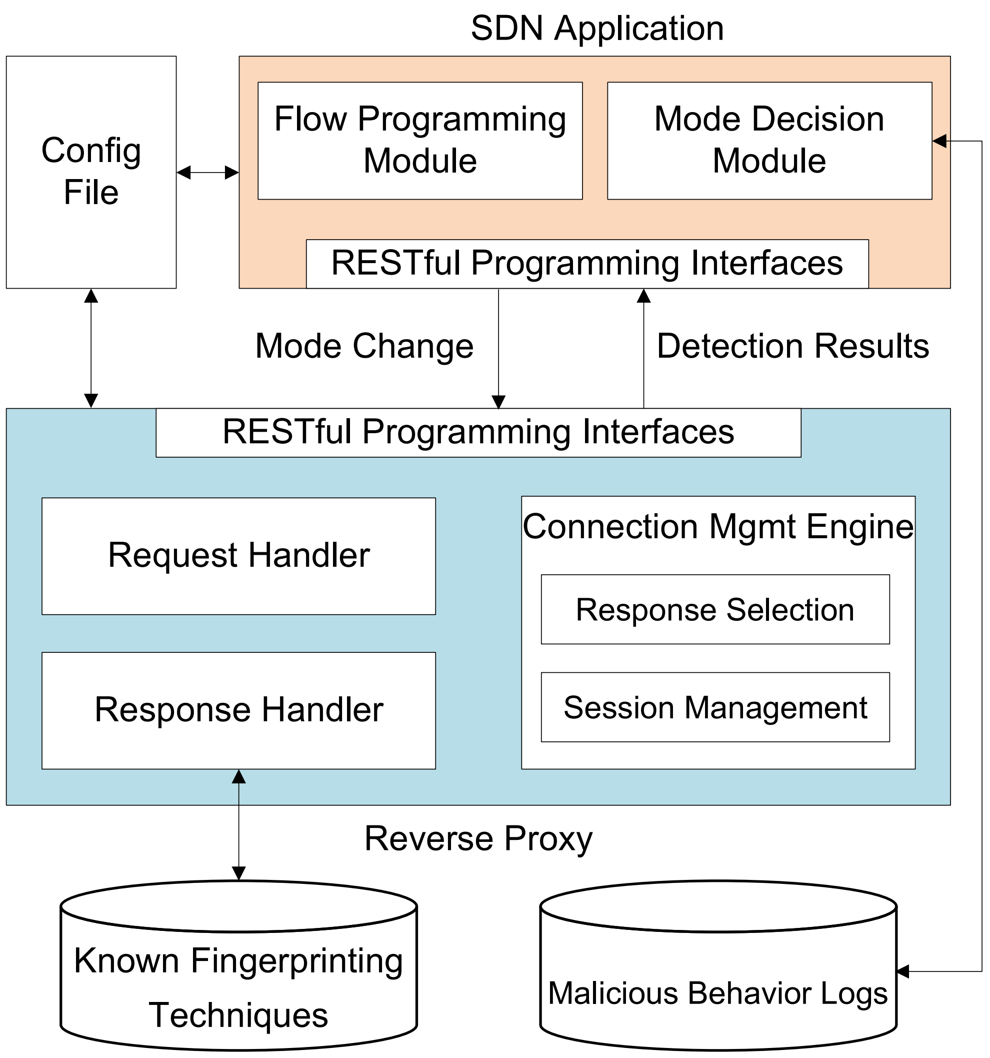


Figure 6.3: HoneyProxy Architecture.

6.3.3 Architecture and Building Blocks

The HoneyProxy architecture is illustrated in Figure 6.3, and HoneyProxy consists of a reverse proxy module and an SDN application. This design separates the concerns of performing network programming and packet processing. The reverse proxy module processes incoming and outgoing traffic using three sub-components: *Request Handler*, *Connection Management Engine*, and *Response Scrubber*. The counterpart, the SDN application, manages network configurations and enforces SDN rules, while monitoring suspicious packets within the network. Detailed building blocks of HoneyProxy are introduced as follows:

Request Handler is responsible for handling the incoming traffic. When a packet arrives at the Request Handler, it first checks the payload to decide if the traffic contains any known fingerprinting attacks, which can fingerprint or compromise the honeypot (see Table [21]). In case of scanning attacks that use OSI L3 or below, the Request Handler adds the *scanning* tag to the packets and directly forwards them so that SDN switches can direct them to IDS running honeypot. Based on this result, the Request Handler signals the Connection Management Engine to perform network address translation (NAT) and deep packet inspection (DPI) to manage the sessions. Thus the request handler mainly monitors the incoming traffic for suspicious attacks and sends the results to the Connection Management Engine.

Connection Management Engine Engine is the core of the reverse proxy module that orchestrates the Request Handler and the Response Handler. The main goal of this engine is to select the response among multiple responses and maintain the sessions to support the three operating modes of HoneyProxy (Section 6.4). The Connection Management Engine also adds tagging information to packet headers of incoming traffic, which allows SDN switches to forward them to the matching destination.

Response Handler is responsible for detecting fingerprinting indicators that may exist in a given response. As defined in Table 6.2, the matching packet would trigger this handler to notify the HoneyProxy controller. Responses from associated honeypots are recorded in the R_Queue, and it waits for the arrival of remaining responses until a size of the queue equals to the number of associated honeypots. If the queue size matches (or by timeout event), the Connection Management Engine selects the most appropriate response from the R_Queue.

Flow Programming Module runs as a part of the SDN application of the

HoneyProxy controller. This module is responsible for notifying the controller to add an SDN rule (i.e., a flow entry) that corresponds to the particular traffic processed by the reverse proxy. The packets marked as *scanning* will be forwarded to the honeypot running intrusion detection systems (e.g., snort [28]). Other packets with the three operating modes would be counted to keep record of attackers' behavior and further utilize them to make a better strategies for attackers.

Mode Decision Module concludes the serving mode of the proxy. For those attackers who conducted fingerprinting attacks are to be mainly served by high-interaction honeypots rather than low-interaction honeypots. Based on several criteria (Section 6.4.3), this modules sends REST request to the proxy to change the operating mode (see Table 6.3).

To achieve the first design goal (globalization), HoneyProxy leverages SDN to globally make a decision on operating modes of HoneyProxy and enforce network and security rules via the SDN controller. HoneyProxy monitors all flows in the network via the SDN controller so that any connection attempts generated by (potentially) compromised honeypots can be logged, monitored, and prevented. To support dynamic transitions across different honeypots in flight (the second design goal), the proxy module in HoneyProxy has *Connection Management Engine* that selects the most appropriate reply from a receiving queue and tracks the state changes of all active connections. In this way, HoneyProxy can also transparently migrate the connection from one honeypot to another. To satisfy the third design goal, stealthiness, HoneyProxy attempts to minimize the performance gaps between different operating modes of HoneyProxy using multi-processing techniques [94]. As elaborated in Section 6.7, latency gaps between different modes are less than a millisecond (< 1 ms), which is not distinguishable when attackers connect over the internet. To

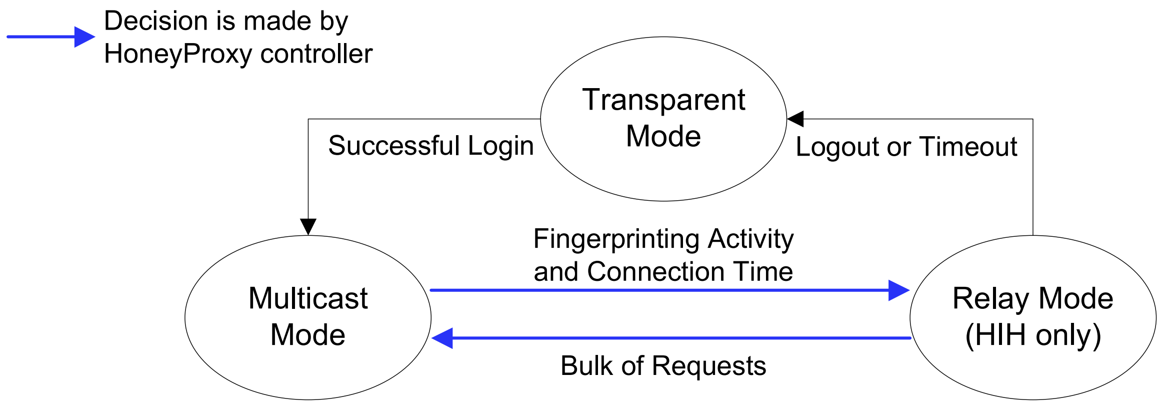


Figure 6.4: Transition Conditions Among Three Operating Modes.

meet the last design goal, generalization, HoneyProxy establishes multiple sockets with the associated honeypots to support L4 or higher in OSI layer. Because vulnerable services are mostly utilizing application layer protocol (L7) except for scanning attacks, HoneyProxy can accommodate most of protocols. For scanning attacks utilizing L3 or below, SDN application of HoneyProxy redirects those packets to one of honeypots that runs an intrusion detection system (which is specifically configured to detect scanning attacks).

