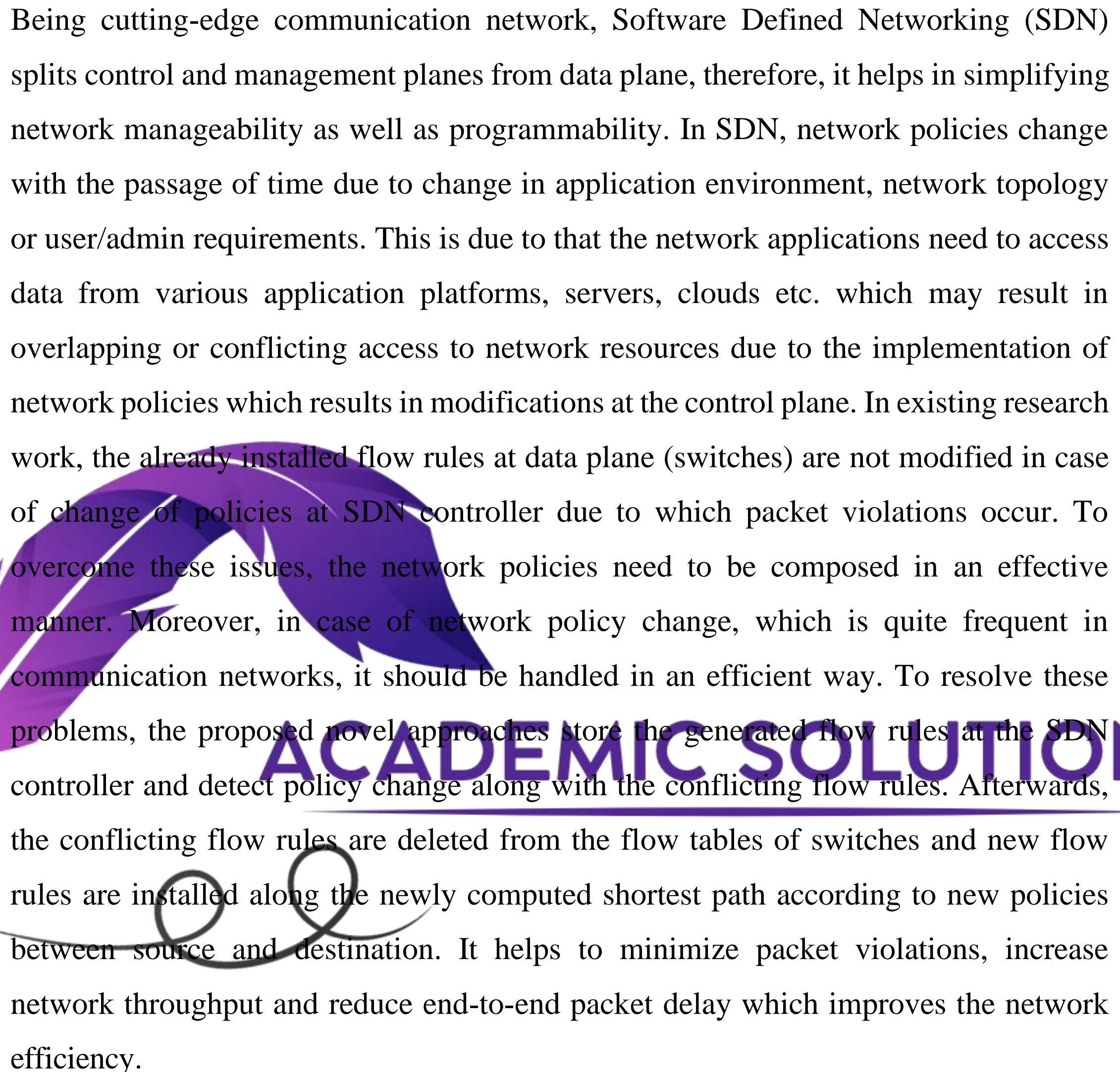
ABSTRACT

Efficient Handling of Network Policy Change in Software Defined Networking



Being cutting-edge communication network, Software Defined Networking (SDN) splits control and management planes from data plane, therefore, it helps in simplifying network manageability as well as programmability. In SDN, network policies change with the passage of time due to change in application environment, network topology or user/admin requirements. This is due to that the network applications need to access data from various application platforms, servers, clouds etc. which may result in overlapping or conflicting access to network resources due to the implementation of network policies which results in modifications at the control plane. In existing research work, the already installed flow rules at data plane (switches) are not modified in case of change of policies at SDN controller due to which packet violations occur. To overcome these issues, the network policies need to be composed in an effective manner. Moreover, in case of network policy change, which is quite frequent in communication networks, it should be handled in an efficient way. To resolve these problems, the proposed novel approaches store the generated flow rules at the SDN controller and detect policy change along with the conflicting flow rules. Afterwards, the conflicting flow rules are deleted from the flow tables of switches and new flow rules are installed along the newly computed shortest path according to new policies between source and destination. It helps to minimize packet violations, increase network throughput and reduce end-to-end packet delay which improves the network efficiency.

In this research work, the network policy change is detected and implemented by two approaches: Matrix-Based Approach, called Efficient Policy Enforcement (EPE) and Graph-Based Approach, called Graph-Based Policy Enforcement (GPE). In EPE, the policy change is detected by comparing different matrices based on Destination IP Address, Switch ID and Forwarding Policy. The matrices are populated as per policy implementation file which consist of a list of policies to allow/deny communication in the network. The EPE is not efficient with respect to access time, cost and space. It is due to that the network policies consist of multiple IP Subnets instead of specific Destination IP Addresses, and switches may have multiple communication links

between source and destination. Similarly, there are also other factors like Protocols, Port Numbers and Service Function Chains (Firewalls, Load Balancers) on which policy can be implemented. Therefore, EPE based policy implementation is not an efficient solution in case of policy change, especially with growing number of matrices due to more factors on which policy is implemented. To handle these problems, the proposed GPE based solution utilizes abstractions to formalize and detect network policies with the help of multi-attributed graphs. Moreover, in this research work, intent-based policies are used for the representation and implementation to resolve the problem of network policy change. In addition, multi-attributed graphs help to reduce access time, cost and space issues to detect and implement policy change mechanism.

For experimentation and analysis of the proposed solutions, different performance metrics are utilized. The proposed solutions are simulated in Mininet Emulator and the results are presented based on different static and dynamic parameters. The results show that the proposed approaches perform better as compared to the existing approaches [52,58] by varying different static and dynamic parameters. The results based on first proposed approach EPE show that the Packet Violation Percentage is reduced up to 99.8%, Successful Packet Delivery Percentage is increased up to 98%, Normalized Overhead Ratio is decreased up to 88.8% and Network Throughput is increased up to 99.8% as compared to existing approaches [52,58]. These results are calculated by varying time instance of policy change, frequency of policy change, data rates, and flow timeout values. In addition, the results based on second proposed approach GPE indicate that Policy Change Detection Time is decreased up to 80%, Packet Violation Percentage is minimized to 82%, Successful Packet Delivery Percentage is increased up to 12%, Normalized Overhead Ratio is decreased up to 13%, Average End-to-End Delay reduced up to 23%, Average Verification Time is reduced up to 77% and Network Throughput is increased up to 14.3% as compared to EPE by varying policy change frequency, data rates and flow timeout values. The results clearly indicate that GPE ultimately increases network performance and efficiency.

ACADEMIC SOLUTIONS

TABLE OF CONTENTS

1. Introduction 1

1.1 Introduction 2

1.2 SDN Advantages 7

1.3 Motivation and Research Objectives 9

1.4 Research Contributions 10

1.5 Thesis Organization... 12

2. Literature Review 14

2.1 Introduction 15

2.2 Network Testing and Verification 16

2.3 Flow Rule Installation Mechanisms 25

2.4 Network Security and Management 33

2.5 Memory Management Studies 40

2.6 SDN Simulators and Emulators 46

2.7 SDN Programming Languages... 49

2.8 SDN Controller Platforms.. 54

2.9 Summary 64

3. Problem Statement.. 65

3.1 Introduction 66

3.2 Network Policy Change at Controller and Flow Rules Installation 66

3.3 Composition and Formalization Issues of Network Policies 70

3.4 Discussion on Cost and Limitations of our Research Work................. 71

4. Proposed Solutions for Change Detection in Network Policies............... 74

4.1 Introduction 75

4.2 Deletion of All Flow Rules 75

4.3 Deletion of Conflicting Flow Rules 75

4.4 Matrix-Based Policy Change Detection and Flow Rules Installation . 78

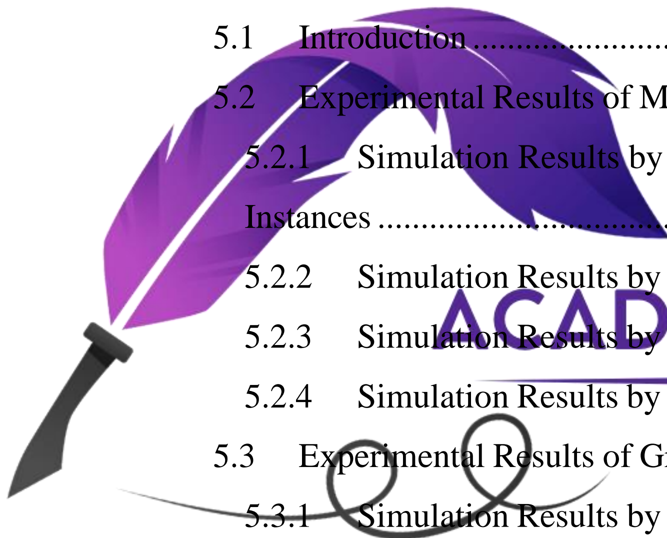
4.4.1 Policy Representation 78

4.4.2 Matrix-Based Network Policy Change Detection 82

4.4.3 Flow Rules Caching... 84

ACADEMIC SOLUTIONS

4.4.4 Complexity of Algorithm 4.1 85
4.5 Graph-Based Policy Change Detection and Flow Rules Installation ... 88
4.5.1 Graph-Based Policy Representation 91
4.5.2 Flow Rules Caching 95
4.5.3 Detecting Policy Change via Graph Matching 96
4.5.4 Comparison of Algorithms Complexities 104
4.5.5 Checking the Flow Rules that Violate New Policies 105
4.5.6 Deleting Flow Rules that Violate New Policies 105
4.5.7 Installing Flow Rules as Per New Policies 105
4.6 Summary 105
5. Results and Discussion 107
5.1 Introduction 108
5.2 Experimental Results of Matrix-Based Proposed Approach 108
5.2.1 Simulation Results by Varying Network Policy at Different Time Instances 110
5.2.2 Simulation Results by Varying Data Rate 113
5.2.3 Simulation Results by Varying Frequency of Policy Change 116
5.2.4 Simulation Results by Varying Timeout Value 119
5.3 Experimental Results of Graph-Based Proposed Approach 122
5.3.1 Simulation Results by Varying Frequency of Policy Change 124
5.3.2 Simulation Results by Varying Packet Transmission Rate 129
5.3.3 Simulation Results by Varying Timeout Value 134
5.3.4 Simulation Results by Varying Number of Switches 139
5.4 Analysis of Proposed Approaches by Utilizing Reactive and Proactive Flow Rule Installation Mechanisms 143
5.4.1 Simulation Results by Varying Frequency of Policy Change 143
5.4.3 Simulation Results by Varying Timeout Value 149
5.5 Summary 152
6. Conclusion and Future Work 153
6.1 Conclusion 154
6.2 Future Work 155


ACADEMIC SOLUTIONS

7. References.. 158

Appendix A.. 178

LIST OF FIGURES

Figure 1.1: SDN System Architecture Showing Data Plane, Control Plane and Management Plane 3
Figure 1.2: SDN Flow Rules Installation Mechanism 6

Figure 2.1: SDN Studies Categorization Hierarchy Comprised of Seven Sections (Section 2.2 to Section 2.8) 15
Figure 2.2: NDB System Architecture [44] Showing Procedure of Identifying Network Wide Invariants by Using Proxy and Collector..........16
Figure 2.3: Veriflow System Architecture [46] Showing Verification of Flow Rules for Network Wide Invariants 17
Figure 2.4: Showcase of a Scenario of SDN [52] without Flow Rule Installation at SW-2 because of Delay between Controller and SW-2 18
Figure 2.5: NICE Working Flow [52] Using Model Checking and Symbolic Execution 19
Figure 2.6: PyResonance Architecture [57] Showing Implementation of State-Based Network Policies via Pyretic Language 20
Figure 2.7: PGA System Architecture [58] Presenting Phenomena of Automatic and Conflict Free Policies via Graph Composer and Veriflow..........21
Figure 2.8: DIFANE Flow Management Architecture [76] (Dashed Lines are Control Messages. Straight Lines are Data Traffic) 28
Figure 2.9: An Overview of Mobi-Flow [77] Presenting Mobility-Aware Adaptive Flow-Rule Placement in Software Defined Access Network..........29
Figure 2.10: BigMac Framework [79] Presenting a Big Switch Abstraction and a Logical Network Plane Specifying Different Forwarding and Management Policies 30
Figure 2.11: Devices Relationships for Flow Rules Categorization [115] to Place Flow Rules in Firewalls..........42

Figure 3.1: OpenFlow Network Scenario of Network Policy Change..........70

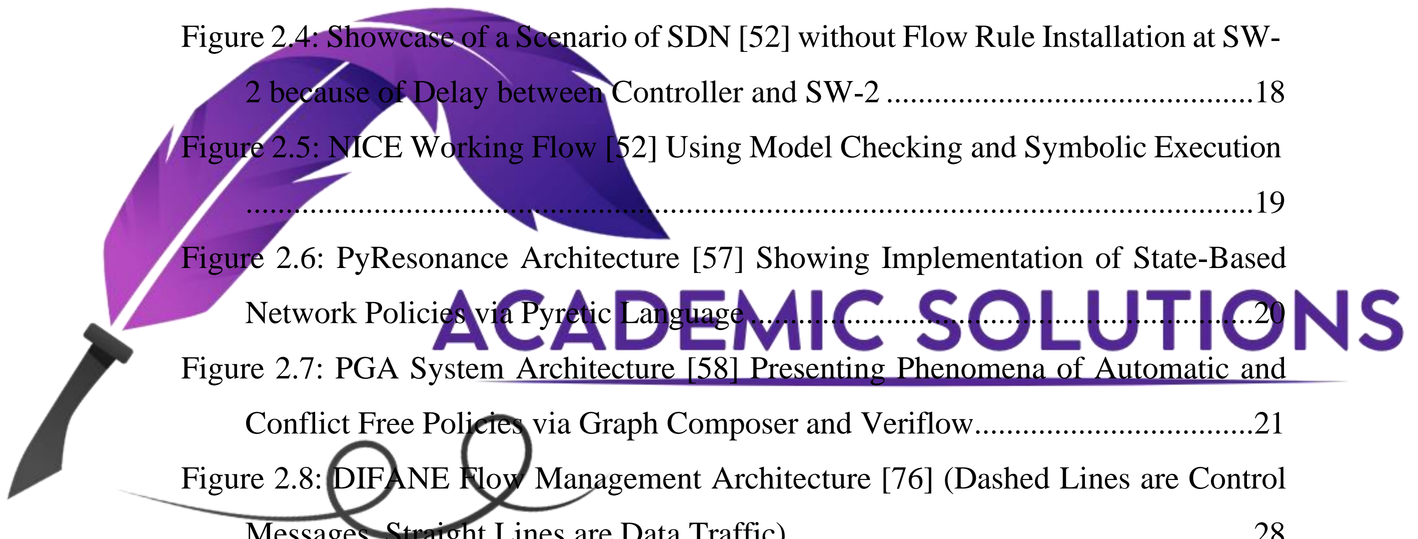


Figure 4.1: OpenFlow Network Scenario in Case of Policy Change and Flow Rules Installation 77
Figure 4.2: Policy Representation File at Time t_1 79
Figure 4.3: Policy Representation File at Time t_2 79
Figure 4.4: Matrices for Detection of Policy Change..82
Figure 4.5: Hash Table Implementation in Case of Matrix-Based Approach..............85
Figure 4.6: Graph-Based Proposed Solution...91
Figure 4.7: Policy Whiteboarding 93
Figure 4.8: Graph Composition..94
Figure 4.9: Input Label Namespace Hierarchy 94
Figure 4.10: Composed Graph based on Label Namespace at Different Time Instances 95
Figure 4.11: Flow Rule Caching based on Graph-Based Approach 96

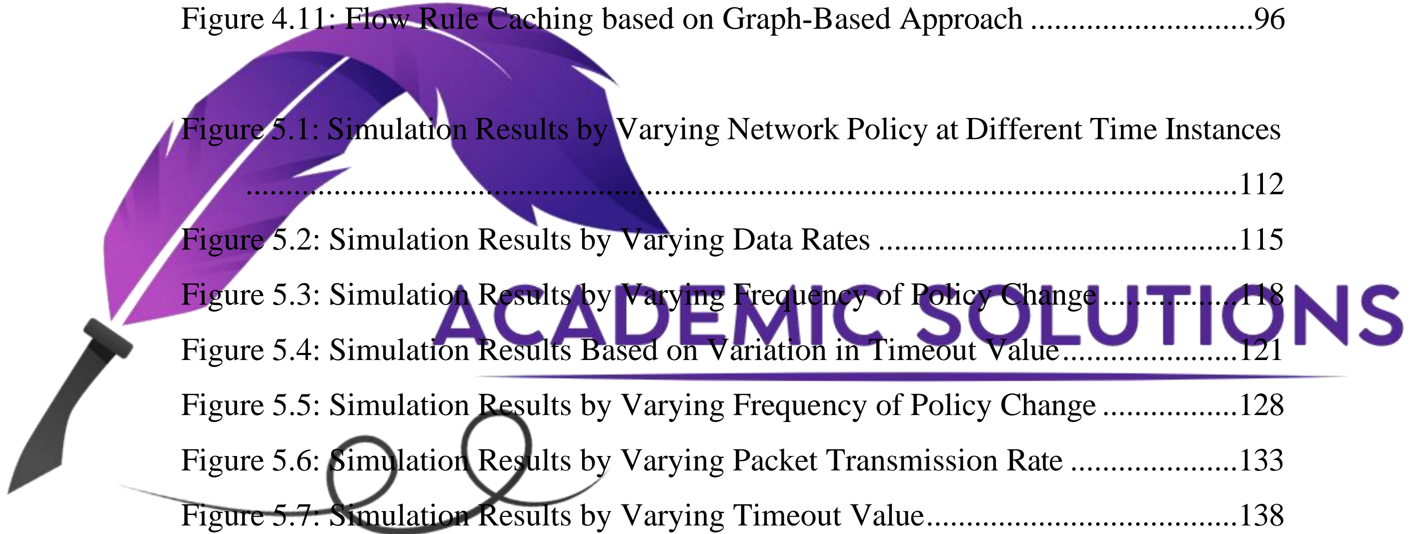


Figure 5.1: Simulation Results by Varying Network Policy at Different Time Instances 112
Figure 5.2: Simulation Results by Varying Data Rates 115
Figure 5.3: Simulation Results by Varying Frequency of Policy Change 118
Figure 5.4: Simulation Results Based on Variation in Timeout Value.....................121
Figure 5.5: Simulation Results by Varying Frequency of Policy Change 128
Figure 5.6: Simulation Results by Varying Packet Transmission Rate 133
Figure 5.7: Simulation Results by Varying Timeout Value......................................138
Figure 5.8: Simulation Results by Varying Number of Switches 142
Figure 5.9: Simulation Results by Varying Frequency of Policy Change 146
Figure 5.10: Simulation Results by Varying Packet Transmission Rate 149
Figure 5.11: Simulation Results by Varying Timeout Value....................................152

LIST OF TABLES

Table 2.1: Summary of Network Testing and Verification Studies 22
Table 2.2: Summary of Flow Rules Installation Mechanisms 31
Table 2.3: Summary of Network Security and Management......................................38
Table 2.4: Summary of Memory Management Studies 44
Table 2.5: Summary of Network Emulators and Simulators 47
Table 2.6: Summary of SDN Programming Languages..51
Table 2.7: Summary of SDN Controller Platforms...59

Table 4.1: Complexity of Matrix-Based Policy Change Detection..............................86
Table 4.2: Complexity of Graph-Based Policy Change Detection 102

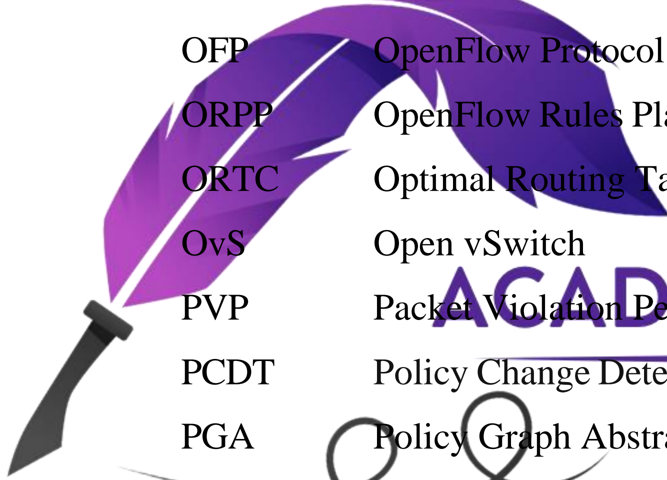
LIST OF ABBREVIATIONS

ACL	Access Control List
AED	Average End-to-End Delay
API	Application Programming Interface
ANN	Artificial Neural Network
AVT	Average Verification Time
CAB	CAching in Buckets
CLI	Command Line Interface
CNN	Convolutional Neural Network
CRM	Customer Relation Management
DoS	Denial of Service
DDoS	Distributed Denial of Service
DOT	Distributed OF Testbed
EC	Equivalence Class
EPE	Efficient Policy Enforcement
FIB	Forwarding Information Base
FIFO	First In First Out
FML	Flow-based Management Language
FPB	Flow-granularity Packet-in Buffer
FSM	Finite State Machine
FTRS	Flow Table Reduction Scheme
GDB	GNU Debugger
GPE	Graph-Based Policy Enforcement
HSA	Header Space Analysis
ILP	Integer Linear Problem
IP	Internet Protocol
IoT	Internet of Things
IT	Information Technology

IVS	Indigo Virtual Switch
KNN	K-Nearest Neighbors
LB	Load Balancing
LRU	Least Recently Used
MAC	Medium Access Control
MILP	Mixed Integer Linear Programming
NAT	Network Address Translation
NBI	Northbound Interface
NDB	Network Debugger
NICE	No bug In Controller Execution
NOH	Normalized Overhead
OFP	OpenFlow Protocol
ORPP	OpenFlow Rules Placement Problem
ORTC	Optimal Routing Table Constructor
OvS	Open vSwitch
PVP	Packet Violation Percentage
PCDT	Policy Change Detection Time
PGA	Policy Graph Abstraction
RFCs	Request of Comments
SDN	Software Defined Networking
SBI	Southbound Interface
SPD	Successful Packet Delivery
SRAM	Static Random Access Memory
SRV	Switch-based Rules Verification

SVM Support Vector Machine

SVM	Support Vector Machine
TCAM	Ternary Content-Addressable Memory
TCP	Transmission Control Protocol
UI	User Interface



VLAN	Virtual Local Area Network
ViNO	Virtual Network Overlay
VM	Virtual Machine

Chapter 1

Introduction

1.1 Introduction

Traditional networks are distributed in nature where control, management and data planes are tightly and vertically coupled in each forwarding device. This reduces network flexibility, hindering innovation and evolution of the networking infrastructure. Although, the traditional networks are broadly adopted, however, these are complex and hard to manage [1]. The forwarding of data packets in traditional networks is controlled by control plane (i.e. routing protocols) or by installing network policies like Access Control Lists (ACLs) on the interfaces of data plane (routers/switches). In addition, the network policies are manually installed at forwarding devices by using low level commands which are often device specific. SDN [2-6] is an emerging network architecture which helps to solve the limitations of traditional networking by separating both control and management planes from the data plane of the forwarding devices (routers and switches). It shifts control functionalities into centralized logical entity called SDN Controller which acts as network brain and leaves data plane functionalities (forwarding table) in the switches. The management plane specifies network applications such as network policies, network monitoring, load balancing etc. which are implemented by the network administrator based on application environment and user's requirements. To interact with SDN planes, there are two standardized application programming interfaces, these are Southbound Interface (SBI) and Northbound Interface (NBI). The SBI is used for communication between data plane and control plane. The popular OpenFlow Protocol (OFP) is utilized for this purpose which provides secure channel in order to communicate between these planes. The NBI is used for communication between control plane and management plane. The SDN system architecture is shown in Figure 1.1.

ACADEMIC SOLUTIONS

SDN has many advantages over traditional networks due to the less maintenance, ease of management and implementation of network policies [7, 8]. As SDN allows vendor independent control over the entire network from the centralized controller, therefore, network management is greatly simplified. Moreover, forwarding devices (switches) become simplified as network intelligence is shifted to the controller, thus, these devices are left with very simple functionalities as they only need to act according to the instructions from controller and do not require understanding and processing heterogeneous algorithms and protocols. For example, they do not need to learn source

addresses, Virtual Local Area Network (VLAN) support, source address spoofing and flow level statistics. In case of layer three switch, it does not need to understand and process routing protocols, network policies, Network Address Translation (NAT) and port replication. In addition, the forwarding devices also help controller for route computations and link/node monitoring along with other tasks like network management and diagnostics [9].

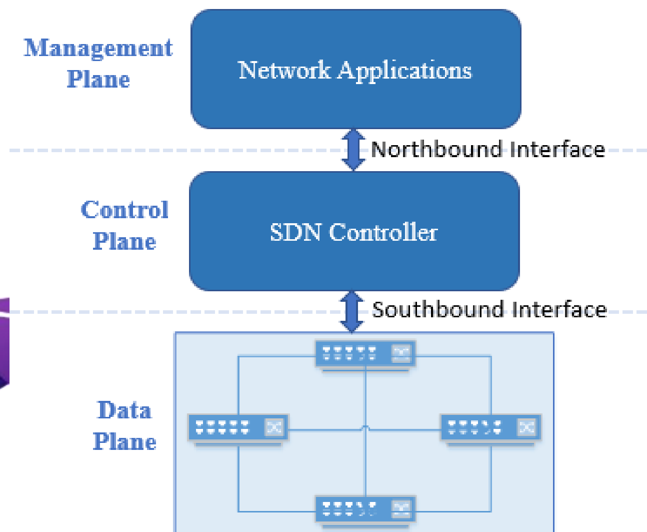


Figure 1.1: SDN System Architecture Showing Data Plane, Control Plane and Management Plane

ACADEMIC SOLUTIONS

The computer networks are mainly configured on the basis of routing protocols and network policies. In SDN, the network policies are specified at the controller which computes the shortest path for data packets between source and destination. In this research work, network policies are implemented based on destination Internet Protocol (IP) addresses and abstract level (high level source and destination names, protocols and destination port numbers). For example, based on destination IP address, a policy is represented via pyretic language as (match (switch = SW1) & match (dstip = '10.0.0.3')>>fwd(2)). This policy states that when a packet is received at switch-1 "SW1" whose destination IP address is "10.0.0.3", will be forwarded to Port 2 of the switch. Similarly, a high-level policy expressed as "Employees (Faculty and Staff) has access to the Servers via Transmission Control Protocol (TCP) and Destination Port Numbers 20, 25, 80 through Firewall (FW) Service Function Chain". This policy is expressed via policy whiteboarding and implemented with the help of Policy Graph

Abstraction (PGA). This policy is more complex than the destination IP address based policy, as it includes multiple parameters, like, high level names "Employees", Protocol "TCP", Port Numbers "20, 25, 80" and Service Function Chain "Firewall". Such policies are implemented at the central controller which installs flow rules at the switches along the shortest path as per policy instructions. Due to this centralization, the network management and policy implementation is greatly simplified in SDN.

In communication networks, the network policies often change to restrict or allow communication between hosts, applications, network nodes or due to change in network topology. In addition, these policies help in implementing access control which restricts/allows communication as per source/destination IP addresses/Medium Access Control (MAC) addresses, destination port numbers, protocols, service function chains, etc. to the users/applications [10]. The high-level languages, Pyretic [11], Frenetic [12] and Maple [13] help to specify these policies as per the application environment. These languages provide parallel and sequential composition operators for efficient implementation of policies. Multiple policies are allowed by these composition operators for the communication of specific set of packets. Also, one policy is allowed to process packets after another policy correspondingly. Moreover, these allow programmers to utilize standard programming language to design an arbitrary, centralized algorithm "ACL policy" to decide the behaviors of the whole network. In addition, these also provide an abstraction for the programmer defined policy which is executed on each inbound network packet. Furthermore, these languages also address the problem of network policy implementation based on abstraction of topology and packet model. So, it becomes beneficial for programmers who can build large and refined network applications out of small and independent modules. However, these languages do not specify the mechanism of network policy change handling and flow rules installation/deletion process.

ACADEMIC SOLUTIONS

In SDN, the network policies are implemented at controller and based on these policies, flow rules are computed and installed at switches. This is still a big challenge to translate a conflict free high-level policy into sets of flow rules on distributed individual switches. Additionally, these policies change rapidly in communication networks due to the change in application environment, network topology, user requirements etc. This rapid changing in policies results in packet violations and network inefficiencies

due to the existing flow rules that are installed at the switches based on old network policies. When the packet is received at a switch and it does not have flow rule entry for the packet in its flow table, then the switch forwards a digest packet to the controller via a secure channel using OpenFlow Protocol (OFP) [14]. On receiving the digest packet, the controller computes the best path as per topology and network policies and installs flow rules accordingly on the computed best path. Afterwards, the switch forwards the packet as per flow rule entry from the controller. The switch also keeps these flow rules in its flow table so that subsequent similar packets are forwarded without contacting the controller. By storing flow rules at the flow tables, it reduces load at the controller as well as the end-to-end packet delay.

For example, as shown in Figure 1.2, Host-A (connected with Switch-A) wants to send data to Host-B (connected with Switch-B), when the first packet, say PKT-1 reaches at Switch-A, it saves that packet in its buffer and sends digest packet to the controller due to unavailability of respective flow rule in its flow table. After installation of flow rules from the controller, Switch-A forwards data packets. Additionally, Switches (Switch-A, Switch-B) store the flow rules received from the controller in flow tables so that the subsequent packets can be forwarded without intervening the controller. These flow rules remain in flow tables of switches till timeout value expires due to inactivity. Moreover, the value of timeout gets refreshed when packets of the flow are forwarded at the switch. Normally, two types of timeouts are set, these are *soft timeout* and *hard timeout*. *Soft timeout* is the number of seconds after which the flow rule is removed from flow table of switch in case if it is not used for specific number of seconds. This indicates that there is no packet matching for those specific number of seconds. *Hard timeout* is the number of seconds after which a flow rule must be removed from the flow table of switch even if it is used by the arriving packets belonging to the flow rule at the switch [15]. The default values of timeouts vary depending on target scenario. Moreover, timeout value can be set different for different flow rules depending upon controller specification [16] and can be set by the administrator as per requirements and specifications of a certain application.

ACADEMIC SOLUTIONS

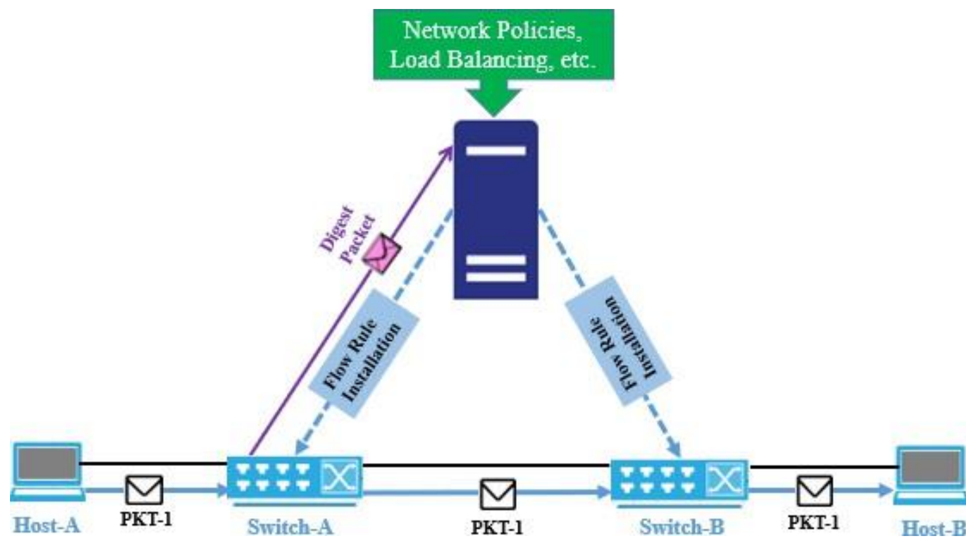


Figure 1.2: SDN Flow Rules Installation Mechanism

There are three flow rule installation mechanisms in SDN. These are, Reactive, Proactive and Hybrid. In reactive mechanism, when a packet is received at switch, it lookup its flow rule in the flow table to initiate forwarding process. In case of flow rule matching, the respective packet is forwarded as per flow entry. However, in case of non-matching, it creates a packet-in message and sends digest packet to the controller. The controller reacts to packet-in message via consulting network topology and network policies which matches to that flow. It computes and installs flow rule along the path between source and destination via packet-out messages. All subsequent packets of a respective flow will follow same path without intervention of controller. In this approach, only required flow rules are installed as per request from communication hosts, so, it helps to reduce load in flow tables of data plane. This in turn efficiently utilizes Ternary Content-Addressable Memory (TCAM) resources. In Proactive approach, the flow rules are pre-populated, that is, before the first packet of a flow arrives at a switch based on network policies and network topology [17]. This approach minimizes the setup delay of flow rules and reduces the overall number of signaling messages because flow rules are already installed for them. When flow rules and their actions are predefined at switches even before the packet arrives, resultantly, all required packets are sent or forwarded to nodes without intervention of controller. Forwarding is done by just matching flow rules in the flow tables, as these are already there, therefore, it saves huge amount of time. However, the TCAM resources of switches are not efficiently utilized due to the installation of those flow rules for which communication is not desired. Hybrid approach consists of both proactive and reactive

ACADEMIC SOLUTIONS

flow rule installations. This approach is flexible in the sense that it includes best characteristics of both proactive and reactive approaches. It offers flexibility, robustness and helps to reduce communication delays.

To resolve the identified research problems of packet violations and network inefficiencies due to change in network policies, two novel approaches are proposed to detect and implement network policy change at control plane. These approaches are Matrix-Based and Graph-Based Policy Change Detection and Implementation. In case of detection of policy change, controller computes shortest path and flow rules as per changed policies. In addition, these newly computed flow rules are installed along the path at switches and old flow rules which conflict with the changed policies are deleted. The simulation results show that the proposed approaches help to minimize packet violations, increase successful packet delivery percentage and reduce end-to-end packet delay which resultantly increase the network efficiency.

1.2 SDN Advantages

SDN has numerous advantages as compared to traditional networking and few of those are mentioned below:

ACADEMIC SOLUTIONS

- **SDN Centralized Management and Control**
  SDN's centralized management and control of networking devices helps to reduce complexity.
- **Directly Programable**
  As network control plane is separated from data plane, so it is directly programable.
- **Easier Network Management and Automation**
  It offers easier network management and automation via common Application Programming Interface (API) to program the applications since the abstractions are provided by the control platform.
- **Rapid Innovation**
  It has rapid innovation to deliver new network capabilities and services without need to configure individual devices or wait for vendor releases.

- **Programmability**

  The network is programmable with the help of network applications which are installed on top of SDN controller to provide vendor independence.

- **Reliability**

  The network reliability is increased due to the centralization and automation of network policy enforcement and minimization of errors. Its smart routing help to mitigate the problem of service unavailability and transmission delay due to link failures.

- **Flexibility**

  SDN is a dynamic and flexible network architecture which protects existing investments while future-proofing the network.

- **Flow Rules Based Forwarding**

  Forwarding decisions are on the basis of flow rules instead of destination-based addresses which broadly implements flow rule matching and action criteria. In this regard, all packets of a flow receive identical network policies at forwarding devices.

- **Network Policy Implementation**

  It allows network administrators to apply network policies at more granular level, including the session, user, device, and application levels in a highly abstracted automated fashion.

ACADEMIC SOLUTIONS

- **Usability**

  It provides better user experience by centralizing network control and making state information available to higher-level applications as it can adopt dynamic user needs easily [18-21].

- **Security**

  It provides centralized security control which improves network visibility through security management. In addition, it offers robust control over network infrastructure to develop efficient and effective security mechanisms [22,23] to detect and prevent security attacks [24,25]. Moreover, softwarized Policy implementation architecture can enhance the security among the autonomous systems which have least human interactions and consequently it can mitigate the security risk of interdomains communication.

1.3 Motivation and Research Objectives

Traditional networks are complex because all the functionalities of data plane, control plane and management plane are vertically and tightly implemented within each forwarding device. These three planes are implemented according to Request of Comments (RFCs) and each vendor implements the desired functionalities as per RFCs in the forwarding devices in their own way. Therefore, in these networks the network administrator has to configure each device separately by using low-level and vendor-specific commands. Due to the manual configurations, many problems occur, for example, misconfigurations, delay in implementing new network policies etc. In [26] it is reported that 62% of network downtime in multi-vendor networks comes from human-error and 80% of Information Technology (IT) budget is spent on maintenance and operations. SDN helps to resolve these issues of complexity and management by separating control and management planes from the data plane. There are different research works as discussed in Chapter 2 (Literature Review) which help to resolve such kind of issues. We categorized these research works based on network wide invariants, memory management, security, programming languages and controller platforms. All these works help to implement SDN in an efficient way by detecting network faults, improving memory utilization of devices, resolving security issues and providing application platforms to develop SDN applications. These also help network administrators to detect and recover network faults which results in improving the network recovery in case of failures.

ACADEMIC SOLUTIONS

Currently, SDN implementation is getting much popularity due to its ease of management and centralization. Google and Facebook are already using SDN for Wide Area Network (WAN) interconnectivity and to improve hyperscale data center connectivity [27-29]. In such kind of data centers and campus area networking, the network policies change rapidly to restrict/allow communication between hosts, change in network topology, application/user requirements etc. There is currently no research carried out to efficiently handle network policy change in software defined networking which is quite frequent in data center and campus area networking.

In order to resolve the identified research problem, this research work aimed to:

- analyze and discuss the existing research works along with their limitations to the research problem in Chapter 2
- formalize the identified research problem in Chapter 3 and represent network policies in a high-level view in Chapter 4 so that it becomes more understandable
- automatically detect policy change at the controller in Chapter 4
- identify conflicting flow rules as per changed policies along with the deletion of exiting conflicting flow rules Chapter 4
- cache flow rules at controller in hash table data structure in Chapter 4 so that it become easy to track conflicting flow rules in case of network policy change
- compute shortest path between source and destination and install new flow rules at data plane along the shortest path in Chapter 4
- evaluate performance of the proposed approaches as compared to the existing approaches to verify network performance and efficiency in Chapter 5.

1.4 Research Contributions

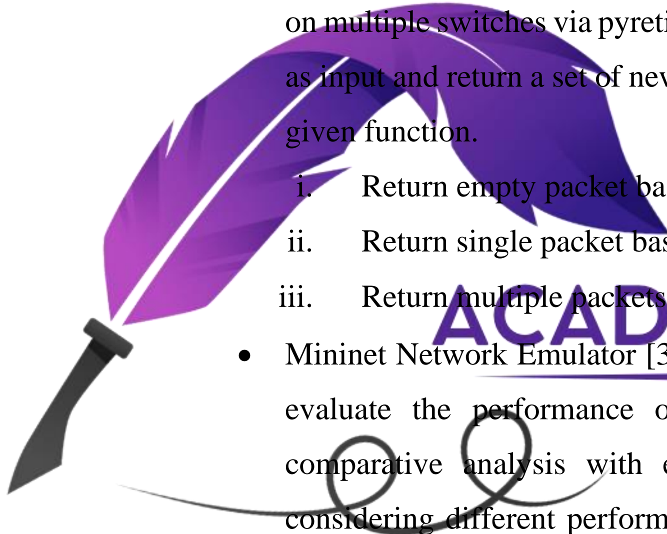
ACADEMIC SOLUTIONS

This thesis is about implementation of SDN in an effective manner by efficiently handling network policy change at SDN controller in order to minimize packet violations and network inefficiencies in addition to increase network throughput. In existing research, such kind of network policy change handling is not implemented, therefore, we propose mechanisms which efficiently detect policy change, delete already installed conflicting flow rules and install new flow rules at data plane as per new policies. We have contributed in different ways and these are enlisted below:

- To the best of our knowledge, we are the first to deal with the network problem of policy change at controller. In this regard, we have presented analysis of state-of-the-art existing approaches in Chapter 2 which are related to research problem and found no such work which deals with this problem of efficient handling of network policy change and flow rules management.
- To resolve the identified research problem of packet violations and network inefficiencies due to change in network policies, two novel approaches are proposed, and these are: Matrix-Based (Efficient Policy Enforcement (EPE))

and Graph-Based (Graph-Based Policy Enforcement (GPE)) to detect and implement the network policy change at controller.

- In case of policy change detection, the flow rules are computed accordingly and cached at controller in a hash table data structure. For this purpose, we used Consistent Hash Function because every time when function is executed, it produces same output for an input, therefore, it helps to accurately find a certain flow rule on a specific switch.
- The computed flow rules are installed by the controller at switches along the shortest path from the controller and old flow rules are deleted from switches which conflict with the new policies between source and destination.
- We have used pyretic language [11] to define network policies via high-level abstractions rather than low-level commands. It helps to implement flow rules on multiple switches via pyretic policy based abstract functions that take packet as input and return a set of new packets. This return function depends upon the given function.
  i. Return empty packet based on the drop.
  ii. Return single packet based on forwarding to new single location.
  iii. Return multiple packets based on multicasting.

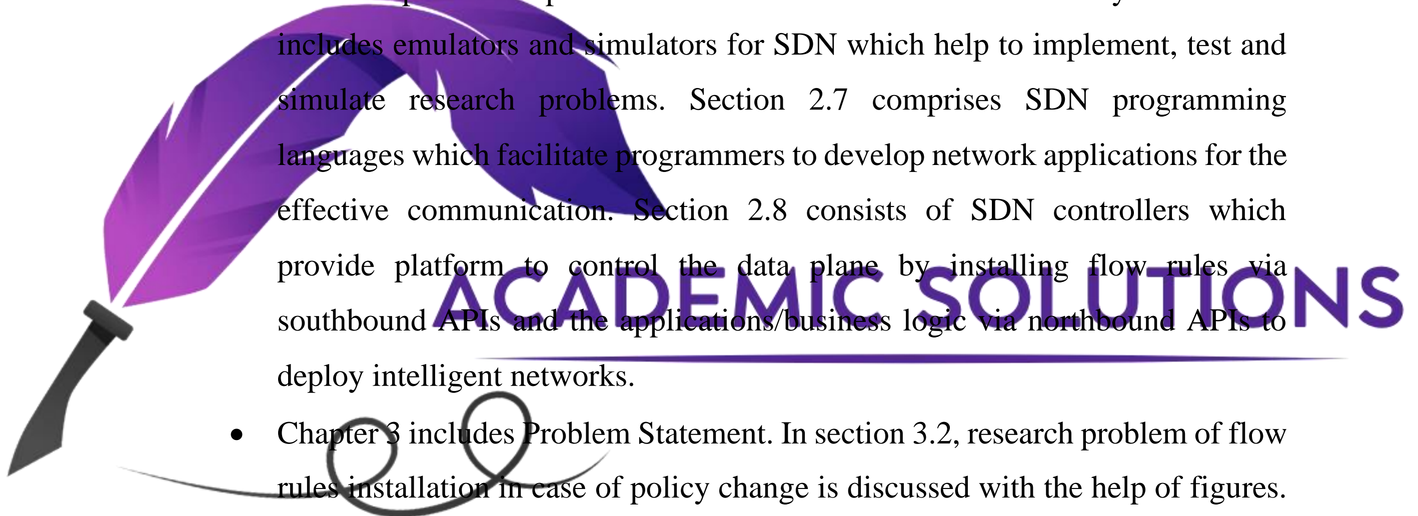

ACADEMIC SOLUTIONS

- Mininet Network Emulator [30] and POX [31] SDN controller are utilized to evaluate the performance of our proposed solutions. A comprehensive comparative analysis with existing approaches [52,58] is performed by considering different performance metrics, like, packet violation percentage, network throughput, normalized overhead, delay etc. The results of the proposed approaches are illustrated by utilizing graphs.
- The following research publications are part of thesis:
  i. Mudassar Hussain, and Nadir Shah. "Automatic rule installation in case of policy change in software defined networks." Telecommunication Systems, 68.3:461-477, 2018.
  ii. Mudassar Hussain, Nadir Shah, and Ali Tahir. "Graph-Based Policy Change Detection and Implementation in SDN." MDPI, Electronics, 8.10: 1136, 2019.
  iii. Mudassar Hussain, et al. "A Comprehensive Survey of Existing Approaches of Software Defined Networking" (Submitted).

1.5 Thesis Organization

The rest of thesis is organized as follows:

- Chapter 2 describes the Literature Review in detail along the limitations of state-of-the-art existing approaches. These approaches are classified into seven categories based on their functionalities and implementation mechanisms. In section 2.2 network testing and verification studies are discussed which consist of mechanisms for testing and debugging tools. Section 2.3 includes flow rule installation mechanisms that comprise reactive, proactive and hybrid flow rule installation techniques. In section 2.4 network security and management issues related to SDN implementation in different scenarios are discussed along with solutions to the problems. Section 2.5 comprises memory management studies which help to utilize precious TCAM resources in an efficient way. Section 2.6 includes emulators and simulators for SDN which help to implement, test and simulate research problems. Section 2.7 comprises SDN programming languages which facilitate programmers to develop network applications for the effective communication. Section 2.8 consists of SDN controllers which provide platform to control the data plane by installing flow rules via southbound APIs and the applications/business logic via northbound APIs to deploy intelligent networks.



- Chapter 3 includes Problem Statement. In section 3.2, research problem of flow rules installation in case of policy change is discussed with the help of figures. Section 3.3 contains composition and formalization issues of network policies.
- Chapter 4 consists of Proposed Solution. In section 4.2, solution of purging all flow rules in case of policy change is discussed which is not an efficient solution. Section 4.3 includes deletion of only those flow rules which conflict with the changed policies. In addition, section 4.4 comprises matrix based proposed solution and section 4.5 includes graph based proposed solution for the efficient handling of policy change.
- Chapter 5 comprises Results and Discussions of the proposed approaches for policy change detection and implementation. Section 5.2 consists of comparison of EPE to the existing approaches [52,58] on the basis of Packet Violation Percentage, Successful Packet Delivery, Normalized Overhead and Network Throughput. In section 5.3, the experimental results of GPE are

compared with EPE and analyzed based on extended performance metrics, like, Policy Change Detection Time, Packet Violation Percentage, Successful Packet Delivery, Normalized Overhead, Average Verification Time, Average End-to-End Delay and Network Throughput.

- Chapter 6 consists of Conclusion and Future Work. Section 6.1 contains the conclusion of our research work which states that the proposed approaches help to improve network efficiency by minimizing packet violations, increasing network throughput and decreasing end-to-end packet delays in the network. Section 6.2 includes future work to carry forward this research further.