6.4 Operating Modes and Connection Management Mechanism

The *Connection Management Engine* supports three operating modes: transparent mode (*T-Mode*), multicast mode (*M-Mode*), and relay mode (*R-Mode*). The purpose of these modes is to efficiently and effectively deliver malicious traffic to relevant honeypots and select the most appropriate reply among multiple responses from honeypots.

6.4.1 Operating Modes

Figure 6.4 illustrates the three operating modes that HoneyProxy supports. Each mode is intended to provide the following features to HoneyProxy:

- **Transparent Mode (*T-Mode*):** T-Mode accounts for the initial stage of attacks such as login trials. Because these attempts are normally launched in an automated manner (e.g., bots or scripts), low-interaction honeypots can effectively handle such attacks. For scalability reason, HoneyProxy only performs network address translation without conducting deep packet inspection.

- **Multicast Mode (*M-Mode*):** Upon the completion of successful login events, HoneyProxy transitions from T-Mode to M-Mode to proactively counteract fingerprinting attacks. In this mode, every incoming payload is delivered to all associated honeypots. However, merely sending multicast messages would not work, because each session has unique session variables such as cookie or shared session key, which are created and managed by the end honeypot. To address this issue, HoneyProxy builds multiple sockets to maintain a set of connections between the honeypots and HoneyProxy and records session data. Section 6.4 elaborates on how HoneyProxy maintains multiple session data and determines the best reply to send to the attacker among multiple responses.

- **Relay Mode (*R-Mode*):** On the receipt of mode change commands issued by the HoneyProxy controller, HoneyProxy transitions from M-Mode to R-Mode or vice versa. R-Mode essentially allows only one connection, which is established by a high-interaction honeypot, to interact with the attacker while other sessions are temporarily suspended. Keeping advanced and motivated attackers connected with low-interaction honeypots is impractical and not feasible. In such case, HoneyProxy is no longer necessarily taking a burden caused by M-Mode (sending multicast messages to associated honeypots). Due to these reasons, R-Mode enhances performance by configuring the rest of the

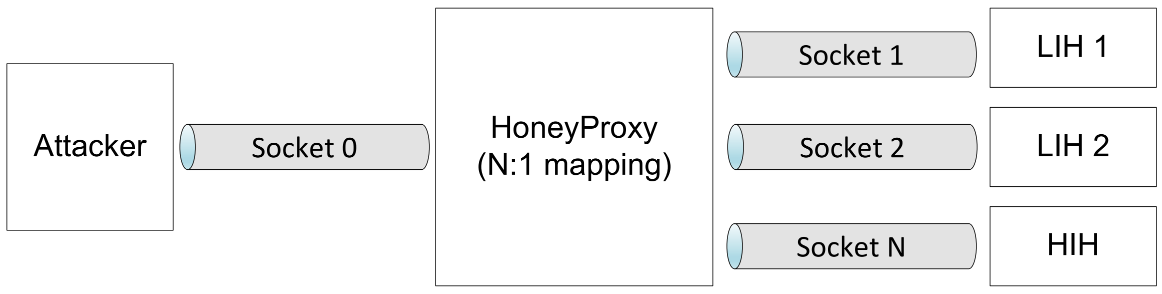
Table 6.2: Example of Known Fingerprinting Indicators for the SSH Honeypot Kippo.

Request		Response	
type	payload	type	payload
exact match	uname -a	exact match	Wed Nov 4 20:45:37 UTC 2009
pattern	.{7,}\n	exact match	bad packet length
exact match	vi	exact match	E558: Terminal entry not found in terminfo
exact match	ifconfig	exact match	HWaddr 00:4c:a8:ab:32:f4

sessions to a standby state. If necessary (e.g., bulk requests that exceed a specified threshold), the controller can transition to M-Mode to let low-interaction honeypots interact with attackers again.

6.4.2 *Response Selection and Session Management*

HoneyProxy maintains known fingerprinting indicators that expose the presence of honeypots or honeynet architecture. For example, Table 6.2 describes several known fingerprints from an ssh honeypot named kippo [21, 6]. Depending on the incoming request, HoneyProxy concludes that fingerprinting attacks are successful if the response contains predefined indicators by an exact or a pattern match. The HoneyProxy controller is notified of the event, to keep a record of attackers' behavior. The proxy module in HoneyProxy discourages sending known fingerprinting responses to the attacker, therefore it selects another response that does not contain the fingerprint(s). Because this task is performed during the deep inspection of packets in flight, the selection decision would not be made by the SDN application but by the proxy module directly. However, fingerprinting traces are reported back to the SDN application of HoneyProxy for later usage by the Mode Decision Module. Note that, finding fingerprints of honeypots or honeynet architecture shown in



```
{SOCKET 0: {“USER”: “AB”, “PASSWD”: “pwd-ab”, “HOST”: “[proxy_ip]:80”, “SESSION”: “x”}}
{SOCKET 1: {“USER”: “AB”, “PASSWD”: “pwd-ab”, “HOST”: “10.0.0.1:80”, “SESSION”: “x”}}
{SOCKET 2: {“USER”: “AB”, “PASSWD”: “pwd-ab”, “HOST”: “10.0.0.2:80”, “SESSION”: “xx”}}
{SOCKET N: {“USER”: “AB”, “PASSWD”: “pwd-ab”, “HOST”: “[ip]:[port]”, “SESSION”: “xxx”}}
```

Figure 6.5: An Illustration of Connection Selection and Session Management.

Table 6.2 is out of our research scope.

HoneyProxy manages multiple sessions in a structured fashion. It establishes a session with an attacker and internally creates a number of sessions (n) with the associated honeypots. Figure 6.5 illustrates a snapshot of active sockets running in M-Mode that maintains $1 : N$ sessions. Socket 0 corresponds to a connection made by the attacker whereas the rest correspond to each of connection with a vulnerable service of N honeypots. Each session is centrally managed by the proxy module of HoneyProxy (Connection Management Engine) including attacker's identity (e.g. a pair of username and password). Because each session data may vary, those information is stored in the table so that HoneyProxy can rewrite the attacker's socket, allowing the vulnerable service to properly accept payloads. Examples of session information include cookies of a HTTP service and a shared session key of an ssh service.

6.4.3 *Transition Criteria of* HoneyProxy *Controller*

To make a reasonable decision of whether or not the attacker needs to be served by a high-interaction honeypot (R-Mode), we develop several criteria, which includes connection duration (δt), fingerprinting attack counts ($\#c$), and previous record of an attacker ($< R_t, R_c >$). The same IP address that has the same identity (username and password pair) is used to locate the previous records. For those who previously accessed our honeynet, we keep records of those attackers into *Malicious Behavior Logs* repository. HoneyProxy looks up the past records from the repository to utilize them to better serve the attackers. The mode of operation function (f_m) for the session (s) is determined by the following equation.

$$f_m(s) = \begin{cases} \text{R-Mode}, & \text{if } \frac{R_t}{R_t+\delta t} \cdot R_c + \frac{\delta t}{R_t+\delta t} \cdot \#c \geq \theta \\ \text{M-Mode}, & \text{otherwise} \end{cases}$$

We configure a threshold value (θ) to balance the workloads of low-interaction honeypots and high-interaction honeypots.

6.5 Flow Programming Mechanism

HoneyProxy takes advantage of the programmability of SDN. The Connection Management Engine classifies the type of incoming packets and adds tagging information to the packets. Using the tags, the SDN controller enforces appropriate actions to process the packets.

6.5.1 Flow Programming

Inspired by tagging techniques [48, 54], we leverage the MPLS field to classify incoming traffic and statically reroute the incoming packets based on the marked tag. The Request Handler first divides incoming traffic into “scanning attacks” (L3 or below) and others that would be further categorized by the Connection Management Engine. The Connection Management Engine is responsible to classify the packet into four types ($T =< S, F, T, M, R >$) such that S belongs to scanning attacks, F

belongs to fingerprinting attempts and T, M, R are an element of T-Mode, M-Mode, and R-Mode, respectively. However, this syntax cannot account for each vulnerable services across diverse honeypots. We thus specify the destined service information ($S =< s, h, d, f \cdots >$) for more accurate analysis of malicious traffic where s, h, d, f stand for the ssh, http, database, and ftp services, respectively. In summary, the total number of SDN rules to process incoming traffic is therefore computed by $|T| \times |S|$. This information is recorded and used by the Mode Decision Module of HoneyProxy to take appropriate actions for attackers.

6.5.2 Blocking Malware Propagation

To block internal propagation of malware within the honeynet, traditional honeynets insert host-based access control rules (e.g., iptables) in each honeypot machine to prevent potential malicious traffic from being generated. However, once a honeypot is compromised, the attacker can circumvent the host-based access control rules. To address this issue, HoneyProxy uses a network-wide monitoring scheme and enforces access control rules via the SDN controller instead of enabling host level protection. To this end, SDN rules are installed in the network to forward outgoing traffic to the specific honeypot that runs the intrusion detection system (IDS), such as snort. In this way, internal traffic between honeypots is also be monitored by an IDS, so it consequently helps network administrators detect internal malware propagation. Note that, the routing path of incoming traffic is not identical to that of outgoing traffic because the incoming packets would pass through IDS. Also, incoming and outgoing flows are physically separated by SDN rules; as all incoming traffic is tagged by the proxy module of HoneyProxy, which is extremely useful for network administrators to manage the network and investigate security breaches.

Table 6.3: RESTful Application Programming Interfaces Between the Reverse Proxy Module (P) and the SDN Application (S).

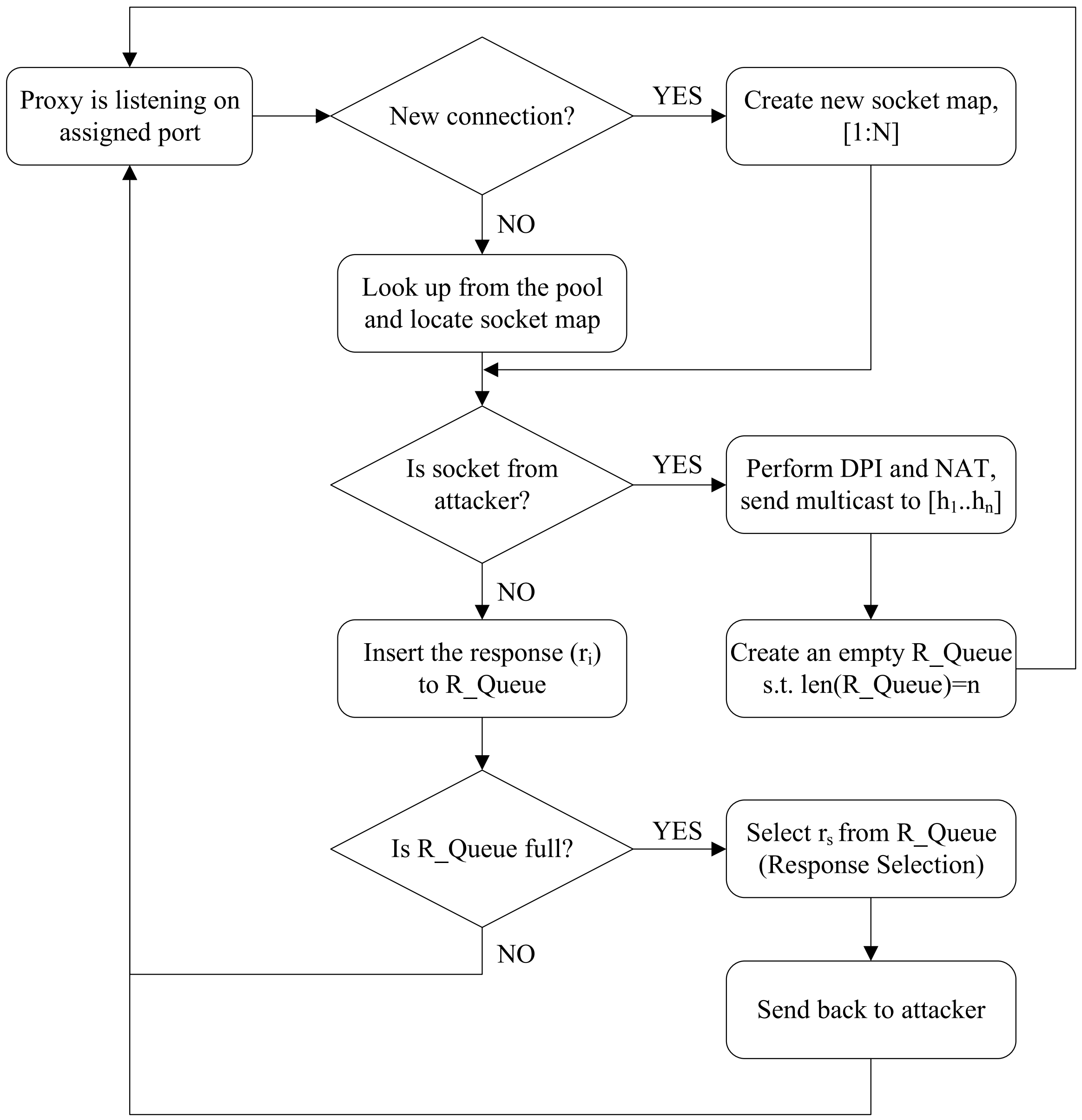
Application Programming Interface	Direction	Type
`/api/addhoneypot/[service]`	S→P	POST
`/api/gethoneypots`	S→P	GET
`/api/runproxy/[service]`	S→P	POST
`/api/killproxy/[service]`	S→P	GET
`/api/sdn/modechange/[id]`	S→P	POST
`/api/sdn/connection/[id]`	P→S	POST
`/api/sdn/connection/[id]/fingerprint`	P→S	POST

6.6 Implementation

We implement HoneyProxy with a commonly used SDN controller, POX [27], along with KVM virtualization infrastructure to run a number of virtual honeypots. For agile development of HoneyProxy, we choose to use the Python language to build the proxy module and the corresponding SDN application of HoneyProxy. As explained in Section 6.3, the Python module has three subcomponents, and it runs a separate RESTful server to communicate with the SDN application over HTTP. The supported RESTful application programming interfaces between the proxy module and corresponding SDN applications are summarized in Table 6.3. To run the proxy instance, the HoneyProxy SDN application should configure each of the honeypots using `/api/addhoneypot/[service]`. Because our proxy module runs at TCP layer (L4), any services built on top of TCP would work, including http, ftp, and database services. It additionally supports transport layer security (TLS) to address https and ssh services. After configuring honeypots, the SDN application can instantiate a number of proxies by sending `/api/runproxy/[service]` request. Each proxy binds to the specified port and serves one vulnerable service per proxy. To enhance the overall performance of a proxy, we implement parallel programming techniques to run multiple worker processes. If a proxy receives fingerprinting indicators, it notifies the server with relevant data, along with associated connection identifier via `/api/sdn/connection/[id]/fingerprint`.