Chapter 2

Literature Review

2.1 Introduction

This chapter includes detailed survey of existing approaches related to the identified research problem. These are approaches are organized into seven categories as shown in Figure 2.1. These include network testing and verification, flow rule installation mechanisms, network security and management, memory management, SDN simulators and emulators, SDN programming languages and SDN controller platforms. We have discussed different kinds of bugs, like, violations and incorrect implementation of network policies, forwarding loops, black holes, flow table consistencies etc. in these studies. The violation and incorrect implementation of network policies are high priority invariants which occur due to their misconfigurations, conflicts, overlapping and mistranslations which may result in packet violations, network attacks and reduce network efficiency. The forwarding loops are second priority bugs that causing a serious network problem in which data packets continue to be routed in a circle within the network. These normally occur due to the incorrect network policies configurations that may have serious impact on the network, and in some cases it completely disable the network. Black holes are the third priority bugs which cause the network to silently drop the network packets without informing the source. These can be detected by monitoring the lost traffic. Flow table inconsistencies are the fourth priority bugs that can be triggered due to network updates and it results in packet violations. OpenFlow helps to resolve such kind of problems by controlling the network from central controller by efficient flow rule installation mechanisms. These approaches are described in detail along with their limitations and relevance to this research.

ACADEMIC SOLUTIONS

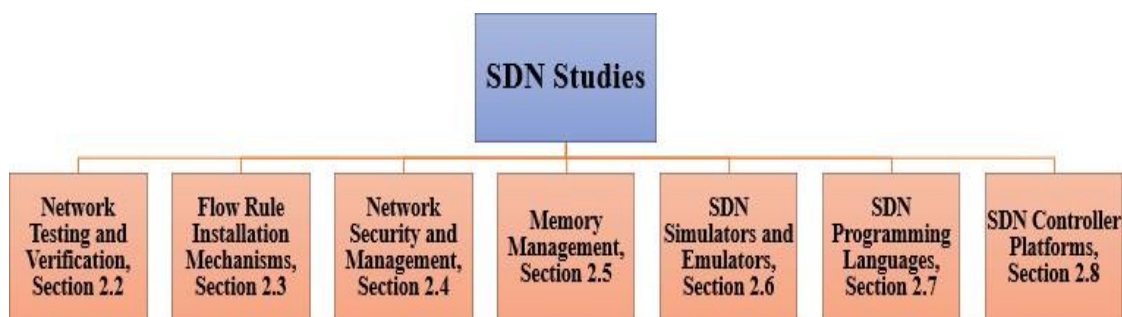


Figure 2.1: SDN Studies Categorization Hierarchy Comprised of Seven Sections (Section 2.2 to Section 2.8)

2.2 Network Testing and Verification

A prototype Network Debugger (NDB) [44] presents a way to debug SDN using two primitives; breakpoints and packet backtraces respectively. The intention of NDB is to debug networks in a way as we do debugging in software programs. The main goal is to help in locating order of events which leads towards error conditions, these can be the violations of network policies. To identify these error conditions, different actions like breakpoints, watches, backtraces, etc. are used. During debugging a program, GNU Debugger (GDB) [45] pauses execution at a breakpoint and then it displays sequence of events which leads to that breakpoint. Like gdb, backtrace in NDB defines a breakpoint which displays sequence of steps which lead to that breakpoint as per specific flow. NDB provides break points and packet backtraces as two primitives to debug SDN for black holes, loops and other network wide invariants. Backtrace outlines a breakpoint that exhibits order of events which leads towards the specific breakpoint as per specific flow.

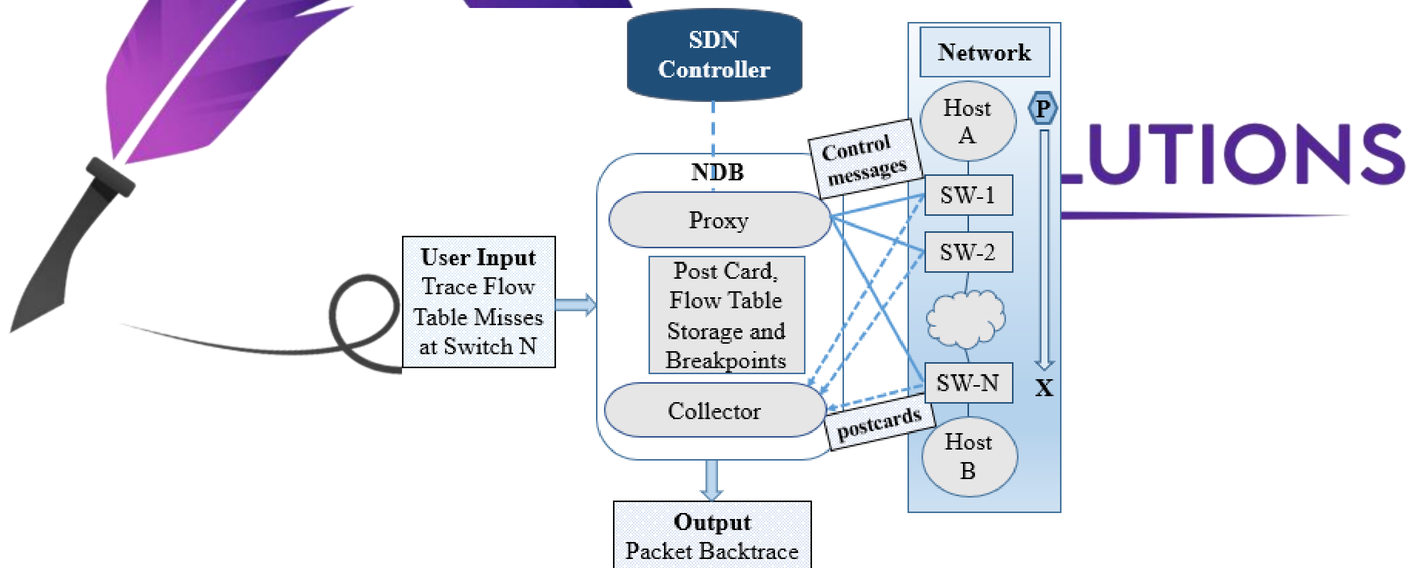


Figure 2.2: NDB System Architecture [44] Showing Procedure of Identifying Network Wide Invariants by Using Proxy and Collector

Figure 2.2 shows two major components of NDB, these are Proxy and Collector. Proxy works at data plane as follows. When the data plane (router/switch) takes an action for a data packet, the proxy creates a postcard message and sends the postcard message to the controller. The postcard message is the truncated copy of the packet's header which contains matching flow rule, switch and output port. After receiving the postcard

message, Collector at controller saves postcards and produces backtrace for the listed data packets. By using hash table data structure, the collector keeps the postcards from where these can be recovered effectively.

Veriflow [46] attempts to detect network wide invariants (like loops and reachability issues) in real time and notifies or blocks those invariants to occur. Veriflow System Architecture is shown in Figure 2.3 which verifies the flow rules for network wide invariants in following three steps. Firstly, network is segmented in a collection of Equivalence Classes (ECs) by using network routing policies. Secondly, Veriflow creates individual graphs for the specific Equivalence Class which denotes respective network behavior. Thirdly, with the help of these graphs status of network invariant is identified. It uses Trie data structures [47] to store the information like network policies, overlapping policies, and computes the affected ECs in systematic way. There are also similar other research work to debug traditional as well as SDN environments like [48,49] and [50] which can detect network anomalies, ensure consistency of data forwarding plane [51,53] and allow several applications to run parallel in a non-conflicting way [54]. However, unlike the proposed approach, none of these provides mechanisms to prevent network policy violation.

ACADEMIC SOLUTIONS

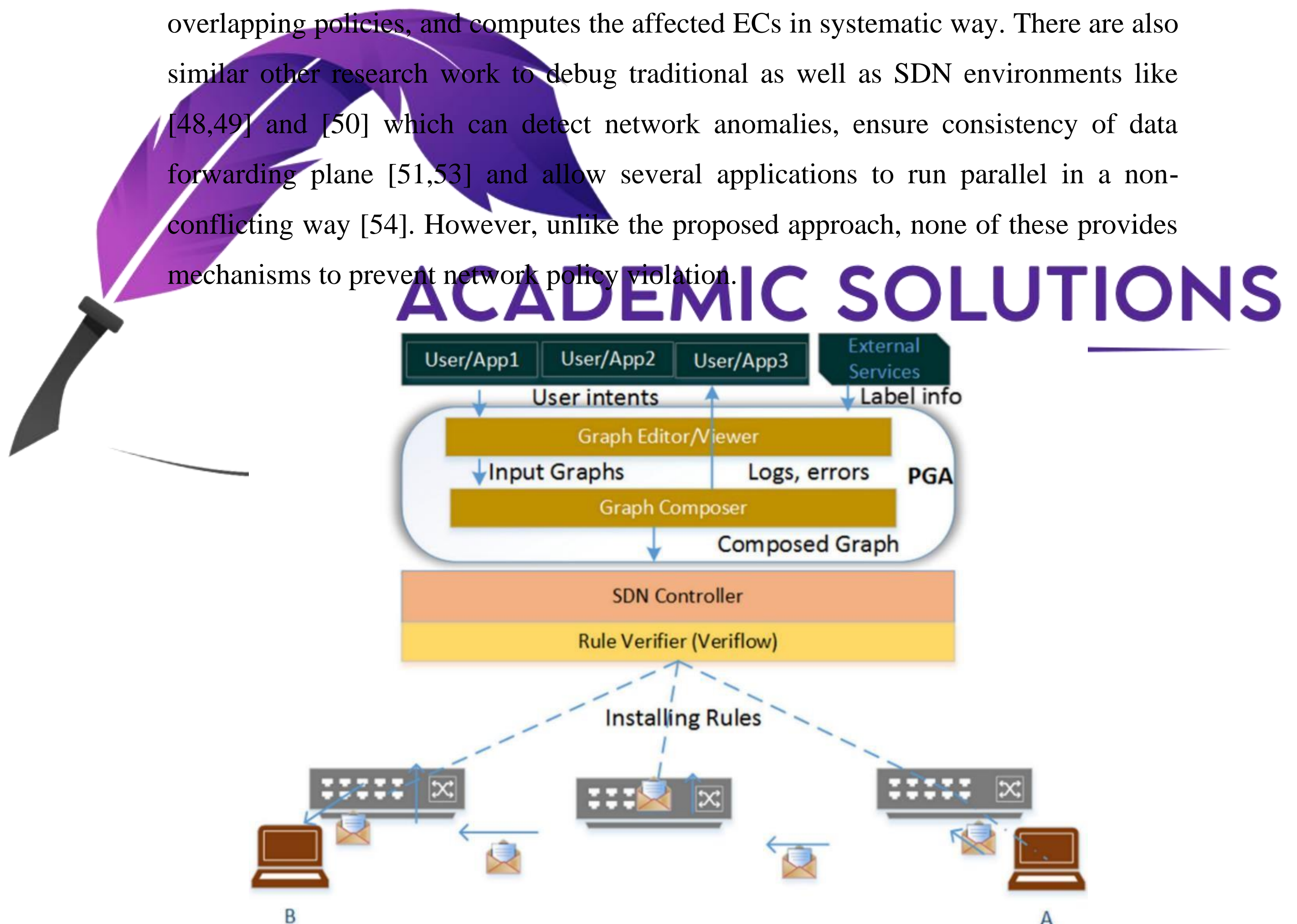


Figure 2.3: Veriflow System Architecture [46] Showing Verification of Flow Rules for Network Wide Invariants

Network wide invariants like loops and black holes are detected in No bug In Controller Execution (NICE) [52] via model checking and symbolic execution. The following illustration makes it clear that how a bug can occur in SDN applications. Figure 2.4 depicts that a data packet entitled PKT-1, before the appropriate installation of flow rules, arrives at Switch-2 (SW-2). In this situation, SW-2 sends a digest packet to controller for the installation of rules. However, by considering this situation of already installed rules for PKT-1, controller discards digest packet. This shows an error of unforeseen behavior of SDN Application. It manifests that, though SDN application is working aptly, yet it has some problems in implementation.

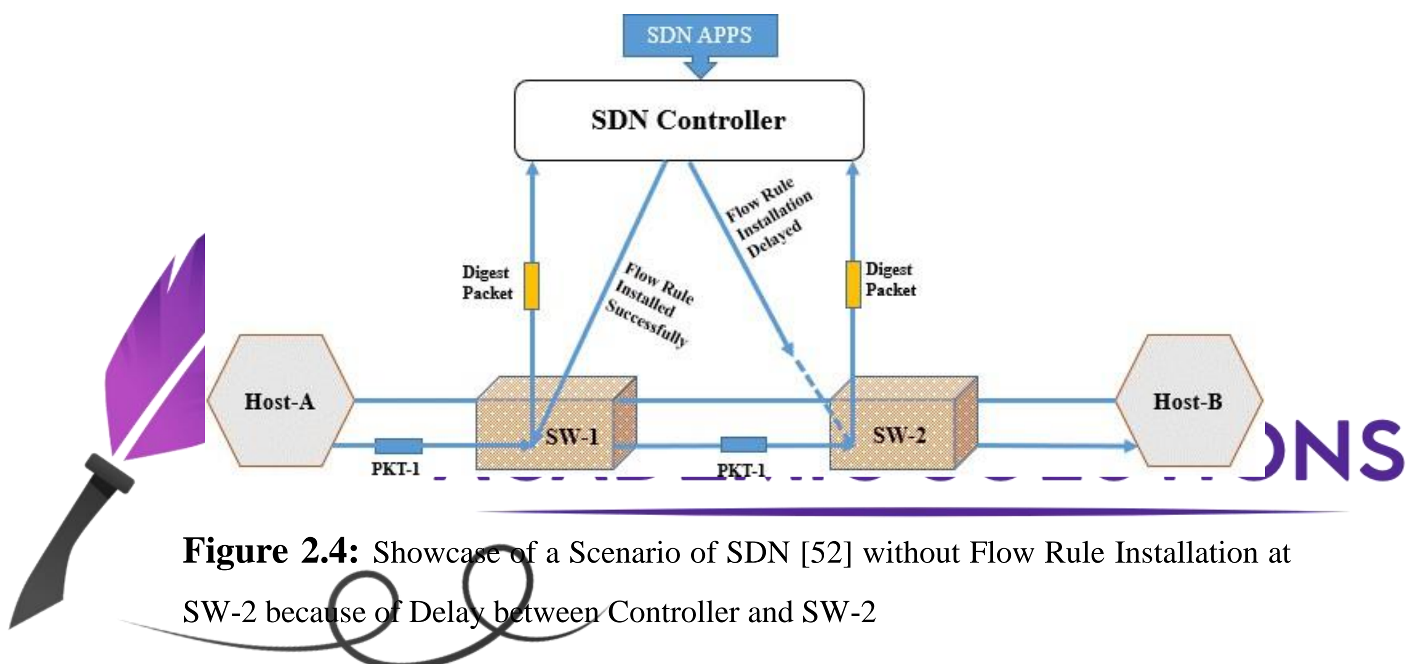


Figure 2.4: Showcase of a Scenario of SDN [52] without Flow Rule Installation at SW-2 because of Delay between Controller and SW-2

To detect such network wide invariants, NICE proactively tests the SDN applications under diverse kind of events by automatically generating stream of packets. In SDN, the details of topology including switches and hosts are available at controller. Subsequently, NICE checks network wide invariant conditions like loops or black holes, etc. Therefore, space of possible system behavior and network invariants are explored by NICE orderly. The required search strategy in NICE can also be configured by the programmer. NICE provides output of instances of network invariants. Additionally, it also provides traces to inevitable consequence of property violations to reproduce them as shown in Figure 2.5. NICE has no mechanism that may detect change in network policy. In addition to it, if policy changes, it cannot detect whether the installed flow rules disrupt network policy or not.

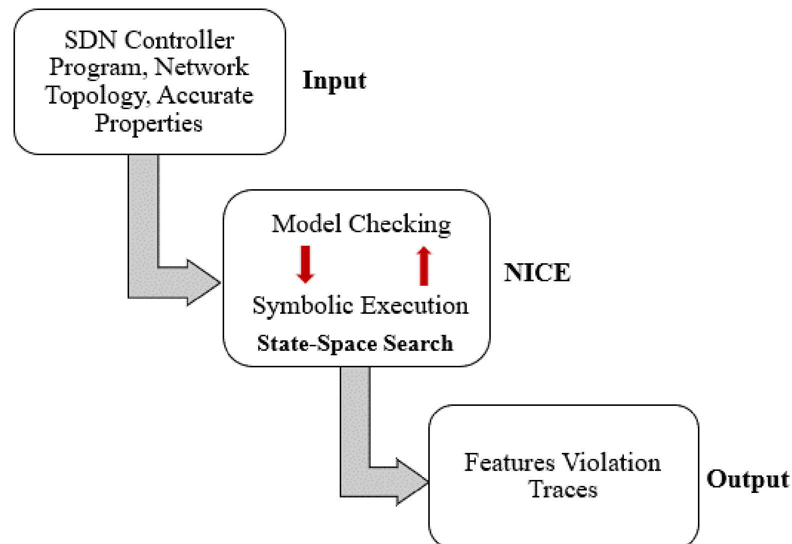


Figure 2.5: NICE Working Flow [52] Using Model Checking and Symbolic Execution

Flow-granularity Packet-in Buffer (FPB) [55] proposes the design of an efficient buffer arrangement at switch for avoiding per flow packet disorder, and thus minimizes packet drop ratio. A long delay incurs to install the flow rule for first packet of a flow "f" at SDN switches. If more subsequent packets belonging to "f" arrive and the flow rules are not installed, then these subsequent packets are sent to the controller too. In FPB, the switch only sends the first packet of flow to controller and subsequent packets are buffered. This offloads the communication and computation from the controller. Moreover, after receiving the commands/flow rules from controller, the switch forwards the packets in first-come-first-out fashion in order to avoid out-of-sequence packets for TCP. However, when network policy changes at controller, the FPB does not focus on policy violation due to already installed rules at switches.

ACADEMIC SOLUTIONS

A tool named as Header Space Analysis (HSA) [56] is beneficial for system admins, as they can statistically examine their networks for the invariants, for instance, network violations, black holes, loops, traffic isolations etc. HSA has the ability to check various hosts, network traffic and isolation of users. For example, it can provide details of "Can host A be prevented from talking to host B?" etc. In this tool, geometric approach is opted as generalization for packet classification. There are three ways to generalize operations of headers and these are listed below:

- HSA devises header length that is, header space of L, and a point in $[0,1]^L$ space is provided to encapsulate each packet.
- As HSA models as transferring function for routers and middle boxes, therefore, packets may be moved from one to another subspace.
- As all devices are modelled as an enormous network function and a topology transfer function in HSA tool, therefore, data forwarding might be drawn as alteration in geometric space.

In addition, HSA can detect different network invariants, however, it cannot auto detect the network wide invariants due to policy change.

PyResonance [57] implements state-based network policies by using Pyretic language. Pyretic has several composition operators to build a complex network control program by combining several modular control programs. Moreover, it utilizes composition operators to express state-based policies to compose multiple tasks via finite state machine and corresponding network application which determines forwarding behavior of that state. These composition operators also allow operators to specify state-based network policies by composing multiple tasks, each of which has a finite state machine and a corresponding network program that determines forwarding behavior for that state. In this way, multiple independent states can be defined along with their forwarding behavior to handle state change of multiple events. The PyResonance architecture is shown in Figure 2.6.

ACADEMIC SOLUTIONS

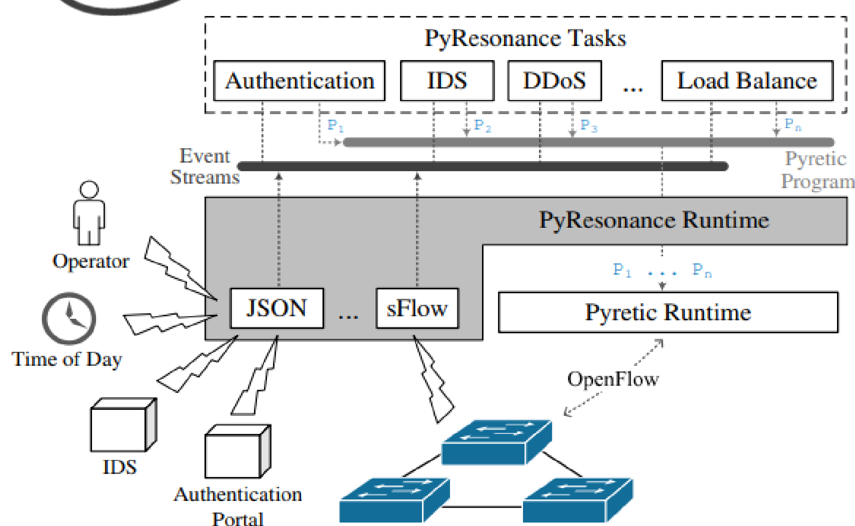


Figure 2.6: PyResonance Architecture [57] Showing Implementation of State-Based Network Policies via Pyretic Language

Although pyResonance along with pyretic provides mechanisms to constitute policies effectively, however, this lack in the effect of policy change at controller on the packet violations due to already installed flow rules at data plane.

A mechanism named as Policy Graph Abstraction (PGA) [58] provides an automatic and conflict free policies, for instance, ACLs, load balancing [59] etc. It examines various network policies which are individually stated for any conflict as shown in Figure 2.7. In different situations, policies conflict with each other due to various perspectives. By combining two policies, the resultant policy will become complex and there will be great number of conflicts/dependencies of access. To detect these conflicts/dependencies, PGA uses a graph-based abstraction to simplify the network policies. In PGA, Users, Administrators, and SDN Applications define their network policies and these policies are specified in the form of graphs.

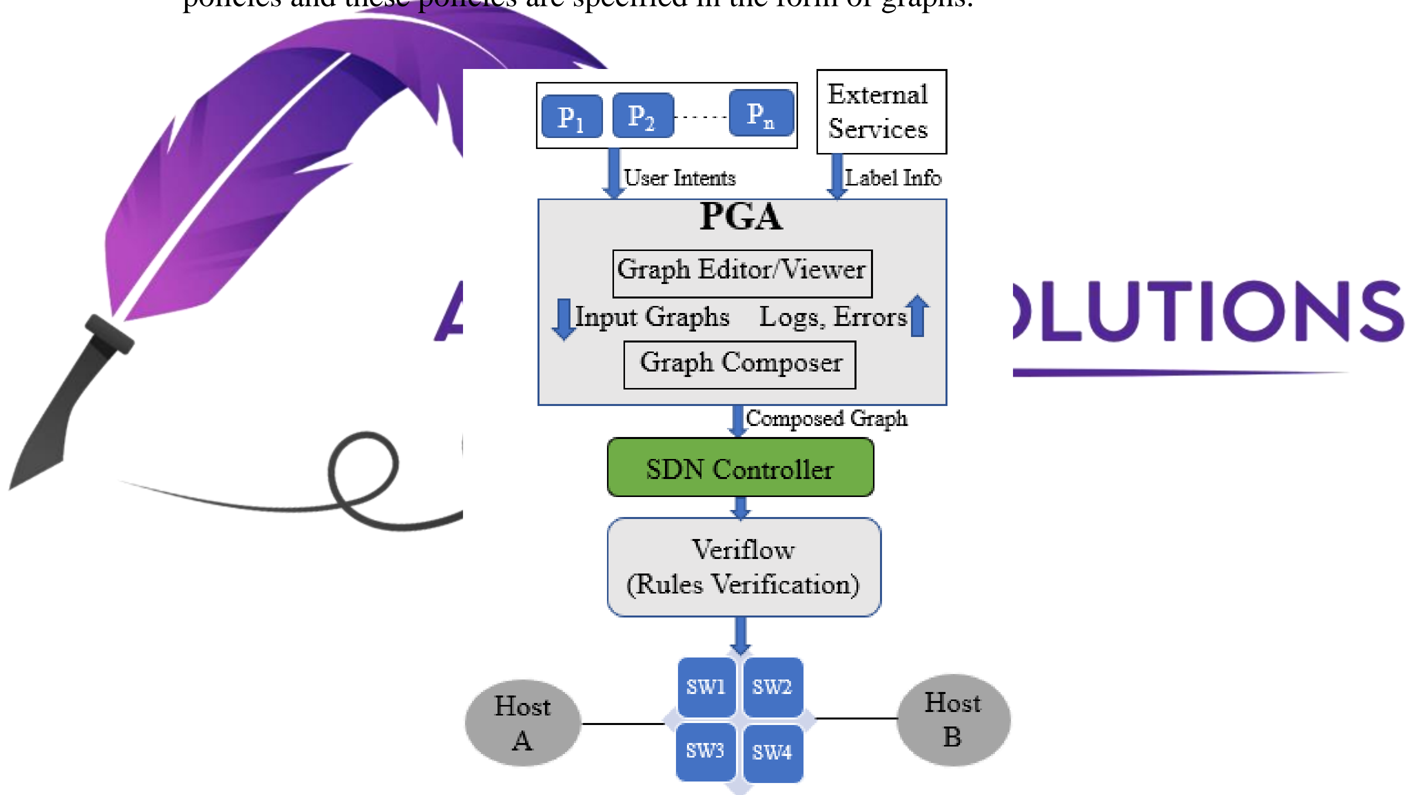


Figure 2.7: PGA System Architecture [58] Presenting Phenomena of Automatic and Conflict Free Policies via Graph Composer and Veriflow

These graphs are submitted to a Graph Composer through a PGA User Interface (UI). Graph composer automatically resolves these conflicts among different graphs. The graph composer can give some possible suggestions to the network administrator to resolve the conflicts among various policies and finally generates an error-free/conflict-

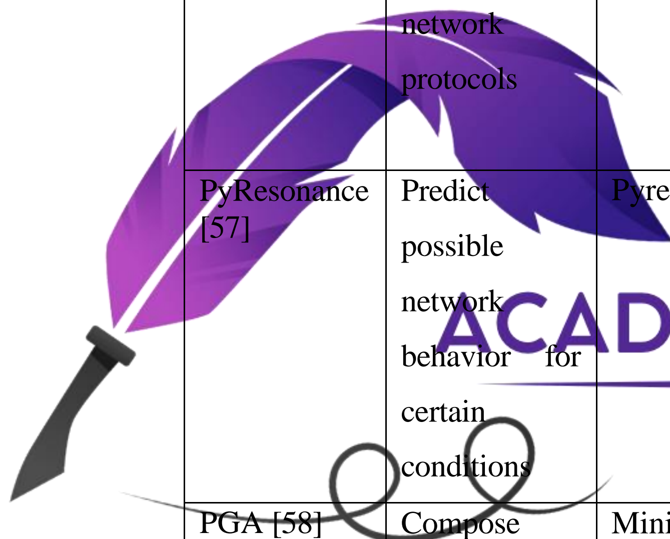
free graph. In automated network infrastructures, such as enterprise networks and clouds, great number of network policies are generated automatically. PGA focuses on the conflict in the implementation of these multiple network policies and resolves these conflicts at controller. In [60] formal approach to specify and verify Service Function Chain policies is presented. The aim of this research is to identify the presence of anomalies in the network policies before deployment. It means that it should be verified before the installation of flow rules at the data plane. In order to achieve the desired goal, the forwarding policies are formally represented and set of anomalies are detected against set of flow rules for the respective policies. In addition, it also provides provision for network administrators to specify their own anomalies. The results state that the proposed approach can verify anomalies of a reasonable sized network in milliseconds.