Next, we elaborate the packet processing logic of M-Mode in HoneyProxy [1] . First, a proxy instance listens on an assigned port. We use the Python *select* module to receive payloads from one or more sockets in an asynchronous manner. Upon the receipt of a new payload via a specific socket, the proxy checks the affiliation of this socket. If the socket is not found from the existing socket pool, it means that a new attack has arrived at our honeynet, causing the proxy to create a new socket map (see Figure 6.5). Otherwise HoneyProxy locates the matching socket map from the pool. In case the socket is originated from the attacker, HoneyProxy performs deep packet inspection (DPI) to search for any known fingerprinting attempts (Table 6.2). It then makes a copy of the payloads and performs network address translation (NAT) to send multicast messages to all associated honeypots. Consequently, HoneyProxy creates an empty receiving queue (*R_Queue*) for this socket map where the size of the queue is set to the number of associated honeypots (N). Returning responses from honeypots are inserted into the R_Queue until it becomes full (i.e., all requests are returned). On this event (or a timeout is reached), HoneyProxy chooses the response from the R_Queue to send back to the attacker. Section 6.4.2 discusses how HoneyProxy selects an appropriate response.

[1]Because algorithms for the rest (T-Mode and R-Mode) are relatively straightforward than that of M-Mode, we only introduce its detailed logic for brevity.



Proxy is listening on assigned port
New connection?
YES
Create new socket map, [1:N]
NO
Look up from the pool and locate socket map
Is socket from attacker?
YES
Perform DPI and NAT, send multicast to [h1..hn]
NO
Insert the response (ri) to R_Queue
Create an empty R_Queue s.t. len(R_Queue)=n
Is R_Queue full?
YES
Select rs from R_Queue (Response Selection)
NO
Send back to attacker

Figure 6.6: Packet Processing Logic of M-Mode of HoneyProxy.

6.7 Evaluation

Monitoring and analyzing payloads of incoming and outgoing packets requires a considerable amount of resources. In particular, when the data arrives at the proxy module of HoneyProxy, it must conduct a pair-wise comparison with known fingerprinting attacks, thereby it could be a bottleneck for processing the requests.

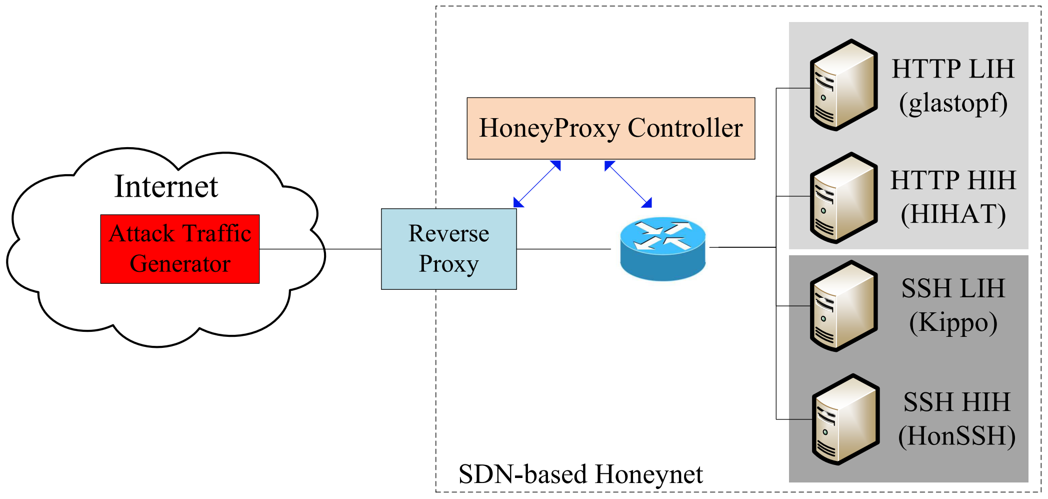


Figure 6.7: Testbed Network Configuration.

The fundamental question, hence, is to quantify the overhead of HoneyProxy and the overhead affect the behaviors of attackers. To this end, we consider three test metrics while conducting the experiments: (1) throughput (Mbits per second), (2) latency (milliseconds), and (3) CPS (connections per second). We first introduce our testbed followed by detailed experimental results.

6.7.1 Test Environment

Figure 6.7 illustrates the testbed setup for the evaluation. Our testbed consists of two physical machines, each of which has Intel Xeon CPUs (E5-2658v3 @ 2.20GHz, 24-cores) and 128GB RAM. One machine runs HoneyProxy (the proxy module and HoneyProxy controller) on CentOS 7.2 (Linux kernel v.3.10.0) and the other runs KVM virtualization infrastructure on Ubuntu 16.04 (Linux kernel v.4.4.0) to emulate a set of honeypots. To create a network of honeypots, we used a software switch (OpenvSwitch v.2.5.0 [25]) that can act as an SDN switch. All incoming traffic destined to our SDN-based honeynet is considered malicious and, therefore, would pass through HoneyProxy. As shown in Figure 6.7, HoneyProxy interconnects the external network (Internet) and honeynet by relaying the packets from the Internet

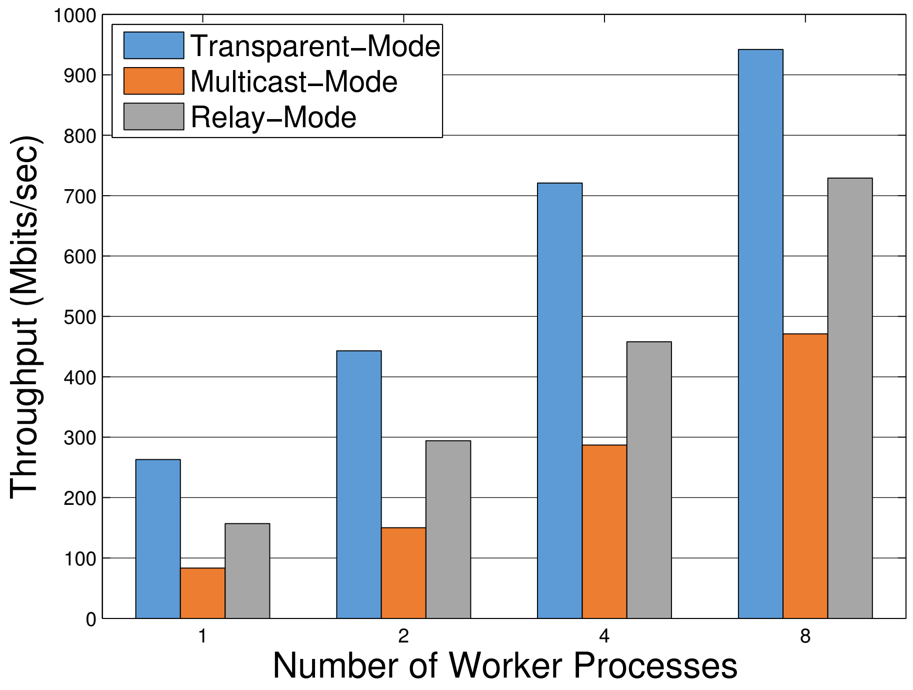


Figure 6.8: TCP Throughput of HoneyProxy with Respect to Three Different Running Modes.

to honeynet or vice versa.

To effectively run the experiments, our testbed was configured to run two representative services: http and ssh. For the http service, we chose a widely deployed low-interaction honeypot, glostopf [12], and HIHAT [14], a high-interaction honeypot. To run the ssh service, kippo [21] was selected as a low-interaction honeypot and HonSSH plays a role of high-interaction honeypot. Each honeypot is configured to have 4 vCPUs along with 4GB RAM, and the link speed for every honeypot was set to maximum 1 Gbps.