Table 2.1: Summary of Network Testing and Verification Studies

Study	Purpose	Tools/Techniques	Description
NDB [45]	Debugging of SDN	Mininet	NDB debugs SDN to systematically track down root causes of bugs.
Veriflow [47]	Verification of Network Wide Invariants	NOX, Mininet	Veriflow checks network invariants in real time by detecting these as and when they happen.
FlowChecker [50]	Flow table verification	OpenFlow Switch, Flow Table	FlowChecker verifies generic properties of global behaviors based on flow tables.
Anteater [51]	Verification of data plane state	Linux, C++, Ruby	Anteater analyzes the data plane state and verifies the network invariants violations.
NICE [52]	Detect bugs in SDN applications	Mininet, OpenFlow Switch, Network X	NICE Detects invariant conditions in SDN applications via model checking and symbolic execution.

ACADEMIC SOLUTIONS

FPB [55]	Relax the concept of per packet consistency	Python, OpenFlow, NOX	It guarantees to preserve properties of flows during the transition time and proposes a formal model which accomplishes consistent policy updates via handling in-transit packets at next hop.
HSA [56]	Statically analyze network invariants without network protocols	Ubuntu, Flow-based Management Language (FML), Prolog,	HSA helps system administrators to statically analyze computer network for network invariants (like ACL policy violation, black hole, reachability, etc.) without depending upon a specific network protocol.
PyResonance [57]	Predict possible network behavior for certain conditions	Pyretic, Python	pyResonance utilizes state-based network policies to predict possible network's forwarding behavior for certain network events using Pyretic composition operators.
PGA [58]	Compose multiple independently policies in an efficient manner	Mininet, Pyretic compiler, POX Controller	PGA is a mechanism for automatic normalization of network policies which inspects multiple network policies that are individually stated for any conflict.
Specifying and checking policies and anomalies in SFC [60]	Anomalies detection in network policies before deployment	Java-based prototype, OpenFlow switch	It provides mechanism to detect anomalies before installation of flow rules at data plane.


ACADEMIC SOLUTIONS

The summary of network testing and verification studies are presented in Table 2.1 and we draw following conclusions based on these studies. NDB [45] helps to debug SDN using two primitives; breakpoints and packet backtraces which facilitate in locating order of events which leads towards error conditions. Though NDB can identify error conditions, yet it does not fix the error conditions. Moreover, the user has to manually check for error condition, like network policy violation, that is, NDB does not auto-detect the network policy violation. Veriflow [47] checks network invariants in real time by detecting them as and when they happen. If there are network invariants, then either alarm is generated or the rules are blocked by not sending to data plane. It checks network invariants in real time but does not prevent network policy violation. In addition, it does not work in multi-controller architecture. NICE [52] detects bugs and invariant conditions in SDN applications by using model checking and symbolic execution. However, it does not detect change in policy nor detects installation of rules which violate the network policy in case policy changes on controller.

FPB [55] proposes the design of an efficient buffer arrangement at switch for avoiding per flow packet disorder, and thus minimizes packet drop ratio. HSA [56] facilitates network administrators to statistically analyze their networks for the network wide invariants. HSA only works for static networks and it neither detects the change of network policy nor identifies the packet violations that occur by already installed flow rules. PyResonance [57] implements state-based network policies by using Pyretic language. Pyretic has several composition operators to build a complex network control program by combining several modular control programs. PGA [58] provides an automatic and conflict free policies by examining various network policies which are individually stated for any conflict. In PGA, the focus is on implementation of various policies in a way so that conflict does not occur. However, it neither detects the change in policy nor it installs/deletes conflicting flow rules on the data plane. There are also similar other works to debug traditional as well as SDN environments like [49-51] which can detect network anomalies, ensure consistency of data forwarding plane [51,53] and allow several applications to run parallel in a non-conflicting way [54]. However, all these mechanisms lack in detecting network policy change and delete conflicting flow rules along with installation of new flow rules as per new network policies to avoid packet violations and network inefficiencies.

ACADEMIC SOLUTIONS

2.3 Flow Rule Installation Mechanisms

The OpenFlow Rules Placement Problem (ORPP) [61, 62] describes how to define and place flow rules in SDN at appropriate place while following all technical and non-technical requirements (like network policies). It proposes two rules placement frameworks; OFFICER and aOFFICER. OFFICER uses optimization techniques to place the flow rules for those requirements whose flow rules are known and stable in a specific time interval, for example, network policies. aOFFICER utilizes adaptive control mechanisms to place the flow rules for those requirements whose flow rules are unknown and vary over a specific time interval (like load balancing requirements). Both OFFICER and aOFFICER techniques help to place flow rules efficiently.

vCRIB: Virtualized Rule Management in the Cloud [63] provides mechanism for data center operators which offers an abstraction for specifying and managing various flow rules. It automatically partitions and places the flow rules on hypervisors and switches to achieve the best trade-off of performance and scalability. DevoFlow [64] provides a mechanism to modify OpenFlow model by breaking the coupling between control and global visibility. It allows operators to target only those flows which are important with respect to the network management. In this way, it minimizes switch-internal communication between control and data planes. Firstly, it minimizes the need to transfer statistics for boring flows, secondly, it also minimizes the need to invoke the control plane for most flow setups. This helps to maintain a certain level of visibility by minimizing communication overhead between control and data planes. However, the prototype is not simulated on actual packets.

ACADEMIC SOLUTIONS

Infinite CacheFlow [65] resolves problem of limited number flow rules at switches by proposing a hybrid switch design (hardware and software) which relies on flow rule caching to provide large flow rule tables at low cost. By having more flow rules stored at data plane, the probability of packet violation is higher for the packets whose network policies change at controller after the corresponding flow rules are installed at data plane. SwitchReduce [66] proposes an approach which reduces switch state and controller involvement in SDN. It assumes that number of flow match rules at any switch should not be more than the set of unique processing actions. Moreover, flow counters for each unique flow may also be maintained at only one switch in the network.

The proposed approach can reduce flow entries up to 49% on first hop switches, and up to 99.9% on interior switches. In addition to that, flow counters are also reduced by 75% on average. It shows some failures due to topology changes. Moreover, analysis in packet violations in case of change in policy is not performed on larger data centers.

A Flow Entry Management Scheme for Reducing Controller Overhead [67] proposes a cache algorithm strategy, "Least Recently Used (LRU)", for flow rules stored at the data plane to minimize the communication overhead between controller and data plane. By using this strategy, a switch can keep the recently used flow entries in the table to avoid cache-miss problem. This in turn increases flow entry matching ratio. This approach also ignores the case when the network rules change for the flow entries present in the data plane. The authors in [68] propose a new flow rule multiplexing mechanism that jointly optimizes the flow rule allocation and traffic engineering (like Quality-of-Service parameters) in SDN. The proposed mechanism states that the same set of flow rules which are installed on each switch may be applied to the whole flow of a session going through but towards different paths. The proposed mechanism is tested by using ITALYNET topology. The results also show that the proposed scheme saves TCAM resources and guarantees high QoS satisfaction. It guarantees performance within specific range of bandwidth resources. However, the proposed approach ignores the change of network policy at controller and its consequences on the already installed flow rules at data plane.

ACADEMIC SOLUTIONS

DomainFlow [69] proposes a solution of flow level control and granularity in ethernet switches via OpenFlow via splitting the network into different slices and exact matches are used to enable practical flow management via OpenFlow protocol for Ethernet switches. The proposed mechanism can only be implemented with commodity switches along with small number of flows. However, it is not practical with respect to the current trends that require extensive number of flows especially in case of change of policies and deletion/installation of flow rules. SourceFlow [70] proposes solution to handle a large number of flows without affecting flow granularity. This scheme helps to minimize problem of expensive and power consuming search engine devices from the core nodes. This in turn helps to expand networks in a scalable way. This scheme is time consuming in a sense that hosts produce millions of flows simultaneously and

these flows are controlled on a per-flow basis. However, this scheme does not consider the policy change at controller and its effect on the data plane performance.

In [71] SDN based proactive flow rules installation approach is proposed for efficient communication in Internet of Things (IoT). It resolves the problem of flow installation delay as well as congestion due to packet-in messages. This saves energy and other potential resources of network nodes. The results reveal that the proposed mechanism reduces congestion between controller and network nodes and reduces average flow rule installation delay by 90%. In SDN, flow rules are installed in switches on the basis of exact matching [72] or wildcard-based matching [73]. The wildcard-based matching improves reusability of flow rules and reduces packet-in messages. This improves scalability both at data and control planes. However, in case of exact matching almost every flow passing the switch will generate a packet-in message to the controller which exhausts precious resources. To resolve this problem, some researchers suggest to use load balancing mechanism by installing proactive flow rules on multiple switches [74] or reactive caching flow rules in each switch. In [75], SDN based wildcard rule caching mechanism namely CAching in Buckets (CAB) is proposed for efficient flow rules placement. It suggests partitioning the field space into logical structures called "buckets", and cache buckets along with all the associated flow rules. The CAB helps to solve the flow rule dependency problem with quite less overhead. In addition, it significantly reduces flow setup time, saves bandwidth and flow setup requests.

ACADEMIC SOLUTIONS

In SDN, most of the operations are performed at the central controller, such as, network topology management, flow rules installation, load balancing, etc. These tasks overburden the controller sometimes at becoming unavailable for some required operations. DIFANE [76] is the solution to this problem in which most of the functionality is placed on the network switches. For this purpose, controller delegates the flow rules to some of the network switches that are called authority switches. These switches give the flow rules to the other switches for specific path. In this way, entire network communication becomes possible. Figure 2.8 shows the model of DIFANE, in which authority switches are directly connected to the controller. When a node that is connected to ingress, switch wishes to communicate with some other node then it sends packet to ingress port of a switch. This switch requests authority switch for the respective flow rules, similarly, as the packet moves forward flow rules are installed on

the switches and communication becomes possible. During this process, controller has less interaction with switches.

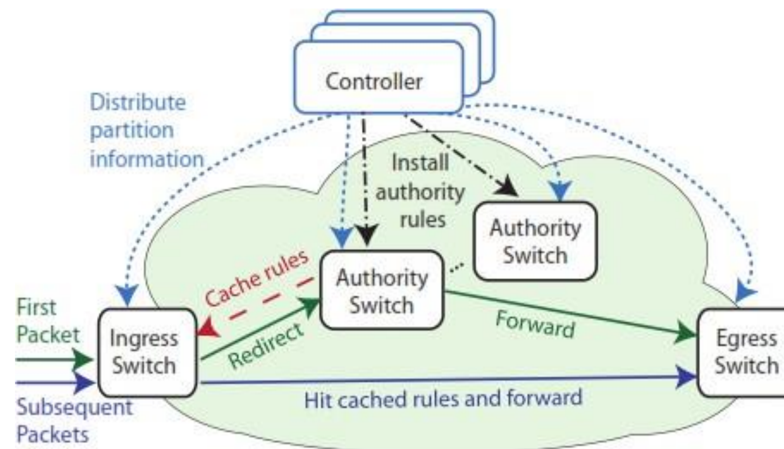


Figure 2.8: DIFANE Flow Management Architecture [76] (Dashed Lines are Control Messages. Straight Lines are Data Traffic)

In [77] the movement of mobile nodes in SDN is discussed and presented Mobi-Flow (Mobility-Aware Adaptive Flow-Rule Placement in Software-Defined Access Network) based system architecture. There are two main components in this approach; first path estimator and second one flow manager. The path estimator helps to find the possible positions of end users in the network based on the location history of the node. For this purpose, we keep tracking the previous positions of the nodes in the network in the database. By performing order-k Markov prediction method, next possible position of end users is predicted. If we have possible location information of the end user, then flow manager determines the set of access points in the path for communication between source to destination. To determine the optimal number of access points on the path, Mixed Integer Linear Programming (MILP) approach is used. After finding minimum number of APs in a path, flow manager installs the rules on that path. An IoT based environment is considered for practical implementation of the proposed system which is shown in Figure 2.9. Simulation result shows that proposed system is beneficial for minimizing delay, the number of activated Aps, control overhead and energy consumption etc.

ACADEMIC SOLUTIONS

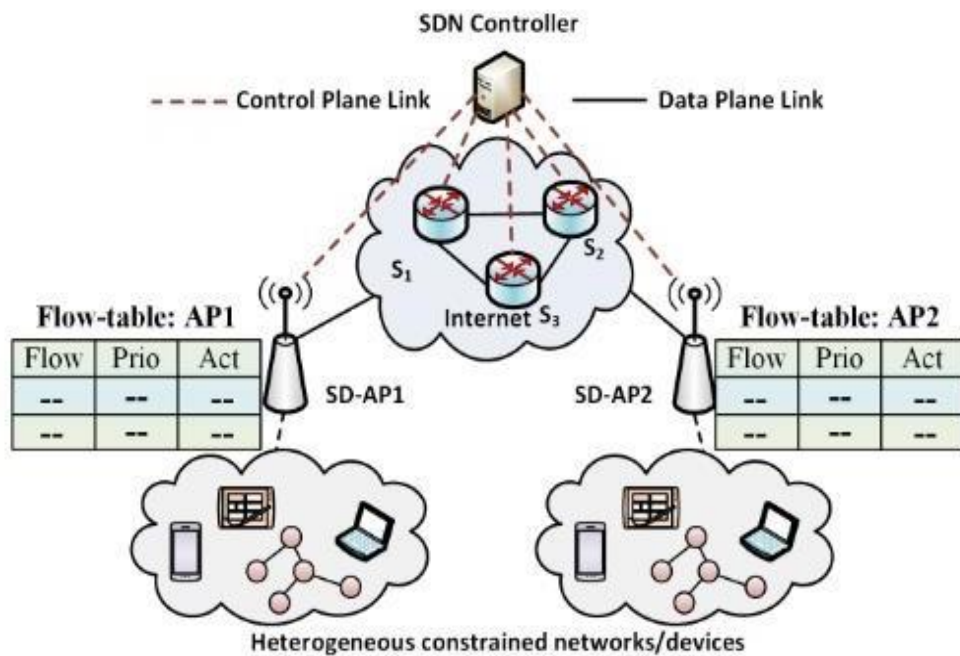


Figure 2.9: An Overview of Mobi-Flow [77] Presenting Mobility-Aware Adaptive Flow-Rule Placement in Software Defined Access Network

In [78], the authors propose a novel technique to install flow rules at the SDN switches before the packet reaches the network devices. In SDN, sometimes it happens that subsequent data packets arrive at the switches where flow rules are not found. So, these packets are discarded by the switch. To solve this issue, a new technique is proposed which computes packet arrival delay and flow rule installation time. After this computation, if there is a delay between flow rule installation and packet arrival then some delay is introduced to the packet at the predecessor switch. For example, if packet arrival is delayed at switch S_1 then the flow rule deployment at the switch S_1 is also delayed. Similarly, the data packets are delayed at the previous switch in the path so that corresponding flow rules are installed at the switch S_1. The proposed technique reduces the packet loss about 12% than the existing mechanism and improves the packet delivery ratio up to 36%.

In SDN, flow rules are installed in the temporary memory known as TCAM. This is very limited memory that cannot adopt the frequent updates of flow rules and other related information. Sometimes this leads to congested control channel and entire network behaves as a faulty network. Some schemes are proposed to deal with these

ACADEMIC SOLUTIONS

situations that install the flow rules proactively on the network switches. But due to these proactive approaches some other problems arise like scalability of the network and overflow of flow table. Moreover, these proactive schemes do not support other network operations such as, detection of heavy hitter, server load balancing and stateful firewall for network security. In current research work, in order to improve the flexibility and scalability of the entire network, a novel mechanism is proposed in which network policies are deployed on the network devices in a wildcard format. Only most important policies are cached in the flow table while unnecessary policies are removed as soon as possible. By using this mechanism, risk of flow table overflow is reduced, and it also simplifies the network policy enforcement. The wild card used in this technique requires a standard way to be adopted in the entire network. To cope this problem a network wide wild card rule engine is introduced for SDN that is known as BigMac [79]. BigMac works by advertising a layered model to publish the higher-level network policies. The policy model shown in Figure 2.10 consists of a big switch abstraction and a logical network plane that specifies the different forwarding and management policies. When a new flow is needed to be installed, then policy Caching and Mapping engine of BigMac is accessed to install flow rules on the entire path. Similarly, when a scheduled traffic is needed to be forwarded then BigMac deploys the requested flow rules on the entire path.

ACADEMIC SOLUTIONS

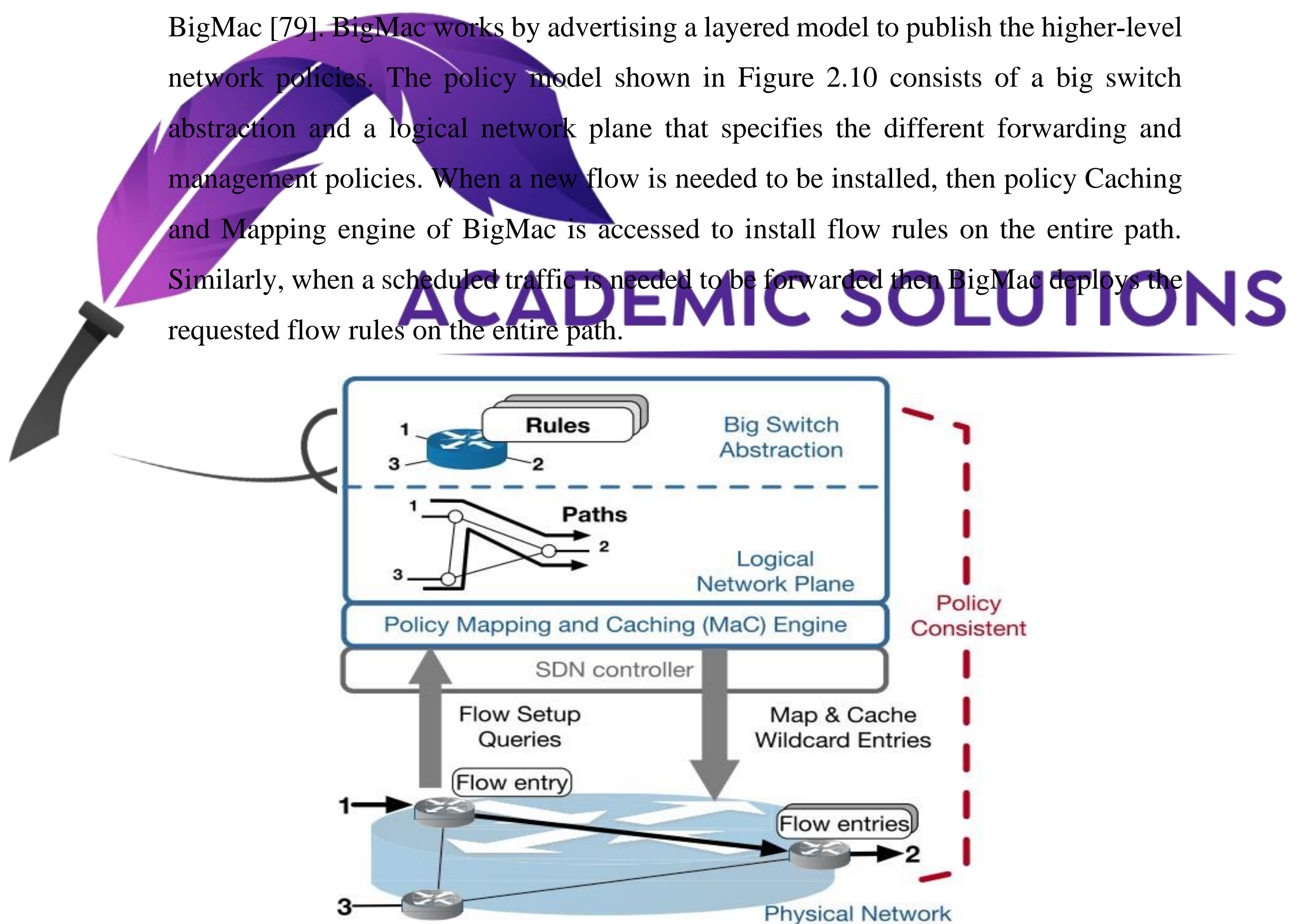


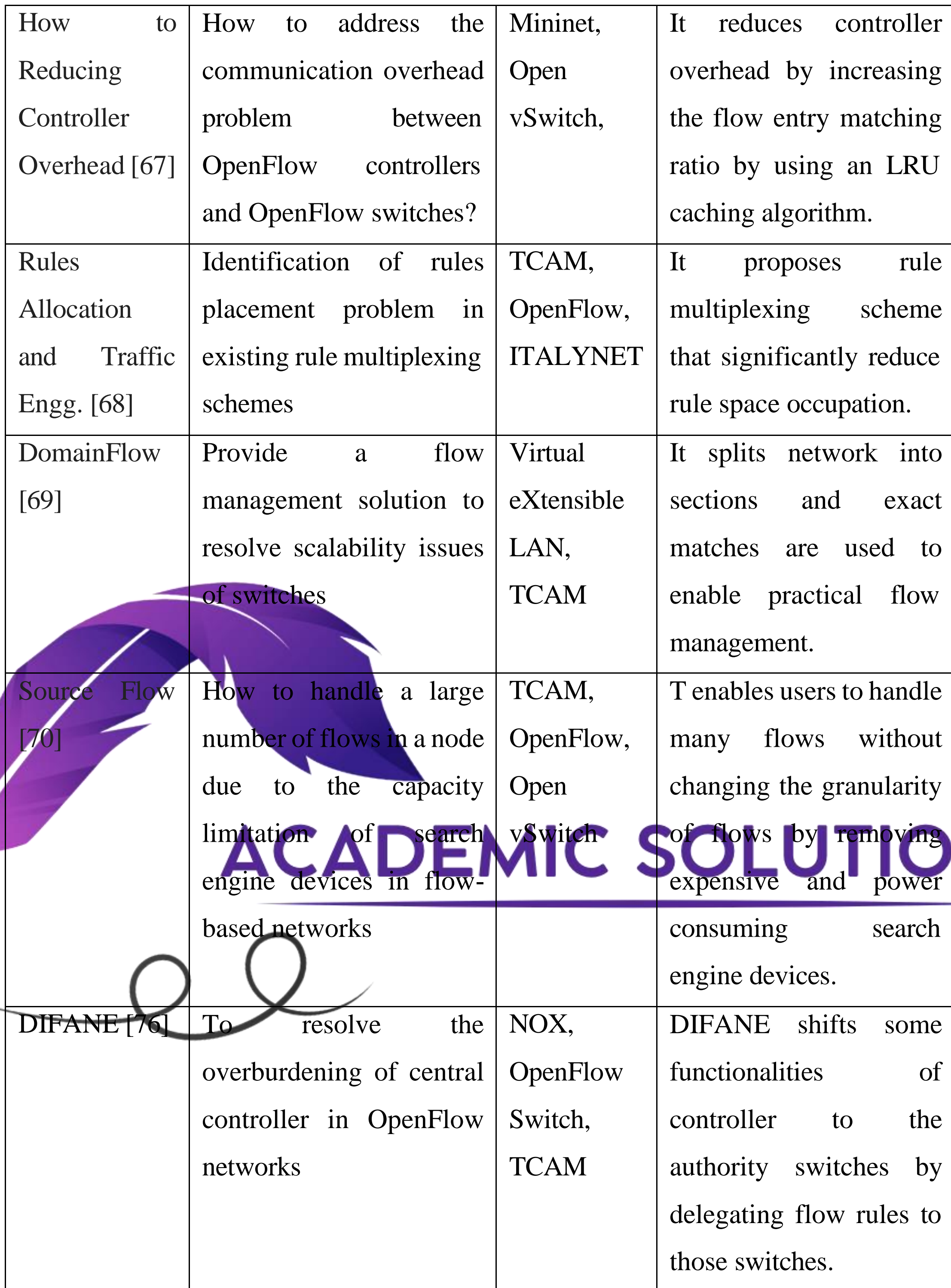
Figure 2.10: BigMac Framework [79] Presenting a Big Switch Abstraction and a Logical Network Plane Specifying Different Forwarding and Management Policies