6.7.2 Performance of HoneyProxy

As explained in Section 6.6, HoneyProxy utilizes parallel programming techniques to effectively utilize multicore CPUs for scalability. Figure 6.8 shows TCP

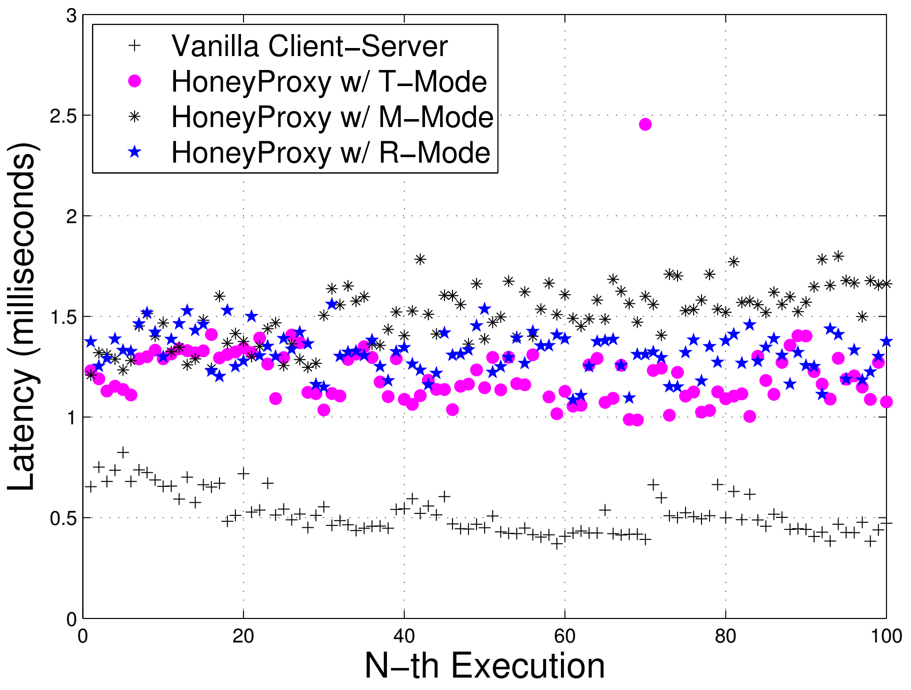


Figure 6.9: File Transmission Latency Incurred by HoneyProxy When the Number of Worker Processes Is Set to 1.

throughput results with respect to different number of worker processes used in HoneyProxy. To measure the test metric, we used iperf [1]. When a single worker process was used, T-Mode achieved 264 Mbits per second (Mbps). M-Mode and R-Mode showed 83 and 157 Mbps, respectively. It is worthwhile to note that M-Mode creates a copy of malicious payloads and sends multicast messages, therefore it requires a considerable amount of resources compared to the others. Overall TCP throughput performance linearly increased with respect to increasing number of worker processes. For example, when eight worker processes were employed, T-Mode scaled up to the near line rate speed (942 Mbps), and we observed that the other modes also showed proportionally increased results (M-Mode 471 Mbps and T-Mode 729 Mbps, respectively).

Although we have obtained the near line rate TCP throughput in T-Mode, we could still enhance the performance of other modes. To this end, we re-implemented the proxy module using C language to investigate how much performance we could achieve. As a result, C-version of HoneyProxy was able to hit the line rate TCP throughput using only one worker process regardless of the selection of modes: 941 Mbps for T-Mode, 940 Mbps for M-Mode, and 934 Mbps for R-Mode, respectively.

We took *latency* as an additional test metric to measure the performance, because latency can show end-to-end responsiveness between attackers and honeypots while *throughput* metric measures the performance with respect to a massive data transaction. To conduct these experiments, we implemented a socket server and client to measure the latency. The client sends a hello message to the server, and the server responds back with an echo message. We measure the round trip time (RTT) for the client to send and receive the message. To obtain ground truth, we first ran the custom server on one of the honeypots and measured the latency without running HoneyProxy. As illustrated in Figure 6.9, RTT took 0.5 milliseconds on average. When HoneyProxy was enabled, the average RTT was observed in the range of 1 and 1.7 milliseconds. Each mode showed 1.24, 1.50, and 1.32 on average, respectively (ordered by T-Mode, M-Mode, and R-Mode). From these results, we could expect 0.5 1.2 milliseconds delay incurred by HoneyProxy. These results are shown in Table 6.4.

Table 6.4 shows latency variations based on differing the number of worker processes and the location of attackers. We repeated the same experiment to measure the latency metric with respect to different number of worker processes. We then conducted the identical experiments over internet to measure how much latency can be attributed to different geolocational network access. Note that the experiments, which were conducted within the same network (the third column), were mainly used to provide the ground information for the latter experiments (the fourth column).

Table 6.4: Latency Variations Based on Different Number of Worker Processes and Network Access Points.

Proxy Mode	Number of Workers	Access within The Network	Access Over The Internet
T-Mode	1	1.24	165.13
	2	1.14	195.71
	4	0.94	168.45
M-Mode	1	1.50	167.81
	2	1.19	167.08
	4	1.15	168.84
R-Mode	1	1.32	166.79
	2	1.15	167.15
	4	1.05	168.14

As expected, the more worker processes were used, the less RTT was observed, regardless of the type of modes. However, if the client (attackers) accessed from the external network (e.g., over internet), RTTs were not distinguishable. In this case, consistent duration time were measured at approximately 166 milliseconds. The time delta between different number of workers observed in the first experiments (within the network) were less than .35 milliseconds, and this pattern was no longer effective due to unforeseeable delays in the internet. In particular, several outliers observed in T-Mode with two worker processes made the average of RTT considerably high. Our experimental results demonstrate that it would be difficult for attackers to identify the operating mode of HoneyProxy remotely.

6.8 Related Work

Software Defined Networking (SDN) is a promising future network technology. When we discuss about applications for security, we look for qualities such as programmability, flexibility, agility, and scalability to easily define and enforce security policies [4, 107]. SDN has the potential to address these requirements of security by providing a global view and centralized control mechanisms to SDN applications. By the same token, SDN can help provide flexibility in monitoring and controlling untrusted traffic within honeynet. We thus leverage the SDN approach in our implementation to centrally monitor and route packets to honeypots, thereby supporting internal traffic monitoring and mitigate the risk of internal malware propagation.

Honeypot farm [17] is an approach which involves deployment of many virtual honeypots in a network. Any malicious traffic directed to the real network will be sent to the dedicated group og honeypots in the network without the knowledge of the attacker. However, this approach only redirects the malicious traffic to the honeypot farm, does not provide any data control mechanisms, and it is also vulnerable to internal propagation of malware. Our prototype makes use of SDN technology, which separates out the control plane to a different entity altogether from the data plane so as to have centralized control over the entire network.

Honeybrid [15] is an architecture which is closely related to our approach which uses connection migration between low-interaction and high-interaction honeypots to take advantage of the functionalities provided by both types of honeypots. However, honeybrid has a few design flaws. In their mechanism, only the first scanning attacks are handled by low-interaction honeypot, and the rest of connection are relayed to a high-interaction honeypot. Therefore,the high-interaction honeypots are active during majority of the connection time. On the other hand, HoneyProxy, based on its analysis of the incoming payloads, will route traffic dynamically to be handled by either low-interaction or high-interaction honeypot at any given time.

Collapsar [71] enables a VM-based honeyfarm architecture, which consists of a group of virtual honeypots. It aims at providing centralized administration, efficient data classification, and distributed view of honeypots. Though this architecture succeeds in providing centralized monitoring of the honeypots, it does not support connection migration between low-interaction and high-interaction honeypots, which becomes important when dealing with a large scale of various attacks to the system.

Our work, HoneyProxy, is greatly influenced by HoneyMix[62], which presents a native SDN-based honeynet architecture. HoneyMix[62] involves deployment of various modules in the SDN controller for dynamic connection selection. However, it does not discuss design issues, implementation, or evaluation details.

6.9 Discussion and Limitations

As most of the internet traffic contains either web traffic (http) or file requests which are carried out using ssh, HoneyProxy aims at detecting attacks that are mainly centered around these protocols, such as fingerprinting attacks, login attempts, and denial of service. Hence, we mainly focused on http and ssh services in mind. However, there may exist a special purpose honeypot that is targeting a very specific service. For example, conpot [9] emulates the industrial control system (ICS), which is also known as supervisory control and data acquisition (SCADA) system. Due to its unique feature, having this honeypot within HoneyProxy architecture does not actually provide an extra benefit over existing honeynet architecture. This is because HoneyProxy basically assumes that multiple honeypots are serving the same service (redundancy of services).