Table 2.2: Summary of Flow Rules Installation Mechanisms

Study	Purpose	Tools/ Techniques	Description
ORPP [61,62]	Define the rules and where to place them in the network while respecting all technical and administrative requirements	Mininet, OpenFlow	In OFFICER set of flows are assumed known and stable over a specific period. The aOFFICER uses adaptive control techniques to solve the online ORPP problem.
vCRIB [63]	Achieve high scalability and performance by placing flow rules either on hypervisors or switches	VM, Open vSwitch, TCAM	It partitions and places the rules on hypervisors and switches to achieve the best trade-off of performance and scalability.
DevoFlow [64]	Reducing OpenFlow excessive overheads during communication between control and data planes	NOX, TCAM, OpenFlow	It proposes modification of the OpenFlow model which breaks coupling between control and global visibility.
Infinite CacheFlow [65]	The need of efficient ways to support the abstraction of a switch with arbitrarily large rule tables to support large number of flow rules	Ryu Controller, OpenFlow 1.0, Open vSwitch,	This research proposes a hardware software hybrid switch design that relies on rule caching to provide large rule tables at low cost.
SwitchReduce [66]	How to enable flow level details in switches for providing monitoring and control over each individual flow?	NOX Controller, OpenFlow	It decreases switch state and controller involvement in OpenFlow networks by reduce flow entries and flow counters.

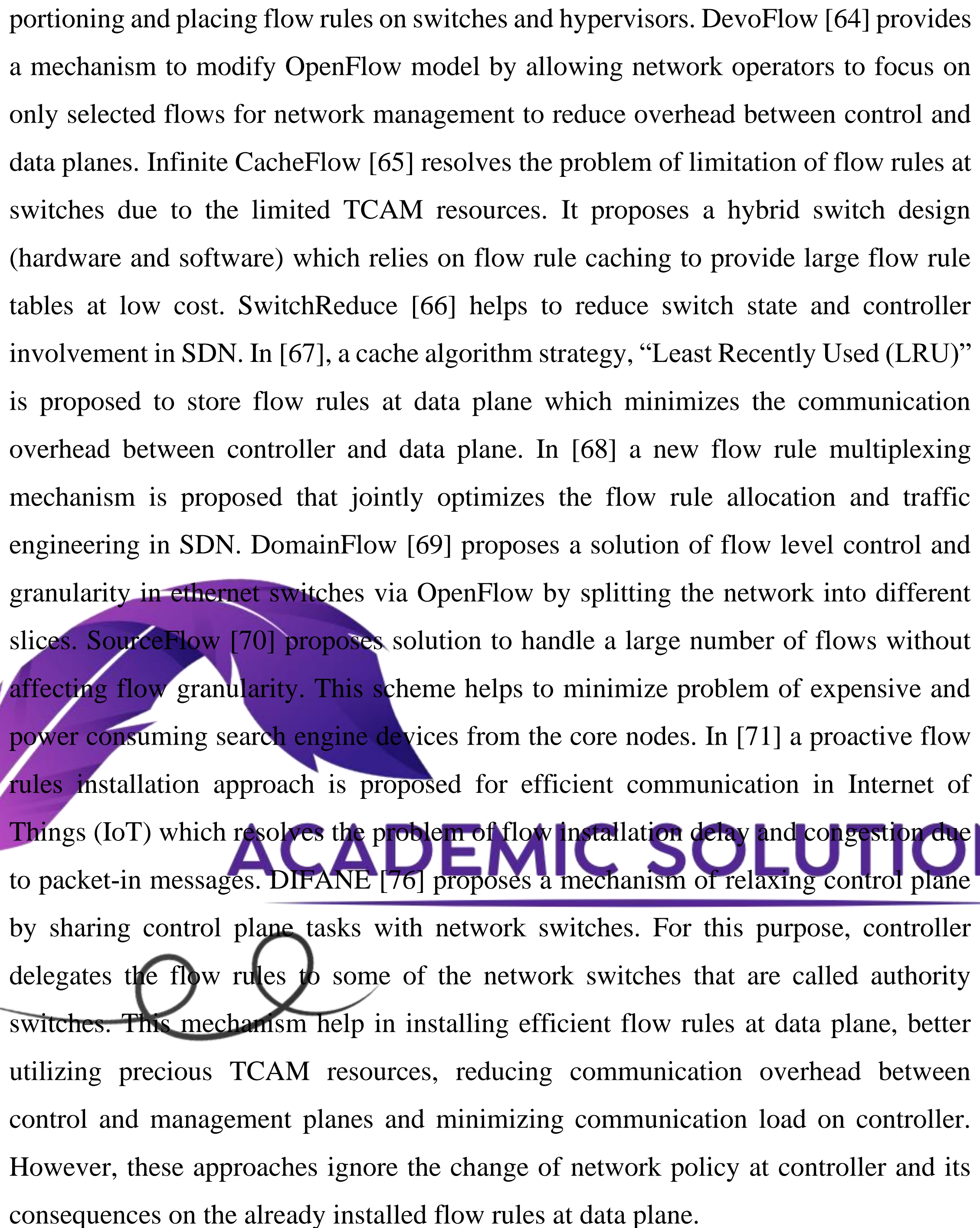
ACADEMIC SOLUTIONS

How to Reducing Controller Overhead [67]	How to address the communication overhead problem between OpenFlow controllers and OpenFlow switches?	Mininet, Open vSwitch,	It reduces controller overhead by increasing the flow entry matching ratio by using an LRU caching algorithm.
Rules Allocation and Traffic Engg. [68]	Identification of rules placement problem in existing rule multiplexing schemes	TCAM, OpenFlow, ITALYNET	It proposes rule multiplexing scheme that significantly reduce rule space occupation.
DomainFlow [69]	Provide a flow management solution to resolve scalability issues of switches	Virtual eXtensible LAN, TCAM	It splits network into sections and exact matches are used to enable practical flow management.
Source Flow [70]	How to handle a large number of flows in a node due to the capacity limitation of search engine devices in flow-based networks	TCAM, OpenFlow, Open vSwitch	T enables users to handle many flows without changing the granularity of flows by removing expensive and power consuming search engine devices.
DIFANE [76]	To resolve the overburdening of central controller in OpenFlow networks	NOX, OpenFlow Switch, TCAM	DIFANE shifts some functionalities of controller to the authority switches by delegating flow rules to those switches.



The summary of flow installation mechanisms is presented in Table 2.2 and we draw following conclusions based on these studies. ORPP [61, 62] resolve the problem of placement of flow rules by using two frameworks. These are OFFICER and aOFFICER. Both frameworks help to place flow rules efficiently. vCRIB [63] provides abstraction for specifying and managing various flow rules to data center operators by

portioning and placing flow rules on switches and hypervisors. DevoFlow [64] provides a mechanism to modify OpenFlow model by allowing network operators to focus on only selected flows for network management to reduce overhead between control and data planes. Infinite CacheFlow [65] resolves the problem of limitation of flow rules at switches due to the limited TCAM resources. It proposes a hybrid switch design (hardware and software) which relies on flow rule caching to provide large flow rule tables at low cost. SwitchReduce [66] helps to reduce switch state and controller involvement in SDN. In [67], a cache algorithm strategy, “Least Recently Used (LRU)” is proposed to store flow rules at data plane which minimizes the communication overhead between controller and data plane. In [68] a new flow rule multiplexing mechanism is proposed that jointly optimizes the flow rule allocation and traffic engineering in SDN. DomainFlow [69] proposes a solution of flow level control and granularity in ethernet switches via OpenFlow by splitting the network into different slices. SourceFlow [70] proposes solution to handle a large number of flows without affecting flow granularity. This scheme helps to minimize problem of expensive and power consuming search engine devices from the core nodes. In [71] a proactive flow rules installation approach is proposed for efficient communication in Internet of Things (IoT) which resolves the problem of flow installation delay and congestion due to packet-in messages. DIFANE [76] proposes a mechanism of relaxing control plane by sharing control plane tasks with network switches. For this purpose, controller delegates the flow rules to some of the network switches that are called authority switches. This mechanism help in installing efficient flow rules at data plane, better utilizing precious TCAM resources, reducing communication overhead between control and management planes and minimizing communication load on controller. However, these approaches ignore the change of network policy at controller and its consequences on the already installed flow rules at data plane.

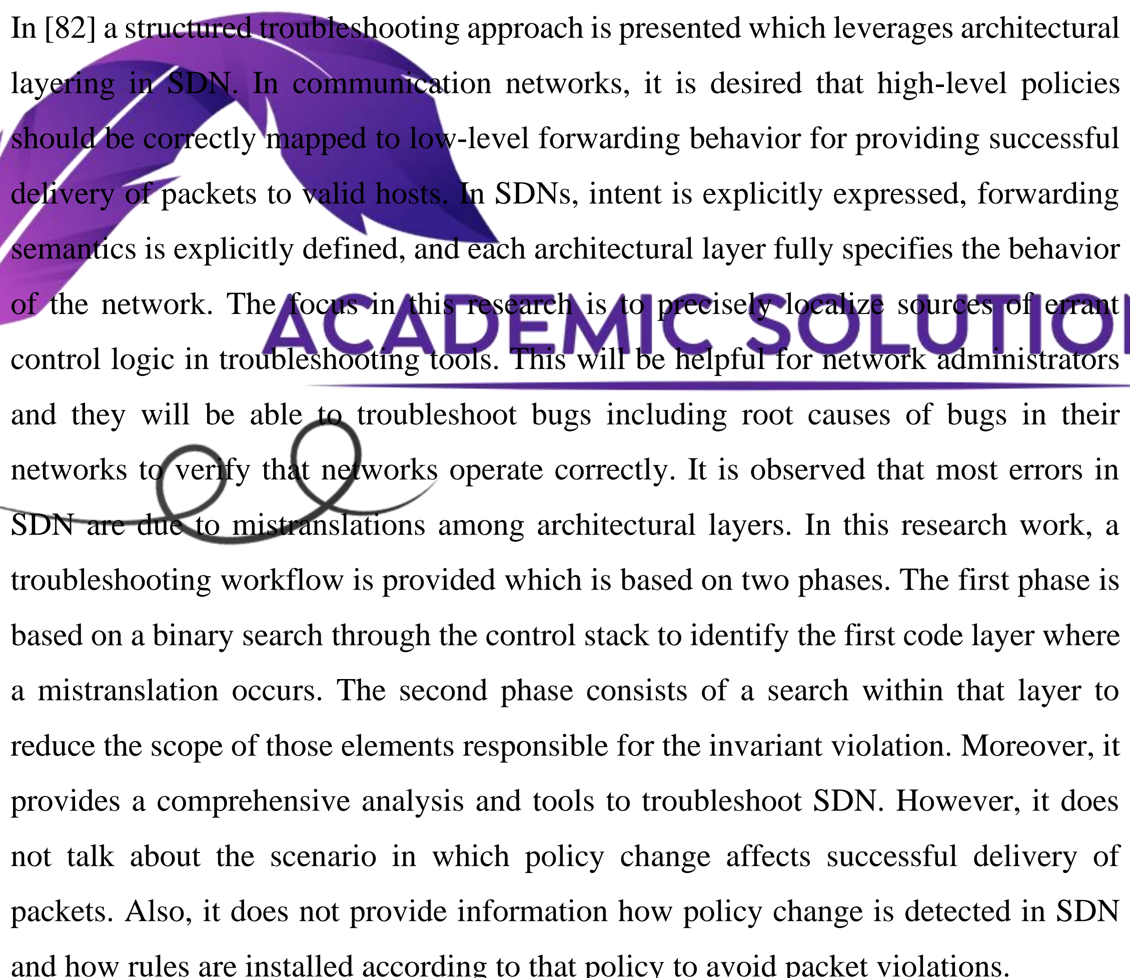


2.4 Network Security and Management

In [80] an innovative SDN based network architecture is proposed which guarantees exclusive access of network resources to a certain flow for which the user/app has made the reservation with the help of token-based authorization. The proposed SDN based system automates the reservation process and creates a strong binding between the end users/apps who request the reservation and flows. This in turn resolves the reservation

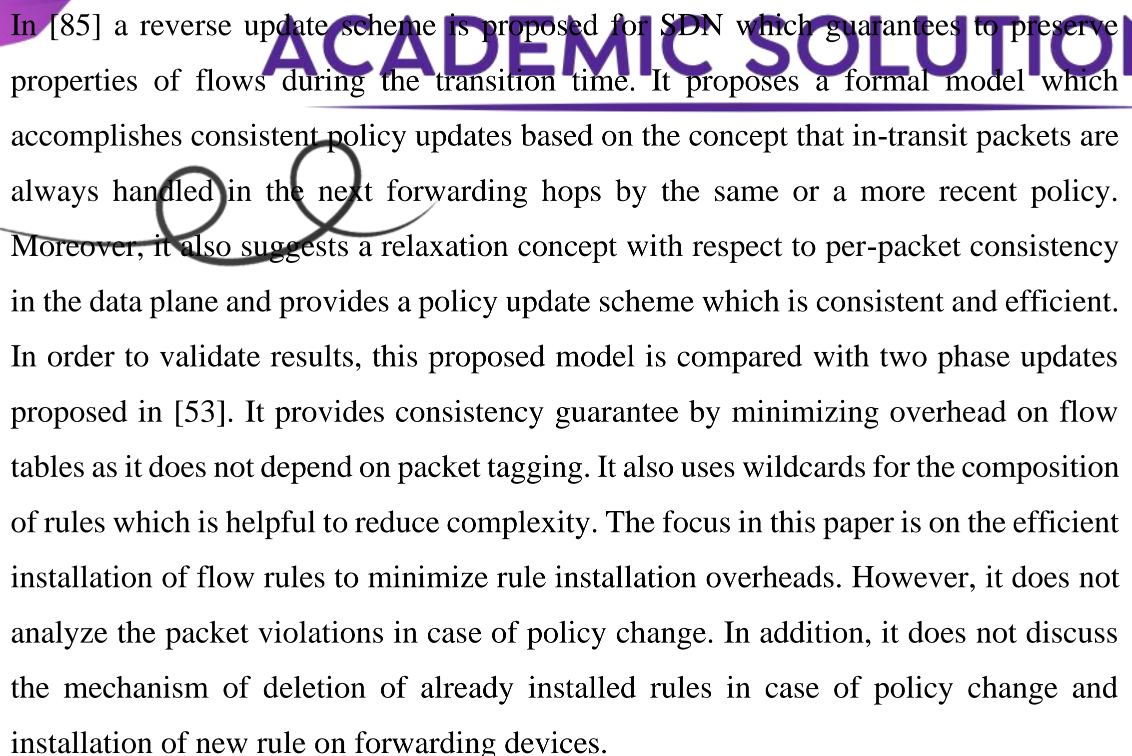
problem of dedicated bandwidth of certain resources in high speed networks for transfer of data in distributed science environments. The research work in [81] resolves the problem of network verification which consists of middleboxes, such as caches and firewalls. This research explores the possibilities of verifying the reachability properties for networks based on the network policies. In order to do that, the complex networks are sliced into small networks as per the network-wide verifications for the correctness properties. The results show that time required to verify a network invariant in a complex network is independent of its size due to slices. This work is quite good for verifying network invariants in an efficient way. The limitation of this approach is also that it has overlooked the network policy change and its effect over the flow rules present in the data plane.

In [82] a structured troubleshooting approach is presented which leverages architectural layering in SDN. In communication networks, it is desired that high-level policies should be correctly mapped to low-level forwarding behavior for providing successful delivery of packets to valid hosts. In SDNs, intent is explicitly expressed, forwarding semantics is explicitly defined, and each architectural layer fully specifies the behavior of the network. The focus in this research is to precisely localize sources of errant control logic in troubleshooting tools. This will be helpful for network administrators and they will be able to troubleshoot bugs including root causes of bugs in their networks to verify that networks operate correctly. It is observed that most errors in SDN are due to mistranslations among architectural layers. In this research work, a troubleshooting workflow is provided which is based on two phases. The first phase is based on a binary search through the control stack to identify the first code layer where a mistranslation occurs. The second phase consists of a search within that layer to reduce the scope of those elements responsible for the invariant violation. Moreover, it provides a comprehensive analysis and tools to troubleshoot SDN. However, it does not talk about the scenario in which policy change affects successful delivery of packets. Also, it does not provide information how policy change is detected in SDN and how rules are installed according to that policy to avoid packet violations.



In [83] a security problem of priority-based flow rules is highlighted, and solution of the identified problem is presented. The problem is that the low priorities malicious flow rules can manipulate the whole OpenFlow network by making the high priority

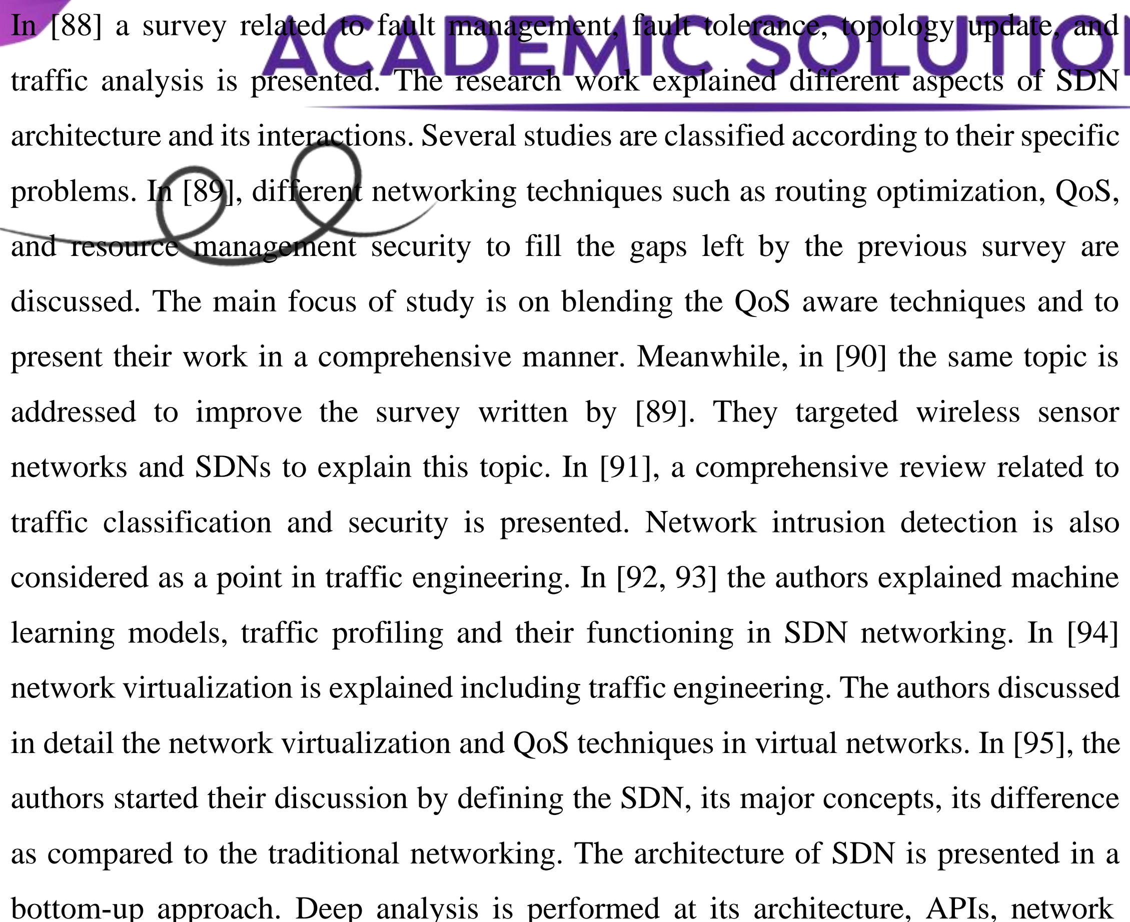
flow rules to fail. This in turn affects whole data communication process. To solve the identified problem, the authors proposed a solution which is called Switch-based Rules Verification (SRV). It works by leveraging the SDN controller to obtain the overall network view of the whole topology and detects the malicious flow rules. On detection of malicious flow rule, SRV module forwards warning messages and refuses the identified flow rule instantly. This solution helps to detect large number of flow rules in an efficient way. But this approach neither detects the policy change at the controller nor identifies the flow rules installed at data plane that violate the changed policy. In [84], Actor Based Modeling framework is proposed to investigate the problem of network verification. In this architecture, Actors are basic units of computation that hold their own memory and can communicate via asynchronous messages. This work shows how network applications are modeled by using Actors and how existing model checking mechanisms verify various properties (loops, policy violations, flow table consistencies) in SDN, so that network behaviors can be predicted to increase network efficiency. However, this work also does not consider the case of policy change and the policy violation by the packets due to policy change at controller.


ACADEMIC SOLUTIONS

In [85] a reverse update scheme is proposed for SDN which guarantees to preserve properties of flows during the transition time. It proposes a formal model which accomplishes consistent policy updates based on the concept that in-transit packets are always handled in the next forwarding hops by the same or a more recent policy. Moreover, it also suggests a relaxation concept with respect to per-packet consistency in the data plane and provides a policy update scheme which is consistent and efficient. In order to validate results, this proposed model is compared with two phase updates proposed in [53]. It provides consistency guarantee by minimizing overhead on flow tables as it does not depend on packet tagging. It also uses wildcards for the composition of rules which is helpful to reduce complexity. The focus in this paper is on the efficient installation of flow rules to minimize rule installation overheads. However, it does not analyze the packet violations in case of policy change. In addition, it does not discuss the mechanism of deletion of already installed rules in case of policy change and installation of new rule on forwarding devices.

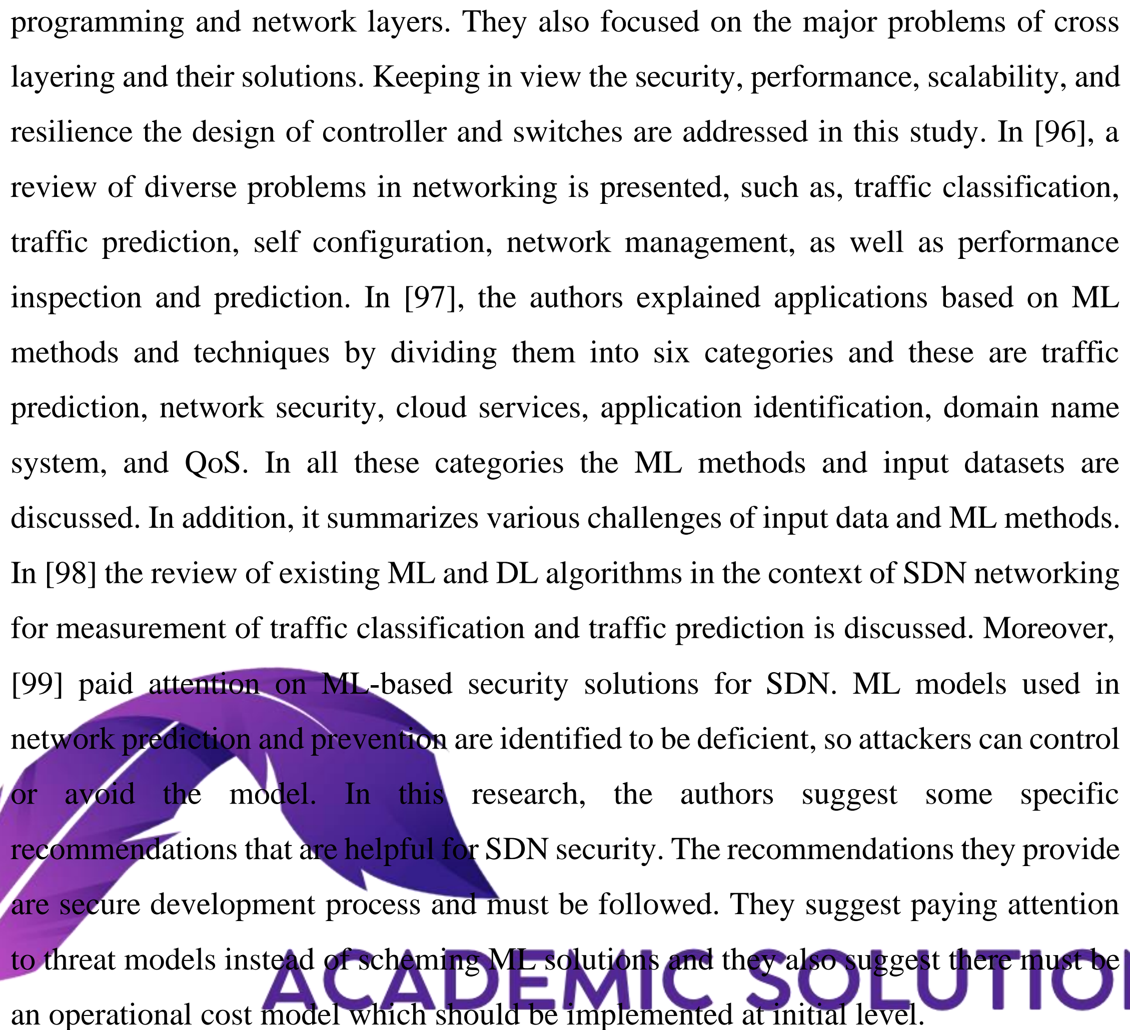
In communication networks, attackers often attack networks via bandwidth and system/application resource utilizations which leads to the popular Denial of Service

(DoS) attack. To detect such kind of attacks is very interesting research topic in networking. In SDN, deep learning algorithms are implemented to model the attack behaviors as per the information retrieved from the central controller. In [86], SDN based simulation environment is created in Mininet Emulator by utilizing Floodlight controller platform. In this environment, the flow table is assumed to be of 6-tuple as per the information retrieved from controller. After that, Distributed DoS attack model is constructed by combining the Support Vector Machine (SVM) classification algorithms. The results show that average accuracy rate of detecting Distributed DoS attack of proposed approach is 95.24% with a limited number of flow rules which are extracted from the controller. A lot of research as discussed above on the verification of correctness properties of communication networks exist. However, none of these touches the identified problem to avoid packet violations by detecting policy change and installation/deletion of rules as per change of network policy. Similarly, other works like [50, 51] are helpful in debugging networks and detecting network anomalies and [87] is helpful to run several applications on the same network in parallel without interfering with each other while maintaining performance of each application.



In [88] a survey related to fault management, fault tolerance, topology update, and traffic analysis is presented. The research work explained different aspects of SDN architecture and its interactions. Several studies are classified according to their specific problems. In [89], different networking techniques such as routing optimization, QoS, and resource management security to fill the gaps left by the previous survey are discussed. The main focus of study is on blending the QoS aware techniques and to present their work in a comprehensive manner. Meanwhile, in [90] the same topic is addressed to improve the survey written by [89]. They targeted wireless sensor networks and SDNs to explain this topic. In [91], a comprehensive review related to traffic classification and security is presented. Network intrusion detection is also considered as a point in traffic engineering. In [92, 93] the authors explained machine learning models, traffic profiling and their functioning in SDN networking. In [94] network virtualization is explained including traffic engineering. The authors discussed in detail the network virtualization and QoS techniques in virtual networks. In [95], the authors started their discussion by defining the SDN, its major concepts, its difference as compared to the traditional networking. The architecture of SDN is presented in a bottom-up approach. Deep analysis is performed at its architecture, APIs, network

programming and network layers. They also focused on the major problems of cross layering and their solutions. Keeping in view the security, performance, scalability, and resilience the design of controller and switches are addressed in this study. In [96], a review of diverse problems in networking is presented, such as, traffic classification, traffic prediction, self configuration, network management, as well as performance inspection and prediction. In [97], the authors explained applications based on ML methods and techniques by dividing them into six categories and these are traffic prediction, network security, cloud services, application identification, domain name system, and QoS. In all these categories the ML methods and input datasets are discussed. In addition, it summarizes various challenges of input data and ML methods. In [98] the review of existing ML and DL algorithms in the context of SDN networking for measurement of traffic classification and traffic prediction is discussed. Moreover, [99] paid attention on ML-based security solutions for SDN. ML models used in network prediction and prevention are identified to be deficient, so attackers can control or avoid the model. In this research, the authors suggest some specific recommendations that are helpful for SDN security. The recommendations they provide are secure development process and must be followed. They suggest paying attention to threat models instead of scheming ML solutions and they also suggest there must be an operational cost model which should be implemented at initial level.


ACADEMIC SOLUTIONS

Due to fast growth of the Internet, network structures have become large and complicated [100]. This complexity initiates a huge amount of traffic data and it has become a challenge to take traffic measurements, like, traffic classification and prediction in a network. To manage networks efficiently ML is used in SDN. SDN controller measures all parameters to take important decisions about routing and resource allocation. For the SDN controller, some efficient algorithms are needed to measure and extract required data or information from received data. In [101] a network management approach called “Smart-Net System” is proposed in which each data plane device keeps a flow rule in its flow table. If a packet reaches at data plane for the flow rule which exists in flow table then it is forwarded to controller. The controller verifies the behavior of that packet and takes preventive measures to avoid attacks. In [102] a software-defined security (SDS) architecture is presented which is open and universal. It offers an open interface for security services, devices and management which is quite helpful for network security vendors to implement network security products and

solutions. In this research work various attack types which can be vulnerable to the networks are analyzed which is helpful to disable such attacks by modifying security configuration mechanism at server. The above cited works lack in detection of change of network policy as well as packet violations analysis due to the already installed conflicting rules on data plane.

The authors in [103] state that changing or modifying the network policies at controller can lead to policy violations by the data packets for the flows whose flow rules are already installed at data plane. They solve this problem as follows. The copies of flow rules generated by the controller are also stored at the controller. When their proposed approach detects the change in network policies at the controller, the controller removes those flow rules installed at the SDN switches that conflict with the changed network policies. This approach takes longer time to detect the change in policy and subsequently leads to a greater number of packets violating the network policy. In [104], a graph based policy change detection and flow rules installation mechanism is proposed which can detect policy change via graph matching with less amount of time to avoid packet violations and network inefficiencies. In case of policy change, the proposed mechanism detects this change and compute shortest path to install the computed flow rules in addition to deletion of old flow rules. In addition, it helps to resolve network policy conflicts and provides a mechanism which can efficiently installs flow rules at data plane.

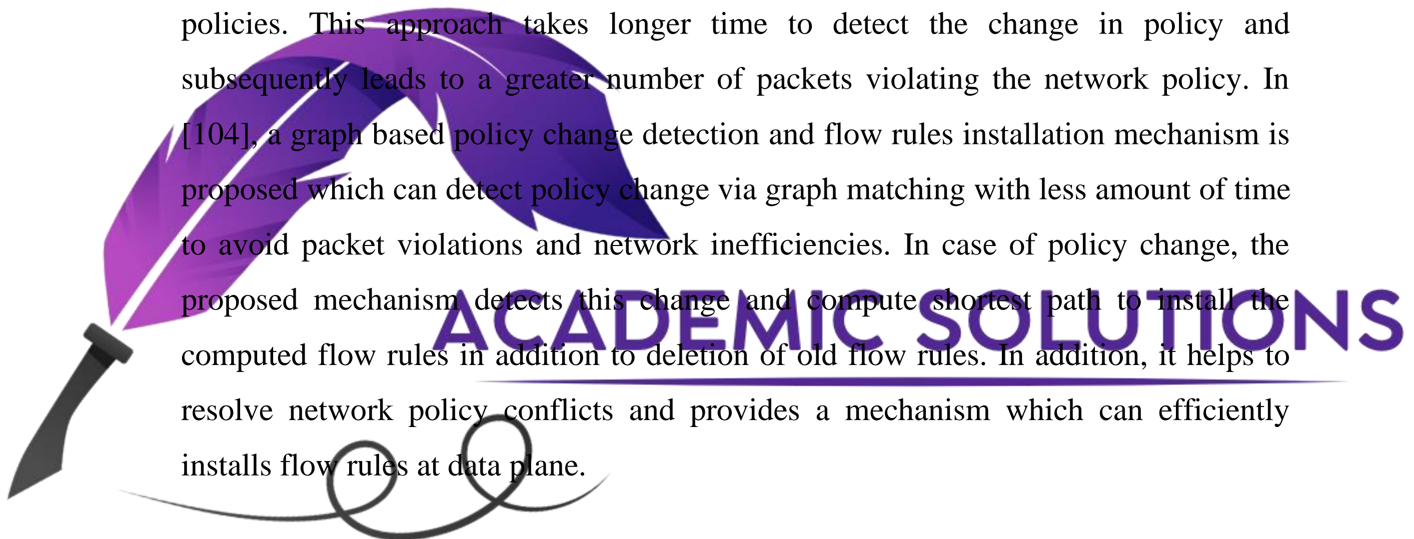


Table 2.3: Summary of Network Security and Management

Study	Purpose	Tools/Techniques	Description
Advance reservation access control [80]	Reservation of intended flow for users and apps	Ryu Controller, OVS switches, ESNet 100G SDN testbed	It guarantees exclusive access of network resources to a certain flow for which the user/app has made the reservation with the help of token-based authorization.
Verifying Reachability [81]	Verification of networks	Z3 version 4.4.2, Intel Xeon	Complex networks are sliced into small networks as per the

	comprising middleboxes	processors with 256 GB of RAM	network-wide verifications for the correctness properties.
Leveraging SDN layering to systematically troubleshoot networks [82]	Precisely localize sources of errant control logic	OpenFlow, TCAM	It helps network admins to troubleshoot bugs including root causes in their networks to verify that networks operate correctly.
Srv: Switch-based rules verification in SDN [83]	Security issue of the priority-based mechanism	Floodlight, Java, OpenFlow App	Leveraging SDN controller to obtain overall network view of whole topology and detects the malicious flow rules.
SDN-Actors: Modeling and Verification of SDN Programs [84]	Provision of network verification mechanism	Erlang, Scala, Akka, OpenFlow	Models network applications using Actors and verifies various properties via existing model checking mechanisms.
Reverse update [85]	Guarantee flow properties during transition	Python, OpenFlow Switch, Naive Controller	It is based on the concept that in-transit packets are always handled in the next forwarding hops by the same or a more recent policy.
SVM [86]	Detection of DDoS attacks in SDN	Mininet Emulator, Floodlight Controller	Switch collects flow status information of network traffic using SVM algorithm to detect DDoS attacks.
Efficient handling of policy change in SDN [103, 104]	Efficiently handling network policy change in SDN	Mininet, POX, OpenFlow, Python, OVS Switch	The proposed approaches detect the change in network policies at the controller install flow rules at data plane efficiently.

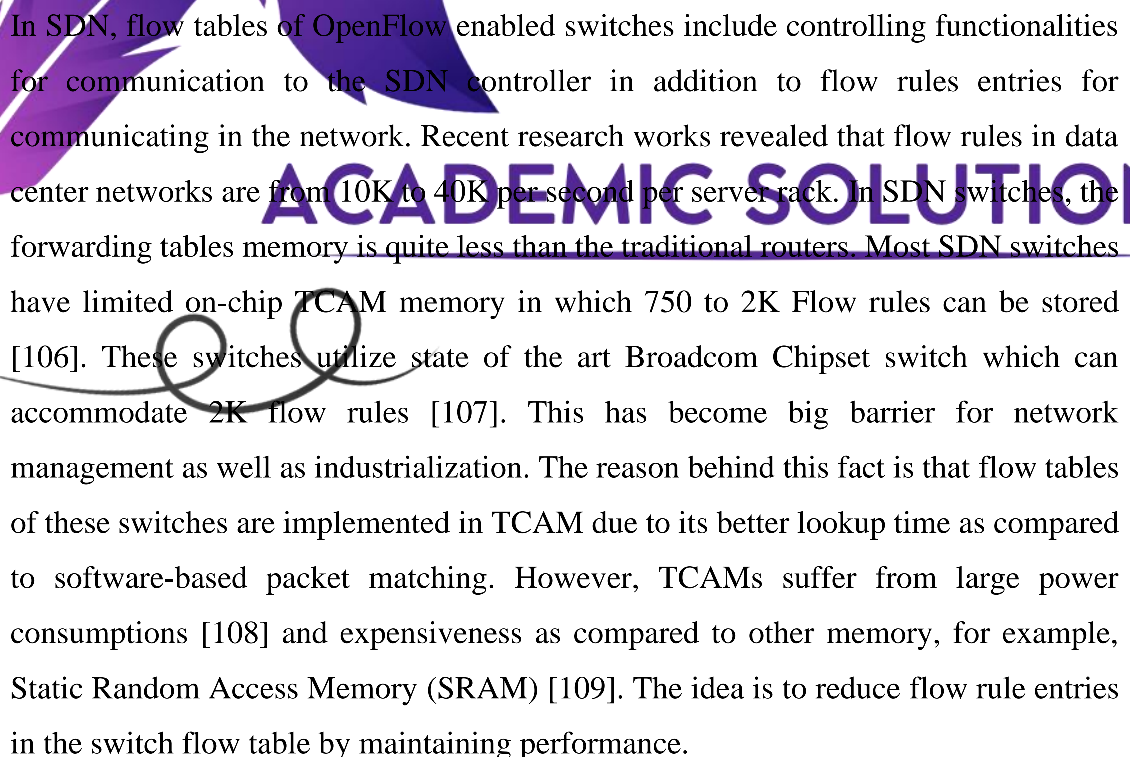
ACADEMIC SOLUTIONS

The summary of network security and management studies is presented in Table 2.3. In [80] a token-based authorization mechanism is proposed which guarantees exclusive

access of network resources to a certain flow for which the user/app has made the reservation. In [81] network verification is performed which consists of middleboxes whose forwarding behavior depends on previously observed traffic. The research work in [82] helps network administrators in a way that they can troubleshoot bugs including root causes of bugs in their networks to verify that networks operate correctly. Similarly, the research work in [85] help network admins in implementing networks in a managed and consistent manner to install flow rules as per network policies. The research works in [83, 87] detect security issues in the network and warn network admin to deal with those identified issues. However, the above presented approaches do not deal with the problem when the access rights (network policies) are changed. In addition, these also do not consider the flow rules already installed at data plane as per old polices.

2.5 Memory Management Studies

In SDN, flow tables of OpenFlow enabled switches include controlling functionalities for communication to the SDN controller in addition to flow rules entries for communicating in the network. Recent research works revealed that flow rules in data center networks are from 10K to 40K per second per server rack. In SDN switches, the forwarding tables memory is quite less than the traditional routers. Most SDN switches have limited on-chip TCAM memory in which 750 to 2K Flow rules can be stored [106]. These switches utilize state of the art Broadcom Chipset switch which can accommodate 2K flow rules [107]. This has become big barrier for network management as well as industrialization. The reason behind this fact is that flow tables of these switches are implemented in TCAM due to its better lookup time as compared to software-based packet matching. However, TCAMs suffer from large power consumptions [108] and expensiveness as compared to other memory, for example, Static Random Access Memory (SRAM) [109]. The idea is to reduce flow rule entries in the switch flow table by maintaining performance.


ACADEMIC SOLUTIONS

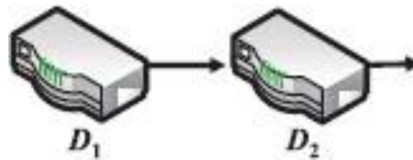
One approach is to efficiently utilize the Forwarding Information Base (FIB) by compression mechanisms to reduce TCAM requirements. It proposes the Expresso heuristic [110] to minimize the logic to compress prefix-based match field which are generated by Optimal Routing Table Constructor (ORTC) algorithm. The simulation

results show that FIB size is reduced by 17% which helps to save TCAMs [111]. Another approach in [112] solves the problem by a Flow Table Reduction Scheme (FTRS) by reducing flow table congestion which help to reduce flow table size. The simulation results suggest that FTRS reduces flow rules in the flow tables by 98% without compromising network performance and efficiency. In [113], a proactive eviction of flow rule entries is proposed for the efficient utilization of TCAM resources inside OpenFlow enabled switches. It is based on intelligent flow management strategy in the SDN controller which combines adaptive idle timeout values for flow rule entries along with proactive eviction mechanism on the current TCAM utilization level. In case of non-matching of packet for a defined idle time period, the respective flow rule is removed from switch flow table. This idle time period is set by the SDN controller before flow rule installation at the data plane. The experimental results show that the proposed scheme SmartTime provides 58% better results in terms of cost as compared to static timeout values or random eviction techniques.

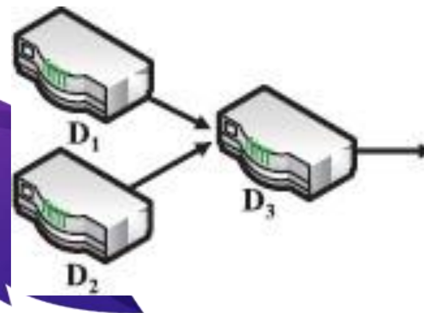
The authors in [114] investigate the effect of flow rule timeout value based on miss rate performance and flow table occupancy of switches. They observe that with the increase in timeout value, miss rate decreases, however, flow table size increases about linearly. They also observe that there is an ideal timeout value where miss rate is ideal and flow table size is also optimal and with the increase in that particular timeout value, flow table size increases in addition to affect miss rate. In this research work, a hybrid flow table management mechanism is proposed which combines timeout value with explicit control plane eviction messages. The proposed scheme is able to reduce flow table size by a lower bound of 57% without affecting miss rate. However, in case of TCP-based application the flow table size decreases around 42%. In addition, this research work analyses performance of various flow table eviction techniques and finds that LRU strategy outperforms all others. However, it cannot be implemented in current SDN switches. Moreover, First-In First-Out (FIFO) strategy does not provide better results than LRU but still it is better than random replacement strategies by 0.1%. The research work in [115], addresses the problem of flow rule placement in firewalls on the basis of ACLs. It aims to reduce the number of flow rules in flow tables of switches by considering conflicts as well as redundancies along with the relationships between neighboring devices. It categorizes relationships of neighboring devices into two groups, i.e. serial relationship and parallel relationship as shown in Figure 2.11 (a) and

ACADEMIC SOLUTIONS

(b) respectively. These relationships depend upon flow of data from one device to another. As shown in Figure 2.11(a), the data is flowing from D_1 to D_2, so it is serial relationship. Similarly, as shown in Figure 2.11(b), the data is not flowing between D_1 and D_2, so it is parallel relationship. Based on these relationships, flow rules are categorized and placed into the devices to reduce number of flow rules.



(a) Serial Relationship



(b) Parallel Relationship

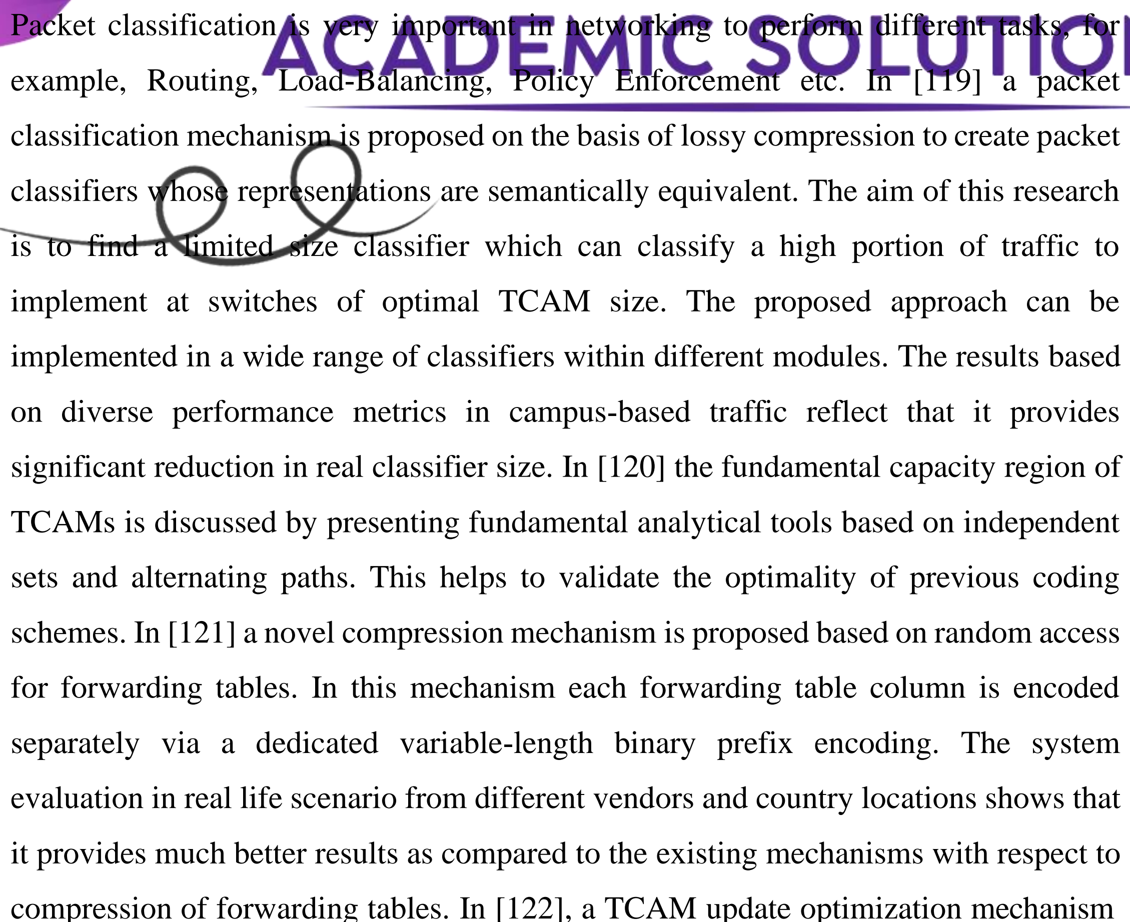
Figure 2.11: Devices Relationships for Flow Rules Categorization [115] to Place Flow Rules in Firewalls

ACADEMIC SOLUTIONS

There are two key challenges to implement the above strategies. The first challenge is to check that a flow rule which is going to be placed in a device is part of a specific rule set or not. The second one is to check that whether the flow rule can be merged with the other flow rules or not. This research resolves these challenges by proposing a novel data structure called OPTree to check that whether the flow rule belong to other or can be merged or not. In addition, it proposes flow rules insertion and search algorithms to resolve the identified problem. The results indicate that the proposed approach considerably reduces number of flow rules.