Our implementation only takes advantage of existing honeypots to provide higher quality of data capture and data control, thereby acting as a mediator that lures the attacker and provides a hoax system which emulates the real network characteristics by taking into account diverse functionality present in various existing honeypots. It does not fix the defects and vulnerabilities in individual honeypots [5, 71]. The level of security would depend on the capabilities of honeypots that are present in the honeynet. Our assumption is that by having foolproof honeypots, we could take advantage of the uniqueness of each of them by connecting them to an entity which could centrally monitor and respond to the attacker behind a honeywall. Thus reducing the chances of revelation of the honeynets and hence provide network administrators sufficient information that would help in detection, prevention and securing the network infrastructure.

HoneyProxy fundamentally relies on intrusion detection systems (IDS) for detecting internal malware propagation by configuring way point of internal traffic. It means that an advanced malware might not be filtered by our approach, depending on the capability of IDS. Another technique that can address this limitation is VMI-based abnormality detection [82]. By monitoring the state of virtual machines via hypervisors, we can detect suspicious virtual machines at first hand. However, VMI-based solutions are valid if and only if honeypots are implemented in a virtualized environment whereas HoneyProxy generally works regardless of the type of residing honeypots.

Due to an exponential increase in attacks on the internet system each day, it is very important to build powerful data capture and data control tools and techniques to efficiently provide early warning and detection of threats so that effective security mechanisms can be installed on the network systems. Instead of focusing on securing various ssh and web transactions, many researchers have shifted their study toward expanding applicable area such as Smart Grid, IoT, etc. Honeypots and honeynets are great technologies that can be exploited by the security community. Implementing a competent honeynet system has the potential of being a catalyst to a more secure internet system. This comes with a lot of unknowns and challenges. We hope this chapter triggers active discussion in honeynet researches and security professionals would work toward inventing new techniques to outplay the attackers [50].

6.10 Conclusions

In this chapter, we have articulated the limitations of existing honeypots and honeynet architecture: (1) fingerprinting attacks, (2) internal malware propagation, and (3) honeypots transition. To overcome these shortcomings, we presented an innovated SDN-based honeynet architecture called HoneyProxy as a next generation honeynet. In HoneyProxy, honeypots were grouped by vulnerable services to reshape the landscape of honeynet architecture toward one '*BIG*' honeypot. To this end, we have designed HoneyProxy that consists of the reverse proxy module and corresponding SDN application. We devised a novel Connection Management Engine as a part of the proxy to select the response that does not have fingerprinting indicators and enable dynamic transitions between low-interaction and high-interaction honeypots. To address internal malware propagation, we introduced a flow programming scheme supported by the SDN application of HoneyProxy. We have implemented HoneyProxy using python and C programming languages.Our experimental results have demonstrated that HoneyProxy was able to support the near line rate throughput (942 Mbps) using parallel programming techniques, and latency incurred HoneyProxy was negligible ($0.5 - 1.2$ milliseconds).

Chapter 7

CONCLUSION

In this dissertation, we have proposed a systematic policy management framework for SDN to address several security challenges. To address complex relations across multiple SDN applications, we remove those dependency relations by applying grid-based policy decomposition mechanism. To prevent indirect security violations, we have built automated policy conflicts detection/resolution mechanisms, which are based on analysis of stateless network rules. To remedy statelessness property of existing OpenFlow, we then come up with an innovative stateful monitoring scheme by extending current OpenFlow specifications. To facilitate the wide adoption of SDN and test its capability for improving security of networks, we also proposed an SDN-based next generation honeynet architecture that enables policy-driven network defense mechanisms.

7.1 Dissertation Contributions

This dissertation makes following contributions:

- We proposed a systematic policy management framework for managing policies for SDN that enables policy-driven network defense mechanisms. The framework includes several mechanisms: reliable network rule generation mechanism, stateless policy violation detection/resolution mechanism, and stateful network monitoring scheme.
- We proposed the grid-based policy decomposition mechanism to generate reliable network rules by eliminating dependency relations in an SDN application

and/or across multiple applications. Our decomposition mechanism computes and eliminates intra-app and inter-app dependencies, and enables a secure and efficient network rule generation.

- We proposed FlowGuard that facilitates not only accurate violation detection but also systematic resolution mechanisms based on the analysis of stateless network rules in OpenFlow-based networks. The violation detection approach in FlowGuard detects the indirect security violations by examining flow path space against firewall authorization space, and it is capable of tracking flow paths in the entire network and checking rule dependencies in both flow tables [74] and firewall policies [122]. In addition, we introduced a flexible and effective violation resolution mechanism with the help of four resolution strategies, namely dependency breaking, update rejecting, flow removing, and packet blocking.

- We proposed a stateful network monitoring framework called StateMon that helps SDN support state-aware security applications by maintaining global connection states and providing common APIs to them. To this end, we also proposed the *OpenConnection* protocol, which is a lightweight extension to OpenFlow, and it retains the simple "match-action" programmable feature of OpenFlow to enable a stateful SDN data plane.

- To facilitate the wide adoption of SDN and test its capability for improving security of networks, we have proposed an SDN-based next generation honeynet architecture called HoneyProxy. HoneyProxy consists of the reverse proxy module and corresponding SDN application. We devised a novel Connection Management Engine as a part of the proxy to select the response that does not have fingerprinting indicators and enable dynamic transitions between low-interaction and high-interaction honeypots. To address internal malware propagation, we introduced a flow programming scheme supported by the SDN application of HoneyProxy.

REFERENCES

[1] Iperf. https://iperf.fr/. 5.5.2, 6.7.2

[2] OpenFlow Switch Specification. https://www.opennetworking.org/sdn-resources/onf-specifications/openflow. 4.3.1

[3] Service Function Chaining (SFC) Architecture: http://tools.ietf.org/pdf/draft-ietf-sfc-architecture-02.pdf. URL http://tools.ietf.org/pdf/draft-ietf-sfc-architecture-02.pdf. 5.7

[4] The Killer App For OpenFlow and SDN? Security. http://www.rationalsurvivability.com/blog/2011/10/the-killer-app-for-openflow-and-sdn-security/. 6.8

[5] FootPrinting. https://communities.vmware.com/thread/8117?start=0&tstart=0. 6.9

[6] Black Hat USA 2015 - Breaking Honeypots For Fun And Profit. https://www.youtube.com/watch?v=Pjvr25lMKSY. 6.1, 6.4.2

[7] Apache CloudStack: Open Source Cloud Computing. https://cloudstack.apache.org/. 1.1

[8] Stateful Connection Tracking & Stateful NAT, . http://openvswitch.org/support/ovscon2014/17/1030-conntrack_nat.pdf. 5.1, 5.1, 5.2, 5.6

[9] CONPOT: ICS/SCADA Honeypot, . http://conpot.org/. 6.1, 6.9

[10] Dionaea - carnivore. https://github.com/rep/dionaea. 6.1

[11] Floodlight: Open SDN Controller. http://www.projectfloodlight.org. 3.3, 4.1, 4.2, 5.1

[12] GHihat-Honeypot. http://glastopf.org/. 6.1, 6.7.1

[13] Header Space Library. https://bitbucket.org/peymank/hassel-public. 3.3, 4.5, 4.5.1

[14] Glastopf Honeypot Project Page. http://hihat.sourceforge.net/. 6.1, 6.7.1

[15] Hybrid Honeypot Framework, . http://honeybrid.sourceforge.net/. 6.1, 6.2, 6.8

[16] Developments of the Honeyd Virtual Honeypot, . http://www.honeyd.org/. 6.1

[17] Honeypot Farms, . http://www.symantec.com/connect/articles/honeypot-farms. 6.8

[18] Honeytrap Honeypot, . https://github.com/armedpot/honeytrap. 6.1

[19] Hewlett Packard SDN Dev Center. https://www.hpe.com/us/en/networking/applications.html. 2.2

[20] The Internet Traffic Archive. http://ita.ee.lbl.gov/. 5.5.2

[21] Kippo SSH Honeypot. https://github.com/desaster/kippo. 6.1, 6.1, 6.2, 6.3.3, 6.4.2, 6.7.1

[22] Mininet: An Instant Virtual Network on Your Laptop. http://mininet.org. 3.3, 4.5.1, 5.5.2

[23] openflow, . https://www.opennetworking.org/sdn-resources/openflow. 2.1, 6.1

[24] OpenStack: Open Source Cloud Computing Software, . https://www.openstack.org/. 1.1