In [116], a flow-rule placement mechanism called FlowStat is proposed which provides per-flow statistics to SDN controller in addition to improve performance of the network. The proposed mechanism comprises three phases. In first phase, a max-flow-min-cost, which is an optimization problem, is formulated to find the optimal forwarding paths.

In second phase, forwarding flow rules for the identified optimal paths are computed via formulating an Integer Linear problem (ILP) in order to minimize exact-match flow rules in the flow tables of switches to reduce rule-space utilization and to accommodate more flow rules. This is achieved with the help of two greedy heuristic approaches to solve the problem in polynomial time. In third phase, a flow rule redistribution mechanism is proposed by detecting flow rule congestion at the switches so that new flows can be accommodated in the network. The results suggest that the proposed approach provides per flow statistics and improves network performance as compared to existing approaches, like ReWiFlow [117] and ExactMatch. In [118], a novel flow rule placement algorithm called hybrid flow table architecture is proposed which exploits benefits of both hardware and software flow table implementations. The decision of placement of flow rules in the flow tables (hardware (expensive) or software (cheap)) of switches is handled dynamically to increase software-based flow tables utilizations which are Non-TCAM modules. This mechanism helps to save expensive TCAM memory of switches without degrading network performance in terms of packet delay and packet loss.



Packet classification is very important in networking to perform different tasks, for example, Routing, Load-Balancing, Policy Enforcement etc. In [119] a packet classification mechanism is proposed on the basis of lossy compression to create packet classifiers whose representations are semantically equivalent. The aim of this research is to find a limited size classifier which can classify a high portion of traffic to implement at switches of optimal TCAM size. The proposed approach can be implemented in a wide range of classifiers within different modules. The results based on diverse performance metrics in campus-based traffic reflect that it provides significant reduction in real classifier size. In [120] the fundamental capacity region of TCAMs is discussed by presenting fundamental analytical tools based on independent sets and alternating paths. This helps to validate the optimality of previous coding schemes. In [121] a novel compression mechanism is proposed based on random access for forwarding tables. In this mechanism each forwarding table column is encoded separately via a dedicated variable-length binary prefix encoding. The system evaluation in real life scenario from different vendors and country locations shows that it provides much better results as compared to the existing mechanisms with respect to compression of forwarding tables. In [122], a TCAM update optimization mechanism

is presented which ensures consistent packet forwarding. This mechanism is based on a modified-entry-first write-back scheme which considerably decreases TCAM entries movements overhead and detects reordering cases with the help of efficient solutions. However, none of the above approaches talk about network policy change and management of flow rules in TCAMs of switches.

Table 2.4: Summary of Memory Management Studies

Study	Purpose	Tools/Techniques	Description
Constructing optimal IP routing tables [110]	To optimize routing tables for better TCAMs utilization	Internet Backbone routing tables (MaeEast, AADS, MaeWest, Paix)	It uses expresso heuristic to minimize logic to compress prefix-based match field which is generated by Optimal Routing Table Constructor algorithm.
Effective switch memory management [113]	Efficient utilization of switch memory in OpenFlow networks	Floodlight, Open Daylight, OVS Switch	It is based on intelligent flow management strategy which combines adaptive idle timeout values for flow rule entries along with proactive eviction mechanism.
OpenFlow timeouts demystified [114]	To analyze the effect of flow rule timeout value	OpenFlow 1.2, CAIDA/32 Dataset, UNIV dataset	It proposes a hybrid flow table management scheme that combines timeout value with explicit control plane eviction messages to reduce flow table size.
OPTree [115]	Optimal Flow rule placement	C++, Binary Search	It addresses the problem of flow rule placement in firewalls based on ACLs to reduce the number of flow rules in flow tables.

ACADEMIC SOLUTIONS

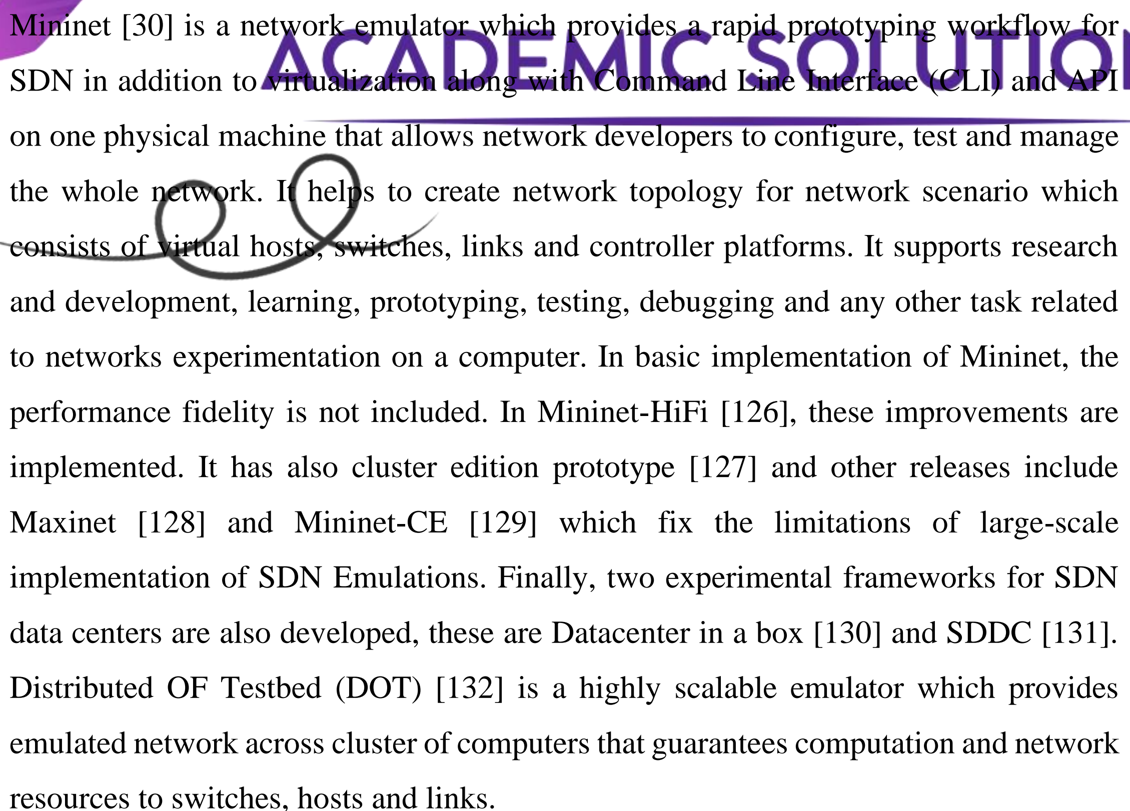
Flowstat [116]	To propose Flow-rule placement scheme to maximize number of flow rules in the network	Mininet Network Emulator, POX SDN controller	Provides per-flow statistics to controller and improve network performance by formulating max-flow-min-cost to find optimal forwarding paths, flow rules computation and redistribution.
Lossy compression of packet classifiers [119]	To classify the traffic for optimal TCAM usage of switches	10 Gigabit Ethernet port of a Cisco 6500 Layer3 switch, Wireshark	Find a limited size classifier which can classify a high portion of traffic to implement at switches of optimal TCAM size.
Compressing forwarding tables for datacenter scalability [121]	To investigate the compressibility of forwarding table	TCAM, Switches	It presents a compression mechanism based on random access for forwarding tables where each table column is encoded separately via a dedicated variable-length binary prefix encoding.

ACADEMIC SOLUTIONS

The summary of memory management studies is presented in in Table 2.4. These studies show the efficient utilization the TCAM resources of switches. In [110] an Expresso heuristic approach is proposed which is based on Optimal Routing Table Constructor (ORTC) algorithm to optimize routing tables. Similarly, the research in [113] represents effective switch memory management scheme for efficient TCAM utilization. OPTree [115] and FlowStat [116] represent two flow rule placement strategies to reduce the number of flow rules in flow tables of switches. In addition, these flow rules are installed on optimal paths to avoid congestion in the network. In [119-121] flow rules compression mechanisms are proposed for efficient memory management of switches. However, these research works do not consider policy change mechanism in addition to analyze packet violations.

2.6 SDN Simulators and Emulators

To analyze the network performance instead of implementing a large experimental testbed, there are two commonly used methods, these are called simulation and emulation. The simulation method provides application environment where we can test our implemented software program without real deployment. The emulation utilizes software program to perform executions with real devices by interacting them as when required. To analyze network performance by simulation is inexpensive, flexible, controllable, and scalable as compared to emulator. In addition, the simulators allow researchers to analyze and test network behaviors as per defined workload. In SDN, with the development of OpenFlow protocol, the simulation tools have extended support to additional network components for the testing and experimentation of OpenFlow based network applications. Moreover, network emulators based on software switches have also been developed to test and analyze network applications, for example, Open vSwitch (OvS) [123], ofsoftswitch13 [124], Indigo Virtual Switch (IVS) [125] etc.



Mininet [30] is a network emulator which provides a rapid prototyping workflow for SDN in addition to virtualization along with Command Line Interface (CLI) and API on one physical machine that allows network developers to configure, test and manage the whole network. It helps to create network topology for network scenario which consists of virtual hosts, switches, links and controller platforms. It supports research and development, learning, prototyping, testing, debugging and any other task related to networks experimentation on a computer. In basic implementation of Mininet, the performance fidelity is not included. In Mininet-HiFi [126], these improvements are implemented. It has also cluster edition prototype [127] and other releases include Maxinet [128] and Mininet-CE [129] which fix the limitations of large-scale implementation of SDN Emulations. Finally, two experimental frameworks for SDN data centers are also developed, these are Datacenter in a box [130] and SDDC [131]. Distributed OF Testbed (DOT) [132] is a highly scalable emulator which provides emulated network across cluster of computers that guarantees computation and network resources to switches, hosts and links.

OFNET [133] emulator provides built-in functionalities to test, debug, traffic generation and monitoring tools which help researchers in debugging process. Virtual Network Overlay (ViNO) [134] network emulation tool provides functionalities that help to create arbitrary network topologies via Open vSwitches and virtual machines. The overlay interconnection between Virtual Machines (VMs) is provided by VXLAN encapsulation [135]. EstiNet [136] provides benefits of both simulation and emulation tools by offering each host with real Linux OS environment and any real application program can run on a simulated host without any modification. FS-SDN [137] is a simulator that is based on the FS [138] simulation platform and is developed in Python language for realistic test and validation of standard networks.

OMNeT++ [139,140] is a network simulator which is developed in C++ language for network modeling, multiprocessors and different distributed or parallel systems. It utilizes INET Framework [141] for the simulations in SDN application environments by integrating OpenFlow components, basic switch functions, basic controllers and OpenFlow messages, for example, "PACKET_IN", "PACKET_OUT", "FLOW_MOD" etc. NS-3 [142] is a network simulator which is implemented in C++ that can use OpenFlow switches. These switches are configurable with the help of OpenFlow API which is designed to express basic use of OpenFlow protocol by maintaining its flow tables and TCAM resources. It implements its own OpenFlow controller which simulates the behavior of a real controller. External modules can be used to extend NS-3, such as OFSwitch13 [143] which brings compatibility with OpenFlow 1.3.

ACADEMIC SOLUTIONS

Table 2.5: Summary of Network Emulators and Simulators

Name	Language	Purpose	Description
Mininet [30]	Python	Network Emulation	It provides a rapid prototyping workflow for SDN in addition to virtualization along with Command Line Interface (CLI) and API on one physical machine that allows network developers to configure, test and manage the whole network.

Distributed OF Testbed (DOT) [132]	Java	Network Emulation	This Emulator provides emulated network across cluster of computers that guarantee computation and network resources to switches, hosts and links.
OFNET [133]	Python	Network Emulation	It provides built-in functionalities to test, debug, traffic generation and monitoring tools.
ViNO [134]	Java	Network Emulation	This tool helps to create arbitrary network topologies via Open vSwitches and virtual machines.
EstiNet [136]	C	Network Simulation and Emulation	It offers each host with real Linux OS environment, and any real application program can run on a simulated host without any modification.
FS-SDN [137]	Python	Network Simulation	This simulator is based on the FS [119] simulation platform which is developed in Python language for realistic test and validation of standard networks.
OMNeT++ [139,140]	C++	Network Simulation	This simulator is used in network modeling, multiprocessors and different distributed or parallel systems.
NS-3 [143]	C++	Network Simulation	This network simulator provides support to OpenFlow which help to program network devices.

The summary of network simulators and emulators is presented in Table 2.5. It provides a comprehensive overview of simulators and emulators which are developed in various programming languages, like, C, C++, Java, Python etc. These tools help in developing and simulating SDN applications in SDN environment.

ACADEMIC SOLUTIONS

2.7 SDN Programming Languages

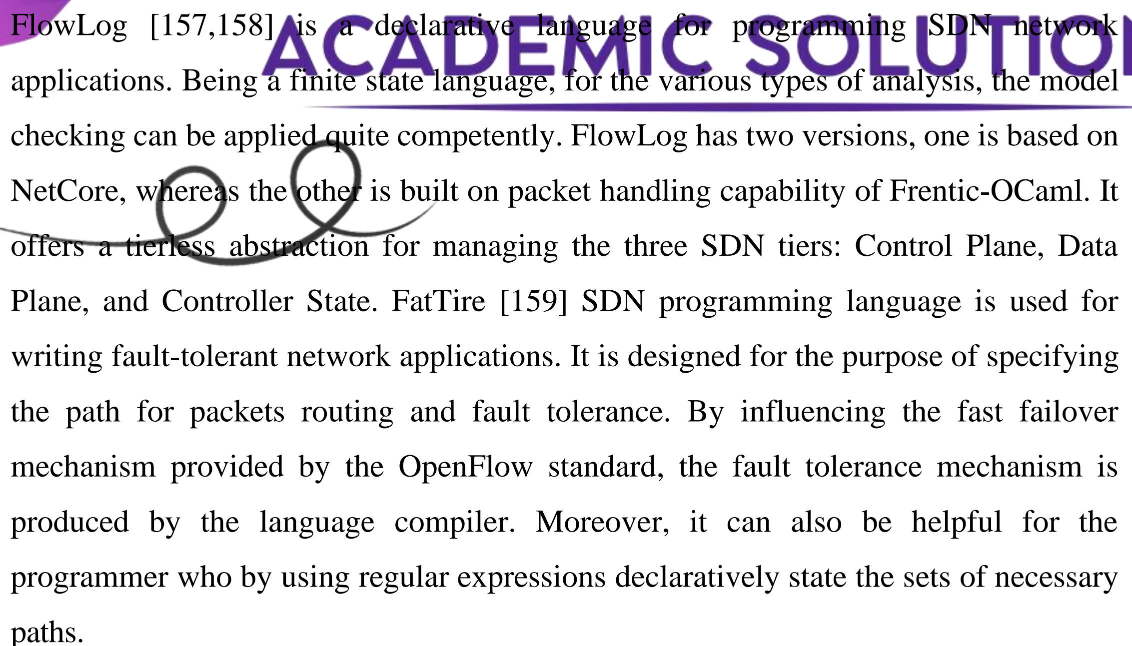
SDN programming languages consist of compilation and validation tools which are helpful for the translation of high-level constructs into messages understandable by the SDN controller API. Some SDN programming languages are explained in this section.

One of the languages is Frenetic [144,145] that is high-level language for the programming of OpenFlow networks and is useful for the categorization and accretion of network traffic. Moreover, it is also helpful for defining packet forwarding policies on the basis of functional reactive combinator library inspired on Yampa [146] and its implementation is based on FlapJax [147]. By providing frenetic runtime environment, facilities pertaining to installation and querying low-level details are managed. In addition, it provides compositional constructs which facilitate modular reasoning and enable code reuse. As NetCore [148] is the successor of the Frenetic, therefore, it carries enhanced policy management library. Moreover, it has also capacity to compile ACL policies and handle the interaction between controller and switch. In addition to it, for the efficient generation of flow rules, run-time system of NetCore is designed. Nettle [149] is a low-level programming language which deals with streams and does not deal with events. It is quite appropriate for various functions like, programming controllers, programming discrete and continuous operations. Moreover, dynamic policies including traffic engineering and load balancing are also generated through it. As it is declarative in nature, therefore, functions like time sensitive and varying can be demarcated. Moreover, sequential operator provided by nettle can also be used for creating compound commands.

Procera [150] is a high-level programming language that can be used to delineate policies regarding networks. It is quite resourceful for the operators as it provides expressive and extensible compositional framework. Moreover, it is also quite useful for designing network applications which not only react to the events produced by OpenFlow switches but also to the external events, for instance, a user authorization and bandwidth usage. Procera was used in several campus networks as well as home networks prototype deployments [151]. Flow-based Management Language (FML) [152] being a high-level declarative language is based on non-recursive Datalog [153] for handling network whose aim is to provide efficient and flexible policies. Moreover,

ACADEMIC SOLUTIONS

it is also helpful for the operators as it provides them eminent management facilities for configuring ACL policies straightforwardly. Flog [154] is an event-driven programming that has adopted ideas from FML and Frenetic and it is based on logic programming for SDN environment on the similar lines like FML. It is composed of three components that are similar to Frenetic. These components include a mechanism for network state collection, information processing and policy generation. Like NetCore, Frenetic-OCaml [155] is also successor of Frenetic. It is beneficial in providing mechanisms for network wide policy implementation. The NetCore is used and is replaced with NetKat for forwarding decision. Moreover, its query language permits querying statistics that include traffic and topology. The core language utilizes the proactive flow rule installation and handles the low-level details of the switch to controller. Being imperative programming paradigm-based language, Pyretic [11,156] by specifying the static and dynamic forwarding policies assists in developing network applications. By utilizing the sequential and parallel operators provided by Pyretic, the forwarding policies can be specified. In addition, it also offers the abstraction of topology and consents the formation of virtual header.



FlowLog [157,158] is a declarative language for programming SDN network applications. Being a finite state language, for the various types of analysis, the model checking can be applied quite competently. FlowLog has two versions, one is based on NetCore, whereas the other is built on packet handling capability of Frentic-OCaml. It offers a tierless abstraction for managing the three SDN tiers: Control Plane, Data Plane, and Controller State. FatTire [159] SDN programming language is used for writing fault-tolerant network applications. It is designed for the purpose of specifying the path for packets routing and fault tolerance. By influencing the fast failover mechanism provided by the OpenFlow standard, the fault tolerance mechanism is produced by the language compiler. Moreover, it can also be helpful for the programmer who by using regular expressions declaratively state the sets of necessary paths.

NetKat [160-163] uses Kleene algebra with Tests (KAT) [164]. This programming language is based on equational theory, for programming and reasoning about the networks. A regular expression can be used for describing end-to-end paths and its semantics is inspired by the NetKat. Moreover, NetKat is also beneficial for defining

virtual topologies. Merlin [165-168] is composed of declarative language and is useful for distributing and coordinating the policy implementation. Moreover, it is the network management framework for programming the network policies, for implementation, it uses Frenetic and OCaml as a core layer. Its run-time monitor is consumed for analyzing the incoming and outgoing traffic on end-host. Being policy specification for SDN, PonderFlow [169] aims to extend the Ponder language [170] for describing OpenFlow flow rules. Ponder language is used to define management and security policies in distributed systems. PonderFlow provides mechanisms for implementing access control and network abstractions.

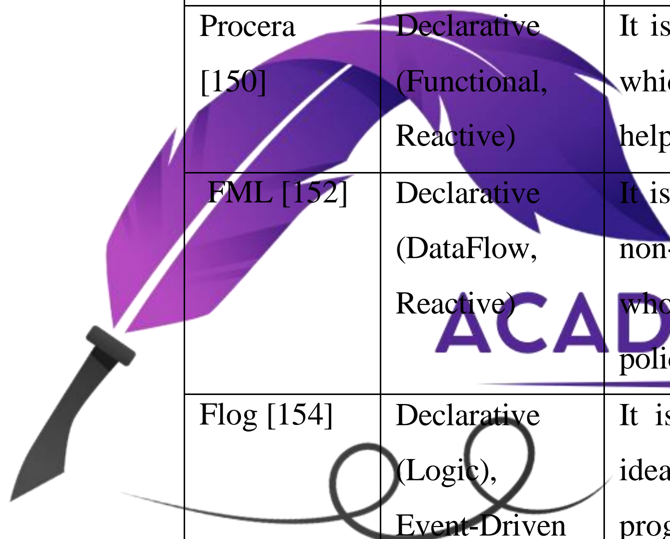
NOF [171] is a programming language with objective of enabling network application to design the network according to the application requirements. It comprises sets of operations and services and these are categorized into three groups, namely, Matching, Timing and Query. Operation can be applied onto conventional network fields which is based on host information. Timing includes information about the services, that is, when they will be installed and how long they will remain functional in the network. Query operation helps to get the network state information, that is, link state, bandwidth usage, transmission errors etc. NOF not only provides flexibility to compile the NOF program to other SDN languages, but it also offers an interface to generate the commands for end-host configurations automatically. Kinetic [172] is a domain specific language which helps network operators to control the dynamic state of their network. In addition, it also offers facilities to validate the accuracy of control programs. The network policies may be stated with respect to Finite State Machines (FSMs) that aids in encapsulating the dynamic state of network. Moreover, Kinetic also inherits runtime features of Pyretic.

ACADEMIC SOLUTIONS

Table 2.6: Summary of SDN Programming Languages

Name	Programming Paradigm	Description
Frenetic [144, 145]	Declarative (Functional)	It is high-level language for the programming of OpenFlow networks and is useful for the categorization and accretion of network traffic. Moreover, it provides compositional constructs

		which facilitate modular reasoning and enable code reuse.
NetCore [148]	Declarative (Functional)	It is the successor of the Frenetic, therefore, it carries enhanced policy management library. Moreover, it has also capacity to compile ACL policies and to handle the interaction between controller and switch.
Nettle [149]	Declarative (Functional, Reactive)	It is based on low-level programming language which deals with streams and not with events. Moreover, dynamic policies including traffic engineering and load balancing are also generated through it.
Procera [150]	Declarative (Functional, Reactive)	It is based on declarative Programming paradigm which contains a set of high-level abstractions that help in portraying reactive and temporal behaviors.
FML [152]	Declarative (DataFlow, Reactive)	It is a high-level declarative language is based on non-recursive Datalog [134] for handling network whose aim is to provide efficient and flexible policies.
Flog [154]	Declarative (Logic), Event-Driven	It is an event-driven programming that adopted ideas from FML and Frenetic and is based on logic programming for SDN environment on the similar lines like FML.
Frenetic-OCaml [155]	Declarative (Functional)	It is beneficial in providing mechanisms for network wide policy implementation. The core language utilizes the proactive flow rule installation and handles the low-level details of the switch to controller.
Pyretic [11,156]	Imperative	It helps in specifying the static and dynamic forwarding policies to assist in developing network applications by utilizing the sequential and parallel operators.