[25] Open vSwitch, . htpp://openvswitch.org/. 1.1, 6.7.1

[26] Public PCAP Files for download. http://www.netresec.com/?page=PcapFiles. 5.5.2

[27] POX Wiki-OPen Networking Lab-Confluence. https://openflow.stanford.edu/display/ONL/POX+Wiki. 6.6

[28] Snort.Org. https://www.snort.org/. 6.3.3

[29] The Xen Project, the powerful open source industry standard for virtualization. http://www.xenproject.org/. 1.1

[30] OpenFlow Switch Specification Version 1.5.1 (Protocol version 0x06), December, 2014. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf. 5.7

[31] OpenFlow Switch Specification Version 1.5.1 (Protocol version 0x06), December, 2014. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf. 1.1, 2.2

[32] F. H. Abbasi and R. Harris. Experiences with a generation iii virtual honeynet. In *Telecommunication Networks and Applications Conference (ATNAC), 2009 Australasian*, pages 1–6. IEEE, 2009. 6.1

[33] F. H. Abbasi and R. Harris. Experiences with a generation iii virtual honeynet. In *Telecommunication Networks and Applications Conference (ATNAC), 2009 Australasian*, pages 1–6. IEEE, 2009. 1.1, 6.1

[34] I. Ahmad, S. Namal, M. Ylianttila, and A. Gurtov. Security in software defined networks: A survey. *IEEE Communications Surveys & Tutorials*, 17(4):2317–2346, 2015. 2.2

[35] 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*, pages 37–44. ACM, 2010. 2.3, 4.2

[36] E. Al-Shaer and H. Hamed. Discovery of policy anomalies in distributed firewalls. In *Proceedings of the IEEE INFOCOM 2004 conference*, volume 4, pages 2605–2616. IEEE. 4.2

[37] J. Alfaro, N. Boulahia-Cuppens, and F. Cuppens. Complete analysis of configuration rules to guarantee reliable network security policies. *International Journal of Information Security*, 7(2):103–122, 2008. 4.2

[38] F. Baboescu and G. Varghese. Fast and scalable conflict detection for packet classifiers. *Computer Networks*, 42(6):717–735, 2003. 4.2

[39] K. Benton, L. J. Camp, and C. Small. Openflow vulnerability assessment (poster). In *Proceedings of ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN'13)*, pages 151–152. ACM, 2013. 2.2, 4.2, 5.2, 5.4.3

[40] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. Openstate: programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 44(2):44–51, 2014. 5.1, 5.1, 5.2, 5.6

[41] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014. 5.1, 5.1, 5.2, 5.6

[42] B. Braga, M. Mota, P. Passito, et al. Lightweight ddos flooding attack detection using nox/openflow. In *Proceedings of the 2010 IEEE 35th Conference on Local Computer Networks (LCN'10)*, pages 408–415. IEEE, 2010. 4.2

[43] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral. Deep packet inspection as a service. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 271–282. ACM, 2014. 6.3.2

[44] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A nice way to test openflow applications. 2012. 3.4

[45] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. Software transactional networking: Concurrent and consistent policy composition. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2013. 3.4

[46] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. Sane: a protection architecture for enterprise networks. In *Proceedings of the 15th conference on USENIX Security Symposium*. USENIX Association, 2006. 1.1, 4.1, 5.1

[47] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proceedings of the ACM SIGCOMM 2007 conference*. ACM, 2007. 1.1, 4.1, 5.1

[48] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: a retrospective on evolving sdn. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 85–90. ACM, 2012. 6.5.1

[49] O. M. E. Committee et al. Software-defined networking: The new norm for networks. *ONF White Paper. Palo Alto, US: Open Networking Foundation*, 2012. 3.1

[50] R. C. Dodge JR, R. T. Brown, and D. J. Ragsdale. Honeynet solutions. *AVOIDING FEAR, UNCERTAINTY AND DOUBT*, page 51, 2004. 6.9

[51] M. Dornseif, F. C. Freiling, N. Gedicke, and T. Holz. Design and implementation of the honey-dvd. In *Information Assurance Workshop, 2006 IEEE*, pages 231–238. IEEE, 2006. 6.1

[52] A. El-Atawy, K. Ibrahim, H. Hamed, and E. Al-Shaer. Policy segmentation for intelligent firewall testing. In *Secure Network Protocols, 2005.(NPSec). 1st IEEE ICNP Workshop on*, pages 67–72. IEEE, 2005. 3.4

[53] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Bohatei: Flexible and elastic ddos defense. *arXiv preprint arXiv:1506.08501*, 2015. 1.1, 2.2

[54] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 19–24. ACM, 2013. 2.3, 3.2.4, 3.4, 4.2, 4.6, 5.1, 5.1, 5.2, 5.6, 6.5.1

[55] 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 Conference on Networked Systems Design and Implementation*, pages 533–546. USENIX Association, 2014. 5.1, 5.2

[56] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: An api for application control of sdns. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 327–338. ACM, 2013. 3.1

[57] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. *ACM SIGPLAN Notices*, 46(9):279–291, 2011. 3.1, 3.4, 4.2, 4.6

[58] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 163–174. ACM, 2014. 5.1, 5.2, 5.6

[59] M. G. Gouda and A. X. Liu. A Model of Stateful Firewalls and its Properties. In *International Conference on Dependable Systems and Networks (DSN)*, pages 128–137. IEEE, 2005. 5.1

[60] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4d approach to network control and management. *ACM SIGCOMM Computer Communication Review*, 35(5): 41–54, 2005. 1.1, 4.1, 5.1

[61] R. Guerzoni et al. Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper. In *SDN and OpenFlow World Congress*, 2012. 1.1, 5.7

[62] W. Han, Z. Zhao, A. Doupé, and G.-J. Ahn. Honeymix: Toward sdn-based intelligent honeynet. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 1–6. ACM, 2016. 6.1, 6.8

[63] A. Hari, S. Suri, and G. Parulkar. Detecting and resolving packet filter conflicts. In *Proceedings of the IEEE INFOCOM 2000 conference*, volume 3, pages 1203–1212. Citeseer, 2000. 2.3, 4.2, 4.3.2

[64] D. Hartmeier and A. Systor. Design and Performance of the OpenBSD Stateful Packet Filter (pf). In *USENIX Annual Technical Conference, FREENIX Track*, pages 171–180, 2002. 5.1

[65] S. Hong, L. Xu, H. Wang, and G. Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *NDSS*, 2015. 2.2

[66] H. Hu, G.-J. Ahn, and K. Kulkarni. Detecting and resolving firewall policy anomalies. *IEEE Transactions onDependable and Secure Computing*, 9(3):318–331, 2012. 3.2.3, 3.4

[67] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao. Flowguard: building robust firewalls for software-defined networks. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*, pages 97–102. ACM, 2014. 5.1, 5.2, 5.6

[68] S. Ioannidis, A. Keromytis, S. Bellovin, and J. Smith. Implementing a distributed firewall. In *Proceedings of the 7th ACM conference on Computer and communications security*, page 199. ACM, 2000. 4.1

[69] J. H. Jafarian, E. Al-Shaer, and Q. Duan. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'12)*, pages 127–132. ACM, 2012. 4.2

[70] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013. 1.1, 5.1

[71] X. Jiang, D. Xu, and Y.-M. Wang. Collapsar: A vm-based honeyfarm and reverse honeyfarm architecture for network attack capture and detention. *Journal of Parallel and Distributed Computing*, 66(9):1165–1180, 2006. 6.8, 6.9

[72] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the ”one big switch” abstraction in software-defined networks. *Proceedings of ACM CoNEXT*, 2013. 3.2.2

[73] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012. 2.3, 4.2, 4.4.1, 5.2, 5.5.1, 5.6

[74] 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 conference on Networked Systems Design and Implementation*, pages 99–112. USENIX Association, 2013. 2.2, 2.3, 3.4, 4.1, 4.2, 4.3.2, 4.4.1, 4.5.1, 4.6, 5.2, 5.6, 7.1

[75] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. Veriflow: Verifying network-wide invariants in real time. *ACM SIGCOMM Computer Communication Review*, 42(4):467–472, 2012. 3.4, 4.2, 5.2, 5.6

[76] 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 conference on Networked Systems Design and Implementation*, pages 15–28. USENIX Association, 2013. 2.3, 4.2, 4.4.1

[77] H. Kim, A. Gupta, M. Shahbaz, J. Reich, N. Feamster, and R. Clark. Simpler network configuration with state-based network policies. 2013. 3.4

[78] D. Kreutz, F. Ramos, and P. Verissimo. Towards secure and dependable software-defined networks. In *Proceedings of ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN'13)*, pages 55–60. ACM, 2013. 2.2, 4.2

[79] 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*, 103(1):14–76, 2015. 2.2

[80] M. Krzywinski. Port knocking from the inside out. *SysAdmin Magazine*, 12(6): 12–17, 2003. 5.5.2

[81] C. Leita, V. Pham, O. Thonnard, E. Ramirez-Silva, F. Pouget, E. Kirda, and M. Dacier. The leurre. com project: collecting internet threats information using a worldwide distributed honeynet. In *Information Security Threats Data Collection and Sharing, 2008. WISTDCS'08. WOMBAT Workshop on*, pages 40–57. IEEE, 2008. 6.1

[82] T. K. Lengyel, J. Neumann, S. Maresca, B. D. Payne, and A. Kiayias. Virtual machine introspection in a hybrid honeypot architecture. In *CSET*, 2012. 6.9

[83] T. K. Lengyel, J. Neumann, S. Maresca, and A. Kiayias. Towards hybrid honeynets via virtual machine introspection and cloning. In *Network and System Security*, pages 164–177. Springer, 2013. 6.2

[84] J. Levine, R. LaBella, H. Owen, D. Contis, and B. Culver. The use of honeynets to detect exploited systems across large enterprise networks. In *Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society*, pages 92–99. IEEE, 2003. 6.1

[85] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 290–301, 2011. 2.3, 4.2, 5.2, 5.6

[86] 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*, 38(2):69–74, 2008. 1.1, 3.1, 4.1, 5.1

[87] S. A. Mehdi, J. Khalid, and S. A. Khayam. Revisiting traffic anomaly detection using software defined networking. In *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection (RAID'11)*, pages 161–180. Springer-Verlag, 2011. 4.2

[88] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. Lakshman. Application-aware data plane processing in sdn. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*, pages 13–18. ACM, 2014. 5.1, 5.1, 5.2, 5.6

[89] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 1–14. USENIX Association, 2013. 3.1, 3.2.2, 4, 3.4, 4.2, 4.6

[90] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan. Flow-level state transition as a new switch primitive for sdn. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*, pages 61–66. ACM, 2014. 5.1, 5.1, 5.2, 5.6

[91] M. Mueter, F. Freiling, T. Holz, and J. Matthews. A generic toolkit for converting web applications into high-interaction honeypots. *University of Mannheim*, 280, 2008. 6.1

[92] Y. Nakajima, T. Hibi, H. Takahashi, H. Masutani, K. Shimano, and M. Fukui. Scalable, high-performance, elastic software openflow switch in userspace for wide-area network. *Proc. of USENIX ONS*, 2014. 1.1

[93] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, 16(3):1617–1634, 2014. 2.2

[94] J. Palach. *Parallel Programming with Python*. Packt Publishing Ltd, 2014. 6.3.3

[95] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for openflow networks. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'12)*, August 2012. 2.2, 2.3, 3.1, 4.2, 4.3.2, 5.6

[96] N. Provos. Honeyd-a virtual honeypot daemon. In *10th DFN-CERT Workshop, Hamburg, Germany*, volume 2, page 4, 2003. 6.1, 6.1

[97] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using sdn. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 27–38. ACM, 2013. 3.4

[98] Z. Qian and Z. M. Mao. Off-path tcp sequence number inference attack-how firewall middleboxes reduce security. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 347–361. IEEE, 2012. 5.4.3

[99] Z. Qian, Z. M. Mao, and Y. Xie. Collaborative tcp sequence number inference attack: how to crack sequence number under a second. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 593–604. ACM, 2012. 5.4.3

[100] S. K. Rao. Sdn and its use-cases-nv and nfv. *Network*, 2:H6. 1.1

[101] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 conference*, pages 323–334. ACM, 2012. 4.4.2

[102] C. Roeckl and C. M. Director. Stateful inspection firewalls. *Juniper Networks White Paper*, 2004. 5.1

[103] C. Röpke and T. Holz. Sdn rootkits: Subverting network operating systems of software-defined networks. In *Research in Attacks, Intrusions, and Defenses*, pages 339–356. Springer, 2015. 2.2

[104] E. E. Schultz. A framework for understanding and predicting insider attacks. *Computers & Security*, 21(6):526–531, 2002. 4.1

[105] S. Scott-Hayward, G. O'Callaghan, and S. Sezer. Sdn security: A survey. In *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*, pages 1–7. IEEE, 2013. 2.2

[106] C. Seifert, I. Welch, P. Komisarczuk, et al. Honeyc-the low-interaction client honeypot. *Proceedings of the 2007 NZCSRCS, Waikato University, Hamilton, New Zealand*, 2007. 6.1

[107] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky. Advanced study of sdn/openflow controllers. In *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, page 1. ACM, 2013. 6.8

[108] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 2009. 3.2.4

[109] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. Fresco: Modular composable security services for software-defined networks. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*, February 2013. 2.3, 3.4, 4.2, 4.6

[110] 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 (CCS'13)*, pages 413–424. ACM, 2013. 2.2, 4.2, 5.7

[111] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang. Rosemary: A robust, secure, and high-performance network operating system. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 78–89. ACM, 2014. 2.2

[112] S. Shirali-Shahreza and Y. Ganjali. Flexam: Flexible sampling extension for monitoring and security applications in openflow (poster). In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*, August 2013. 4.6

[113] H. Song. Protocol oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*, August 2013. 4.6

[114] O. S. Specification. Version 1.4.0. *Open Networking Foundation*, 2013. 2.1

[115] L. Spitzner. The honeynet project: Trapping the hackers. *IEEE Security & Privacy*, (2):15–23, 2003. 1.1, 6.1

[116] L. Spitzner. Honeypots: Catching the insider threat. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 170–179. IEEE, 2003. 6.1

[117] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter. Past: Scalable ethernet for data centers. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies (CoNEXT'12)*, pages 49–60. ACM, 2012. 3.2.2, 4.6

[118] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On controller performance in software-defined networks. In *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, volume 54, 2012. 1.1

[119] G. Tsirtsis and P. Srisuresh. Network address translation-protocol translation (nat-pt). Technical report, 2000. 6.3.2

[120] A. Wang, Y. Guo, F. Hao, T. Lakshman, and S. Chen. Umon: Flexible and fine grained traffic monitoring in open vswitch. In *Proceedings of the 11th International Conference on emerging Networking EXperiments and Technologies (CoNEXT'15)*, December 2015. 5.1, 5.1, 5.2, 5.6

[121] X. Wen, Y. Chen, C. Hu, C. Shi, and Y. Wang. Towards a secure controller platform for openflow application (poster). In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*, August 2013. 5

[122] L. Yuan, H. Chen, J. Mai, C. Chuah, Z. Su, P. Mohapatra, and C. Davis. Fireman: A toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy*, page 15, 2006. 2.2, 2.3, 4.1, 4.2, 4.3.2, 4.3.2, 4.4.1, 7.1

[123] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006. 3.4

[124] S. Zhu, J. Bi, C. Sun, C. Wu, and H. Hu. Sdpa: Enhancing stateful forwarding for software-defined networking. In *Proceedings of the 23rd IEEE International Conference on Network Protocols (ICNP 2015)*, pages 10–13, 2015. 5.1, 5.1, 5.2, 5.6

[125] M. Zimmerman, D. Allan, M. Cohn, N. Damouny, C. Kolias, J. Maguire, S. Manning, D. McDysan, E. Roch, and M. Shirazipour. Openflow-enabled sdn and network functions virtualization. *Solution Brief, ONF, Solution Brief sbsdn-nvf-solution. pdf*, 2014. 1.1