ACADEMIC SOLUTIONS

FlowLog [157, 158]	Declarative (Functional)	It is a declarative language for programming SDN network applications. Being a finite state language, for the various types of analysis, the model checking can be applied quite competently.
FatTire [159]	Declarative (Functional)	It is used for writing fault-tolerant network applications.
NetKat [160-163]	Declarative (Functional)	It uses Kleene algebra with Tests (KAT) [145]. This programming language is based on equational theory, for programming and reasoning about the networks.
Merlin [165-168]	Declarative (Logic)	It is based on declarative language and is useful for distributing and coordinating the policy implementation.
PonderFlow [169]	Policy Specification Language	PonderFlow provides mechanisms for implementing access control and network abstractions.
NOF [171]	Declarative	Its objective is to enable network application to design the network according to the application requirements.
Kinetic [172]	Domain Specific Language	It helps network operators to control the dynamic state of their network and it also inherits runtime features of Pyretic.

ACADEMIC SOLUTIONS

The summary of SDN programming languages is presented in Table 2.6. Different SDN programming languages are developed to handle specific problems or to provide specific functionalities in network applications in more refine and abstract manner. We have described these languages along with their programming paradigms. These languages help network administrators to implement access control, develop and test network applications on the basis of low-level constructs as well as high level abstractions.

2.8 SDN Controller Platforms

The controllers are the brain of SDN networks and act as strategic control point. These contain collection of modules that can perform different network tasks including network topology, network statistics, etc. Different network applications like network policies are installed on the controllers for data communication between end nodes. In this section, we have discussed some controller platforms which are described below.

Beacon [173] is implemented in Java programming language and uses centralized architecture. Moreover, it uses ad-hoc northbound API and Southbound API with OpenFlow 1.0. It supports CLI and web UI. In addition, it also supports multicommand line threading and modularity functionality. It serves as basis of Floodlight. To conclude, with a focus on being developer friendly, high performance and with the ability to start and stop existing approaches, it has explored areas of OpenFlow controller design. Beehive [174] is a distributed control platform implemented in Go programming language with a distributed hierarchical architecture. It utilizes REST northbound API and Southbound API with OpenFlow 1.0 and 1.2. It utilizes Linux supporting platform and supports CLI. The implementation of DCFabric [175] is based on C and Java script programming language with a centralized architecture. It utilizes REST northbound API and Southbound API with OpenFlow 1.3. It uses Linux supporting platform and supports CLI and Web UI. Moreover, it supports multithreading and have a good modularity functionality with good consistency. Disco [176] is implemented in Java programming language with a Distributed Flat architecture. It utilizes northbound, Southbound and East/Westbound API with REST with OpenFlow 1.0 and AMQP respectively. It supports Proprietary license. It has good modularity with Limited documentation. The implementation of Faucet [177] is based on Python programming language with a centralized architecture. It utilizes Southbound API with OpenFlow 1.3. It employs Linux supporting platform and supports CLI and WebUI. It supports Apache 2.0 license and multithreading with good consistency.

Floodlight [178] is implemented in Java programming language with a centralized architecture. It utilizes REST, Java, RPC and Quantum Northbound API and Southbound API with OpenFlow 1.0 and 1.3. It utilizes Linux, MacOS and Windows

ACADEMIC SOLUTIONS

supporting platform and provides CLI and Web UI. It supports Apache 2.0 license. It supports multithreading and have a fair modularity with good consistency and documentation. Flow Visor [179] is implemented in C programming language with a centralized architecture. It utilizes JSON and RPC northbound API and Southbound API with OpenFlow 1.0 and 1.3. It utilizes Linux supporting platform and supports CLI interface. Moreover, it supports Proprietary license. It has no consistency; however, its documentation is fair. HyperFlow [180] is implemented in C++ programming language with a Distributed Flat architecture. It utilizes Southbound API with OpenFlow 1.0 and East/Westbound API with publishing and subscribing messages. It supports Proprietary license and multithreading and have no consistency. Kandoo [181] is implemented in C, C++ and Python programming language with a distributed hierarchical architecture. It utilizes Java RPC northbound API and Southbound API with OpenFlow 1.0-1.2 and East/Westbound API with messaging channel. It utilizes Linux supporting platform and supports CLI and Proprietary license.

ACADEMIC SOLUTIONS

Loom [182] is implemented in Erlang programming language with a Distributed Flat architecture. It utilizes JSON northbound API and Southbound API with OpenFlow 1.3-1.4. It utilizes Linux supporting platform and supports CLI. It supports Apache 2.0 license and multithreading and have a good modularity with good consistency. However, its documentation is limited. Maestro [183] is implemented in Java programming language with a centralized architecture. It applies ad-hoc northbound API and Southbound API with OpenFlow 1.0. It utilizes Linux, MacOS and Windows supporting platform and supports Web UI. It supports LGPL 2.1 license. It supports multithreading and have a fair modularity with no consistency and its documentation is also limited. MsNettle [184] is implemented in Haskell programming language with a centralized architecture. It utilizes Southbound API with OpenFlow 1.0. It utilizes Linux supporting platform and supports CLI. It supports Proprietary license and multithreading and have a good modularity with no consistency and its documentation is also limited. Meridian [185] is implemented in Java programming language with a centralized architecture. It utilizes REST northbound API and Southbound API with OpenFlow 1.0 and 1.3. It utilizes cloud-based supporting platform and supports WebUI. It supports multithreading and have a good modularity with no consistency and its documentation is also limited.

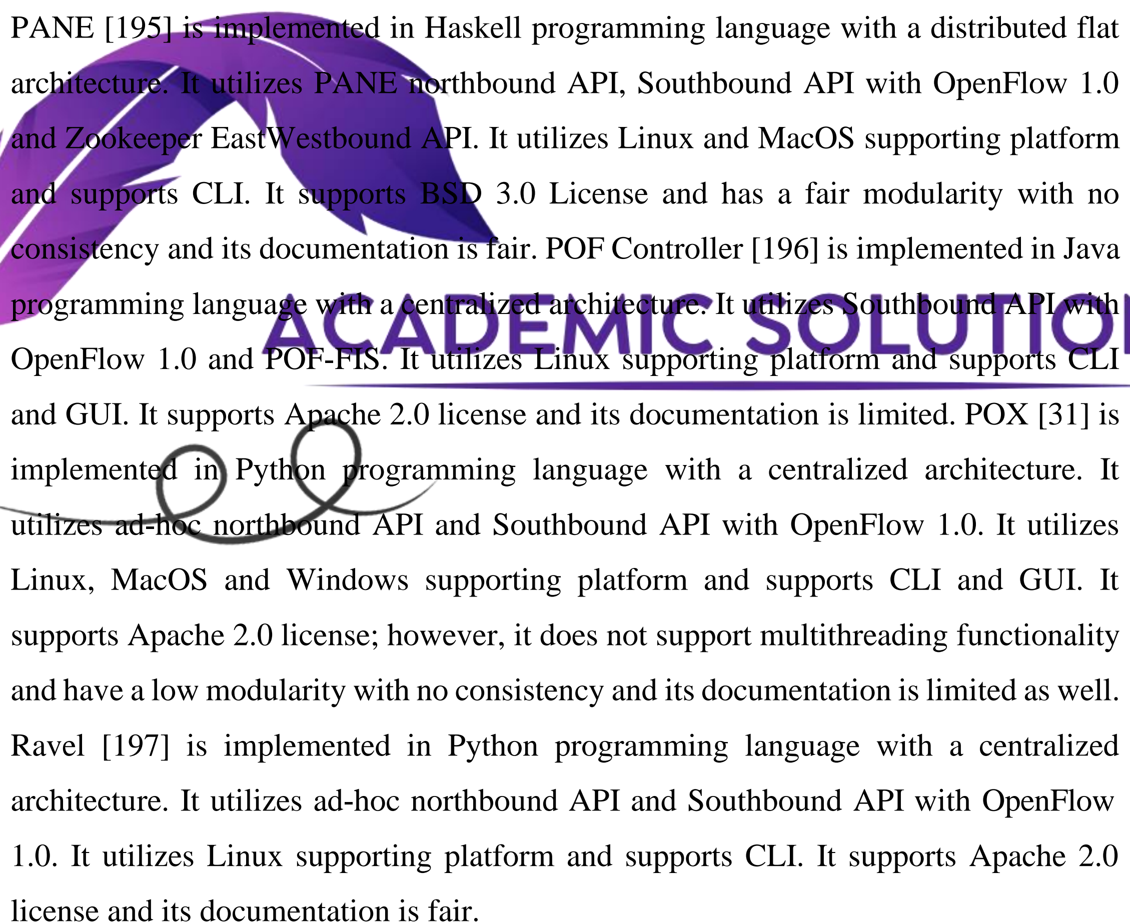
Microflow [186] is implemented in C programming language with a centralized architecture. It utilizes Socket northbound API and Southbound API with OpenFlow 1.0-1.5. It utilizes Linux supporting platform and supports CLI and WebUI. It supports Apache 2.0 license and multithreading; however, its documentation is limited. Nodeflow [187] is implemented in JavaScript programming language with a centralized architecture. It utilizes JSON northbound API and Southbound API with OpenFlow 1.0. It utilizes Node.js supporting platform and supports CLI. It supports Cisco license; however, its documentation is limited. NOX [188] is implemented in C++ programming language with a centralized architecture. It utilizes ad-hoc northbound API and Southbound API with OpenFlow 1.0. It utilizes Linux supporting platform and supports CLI and WebUI. It supports GPL 3.0 license and multithreading (Nox-MT) and have low modularity with no consistency. Moreover, its documentation is also limited. ONIX [189] is implemented in C++ programming language with a distributed flat architecture. It utilizes Onix API northbound API and Southbound API with OpenFlow 1.0 and OVSDB and East/Westbound API with Zookeeper. It supports Proprietary license and multithreading and have a good modularity with no consistency and its documentation is limited.

ACADEMIC SOLUTIONS

ONOS [190] is implemented in Java programming language with a distributed flat architecture. It utilizes REST and Neutron northbound API and Southbound API with OpenFlow 1.0 and 1.3 and EastWestbound API with Raft. It utilizes Linux, MacOS and Windows supporting platform and supports CLI and Web UI. It supports Apache 2.0 license and multithreading functionality and have a high modularity, consistency and good documentation. OpenContrail [191] is implemented in C, C++ and Python programming language with a centralized architecture. It utilizes REST northbound API and Southbound API with BGP and XMPP. It utilizes Linux supporting platform and supports CLI and WebUI. It supports Apache 2.0 license and multithreading functionality and have a high modularity with good consistency. Moreover, its documentation is also good. OpenDaylight [192] is implemented in Java programming language with a distributed flat architecture. It utilizes REST, RESTCONF, XMPP and NETCONE northbound API and Southbound API with OpenFlow 1.0 and 1.3 and East/Weast bound API. It utilizes Linux, MacOS and Windows supporting platform and supports CLI and Web UI. It supports EPL 1.0 license and multithreading functionality and has a high modularity with consistency and its documentation is also

good. OpenIRIS [193] is implemented in Java programming language with a distributed flat architecture. It utilizes REST northbound API, Southbound API with OpenFlow 1.0-1.3 and East/Westbound API with custom protocol. It utilizes Linux supporting platform and supports CLI and Web UI. It supports Apache 2.0 and multithreading functionality and have a fair modularity with no consistency. Moreover, its documentation is also limited. OpenMul [194] is implemented in C programming language with a centralized architecture. It utilizes REST northbound API and Southbound API with OpenFlow 1.0, 1.3, OVSDB and Netconf. It utilizes Linux supporting platform and supports CLI. It supports GPL 2.0 license and multithreading functionality and has a high modularity with no consistency. However, its documentation is good.

PANE [195] is implemented in Haskell programming language with a distributed flat architecture. It utilizes PANE northbound API, Southbound API with OpenFlow 1.0 and Zookeeper EastWestbound API. It utilizes Linux and MacOS supporting platform and supports CLI. It supports BSD 3.0 License and has a fair modularity with no consistency and its documentation is fair. POF Controller [196] is implemented in Java programming language with a centralized architecture. It utilizes Southbound API with OpenFlow 1.0 and POF-FIS. It utilizes Linux supporting platform and supports CLI and GUI. It supports Apache 2.0 license and its documentation is limited. POX [31] is implemented in Python programming language with a centralized architecture. It utilizes ad-hoc northbound API and Southbound API with OpenFlow 1.0. It utilizes Linux, MacOS and Windows supporting platform and supports CLI and GUI. It supports Apache 2.0 license; however, it does not support multithreading functionality and have a low modularity with no consistency and its documentation is limited as well. Ravel [197] is implemented in Python programming language with a centralized architecture. It utilizes ad-hoc northbound API and Southbound API with OpenFlow 1.0. It utilizes Linux supporting platform and supports CLI. It supports Apache 2.0 license and its documentation is fair.



Rosemary [198] is implemented in C programming language with a centralized architecture. It utilizes ad-hoc northbound API and Southbound API with OpenFlow 1.0, 1.3 and XMPP. It utilizes Linux supporting platform and supports CLI. It supports Proprietary and multithreading functionality and has a good modularity with no

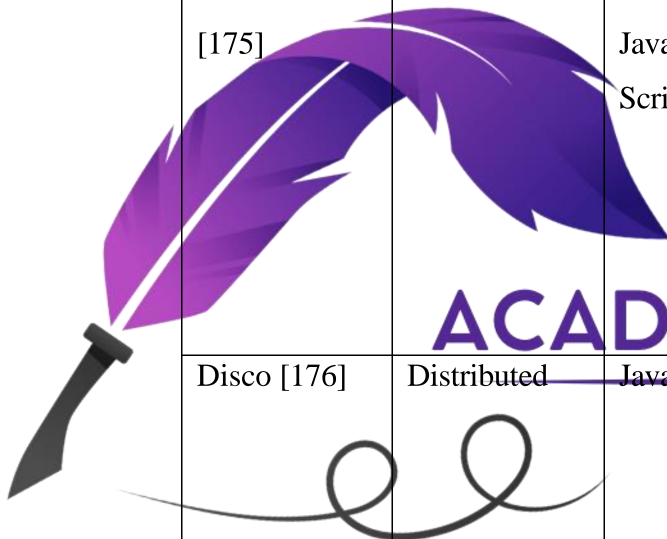
consistency. Moreover, its documentation is limited. RunOS [199] is implemented in C++ programming language with a distributed flat architecture. It utilizes REST northbound API, Southbound API with OpenFlow 1.3 and Maple EastWestbound API. It utilizes Linux supporting platform and supports CLI and WebUI. It supports Apache 2.0 license and multithreading functionality and have a high modularity with consistency and its documentation is fair. Ryu [200] is implemented in Python programming language with a centralized architecture. It utilizes REST northbound API and Southbound API with OpenFlow 1.0-1.5. It utilizes Linux and MacOS supporting platform and supports CLI. It supports Apache 2.0 license and multithreading functionality, moreover it has a fair modularity consistency and its documentation is also good. SMaRtLight [201] is implemented in Java programming language with a distributed flat architecture. It utilizes REST northbound API, Southbound API with OpenFlow 1.3 and BFT-SMaRt EastWestbound. It utilizes Linux supporting platform and supports CLI. It supports proprietary license and has no consistency, moreover, its documentation is also limited.

TinySDN [202] is implemented in C programming language with a centralized architecture. It utilizes Southbound API with OpenFlow 1.0. It utilizes Linux supporting platform and supports CLI. It supports BSD 3.0 License and has no multithreading functionality and consistency. Moreover, its documentation is also limited. Trema [203] is implemented in C and Ruby programming language with a centralized architecture. It utilizes ad-hoc northbound API and Southbound API with OpenFlow 1.0. It utilizes Linux supporting platform and supports CLI. It supports GPL 2.0 License and has a good modularity, whereas it has no consistency. However, its documentation is fair. Yanc [204] is implemented in C and C++ programming language with a distributed flat architecture. It utilizes REST northbound API, Southbound API with OpenFlow 1.0-1.3 capabilities. It utilizes Linux supporting platform and supports CLI. It supports proprietary license and its documentation is limited. ZeroSDN [205] is implemented in C++ programming language with a distributed flat architecture. It utilizes REST northbound API, Southbound API with OpenFlow 1.0 and 1.3 and ZeroMQ of East/Westbound API. It utilizes Linux supporting platform and supports CLI and Web UI. It supports Apache 2.0 license. It has a high modularity and consistency. Moreover, its documentation is also fair.

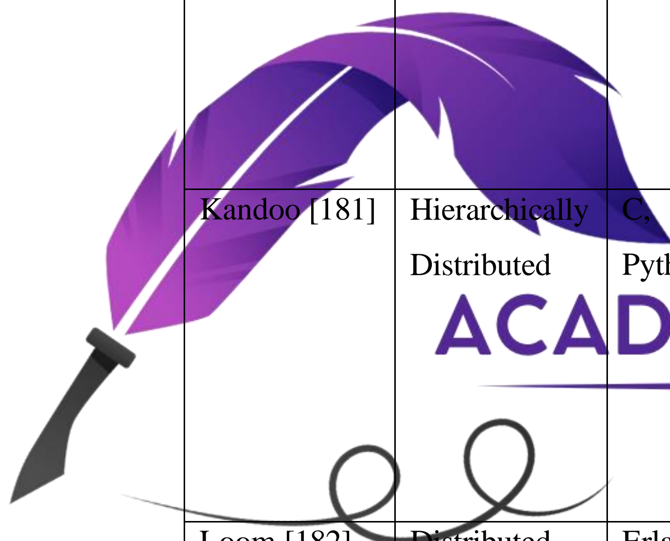
ACADEMIC SOLUTIONS

Table 2.7: Summary of SDN Controller Platforms

Name	Architecture	Language	Description
Beacon [173]	Centralized Multi-threaded	Java	It uses ad-hoc northbound API and Southbound API with OpenFlow 1.0. It offers high performance flow processing capabilities using pipeline threads and shared queues.
Beehive [174]	Distributed	Golang	It utilizes REST northbound API and southbound API with OpenFlow 1.0 and 1.2 and uses Linux supporting platform.
DCFabric [175]	Centralized	C and Java Script	It utilizes REST northbound API and southbound API with OpenFlow 1.3 that supports Linux platform along with CLI and Web UI. Moreover, it supports multithreading and have a modularity functionality with good consistency.
Disco [176]	Distributed	Java	It is based on distributed flat architecture which utilizes northbound, southbound and east/westbound API with REST with OpenFlow 1.0 and AMQP respectively.
Faucet [177]	Distributed	Python	It supports multithreading with good consistency and utilizes Southbound API with OpenFlow 1.3. It utilizes Linux supporting platform and supports CLI and Web UI.
Floodlight [178]	centralized multi-threaded	Java	It supports multithreading and has a fair modularity with good consistency and documentation. It utilizes REST,

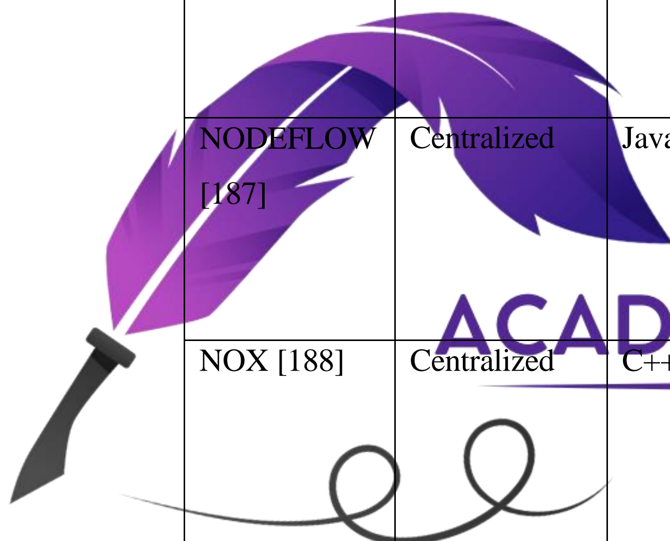

ACADEMIC SOLUTIONS

			Java, RPC and Quantum Northbound API and Southbound API with OpenFlow 1.0 and 1.3.
FlowVisor [179]	Distributed	C	It provides functions to slice the network resources and is located between guest controllers and switching devices acting as a transparent proxy to filter control messages.
HyperFlow [180]	Distributed	C++	It utilizes Southbound API with OpenFlow 1.0 and East/Westbound API with publishing and subscribing messages. It supports Proprietary license and multithreading and have no consistency.
Kandoo [181]	Hierarchically Distributed	C, C++, Python	It utilizes Java RPC northbound API and Southbound API with OpenFlow 1.0-1.2 and EastWestbound API with messaging channel. It utilizes Linux supporting platform and supports CLI and Proprietary license.
Loom [182]	Distributed	Erlang	It provides an experimental network switch controller that implements the OpenFlow 1.3.x and 1.4 protocols. In addition, it offers scalability and robustness for large scale implementations.
Maestro [183]	Centralized Multi-threaded	Java	It proposes a simple programming model for programmers, and exploits parallelism along with additional throughput optimization techniques that supports multithreading and has a fair modularity with no consistency.

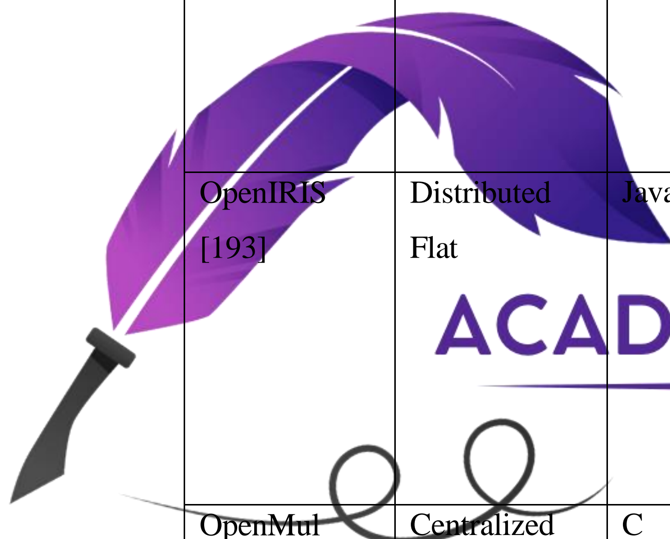


ACADEMIC SOLUTIONS

MsNettle [184]	Centralized Multi-threaded	Haskell	It utilizes southbound API with OpenFlow 1.0 and Linux supporting platform in CLI mode. It supports Proprietary license and has a good modularity.
Meridian [185]	Centralized	Java	It utilizes REST northbound API and Southbound API with OpenFlow 1.0 and 1.3. Moreover, it uses cloud-based platform and supports Web UI.
Microflow [186]	Centralized	C	It utilizes Socket northbound API and Southbound API with OpenFlow 1.0-1.5. It utilizes Linux supporting platform with both CLI and Web UI modes.
NODEFLOW [187]	Centralized	JavaScript	It utilizes JSON northbound API and Southbound API with OpenFlow 1.0. It supports Cisco license, however, its documentation is limited.
NOX [188]	Centralized	C++	It utilizes ad-hoc northbound API and Southbound API with OpenFlow 1.0. It supports Linux platform in CLI and Web UI modes.
ONIX [189]	Distributed Flat	C++	It utilizes Onix API, northbound API and Southbound API with OpenFlow 1.0 and OVSDB and East/Westbound API with Zookeeper.
ONOS [190]	Distributed Flat	Java	It utilizes REST and Neutron northbound API and Southbound API with OpenFlow 1.0 and 1.3 and EastWestbound API with Raft. It uses Linux, MacOS and Windows supporting platform and supports CLI and Web UI.


ACADEMIC SOLUTIONS

OpenContrail [191]	Centralized	C, C++, Python	It utilizes REST northbound API and Southbound API with BGP and XMPP. It uses Linux supporting platform and supports CLI and Web UI. It supports Apache 2.0 license and multithreading functionality and has a high modularity with good consistency.
OpenDaylight [192]	Distributed Flat	Java	It utilizes REST, RESTCONF, XMPP and NETCONF northbound API and southbound API with OpenFlow 1.0 and 1.3 and East/Westbound API. It utilizes Linux, MacOS Windows supporting platform and supports CLI and Web UI.
OpenIRIS [193]	Distributed Flat	Java	It utilizes REST northbound API, Southbound API with OpenFlow 1.0-1.3 and East/Westbound API with custom protocol. It utilizes Linux supporting platform and supports CLI and Web UI.
OpenMul [194]	Centralized	C	It utilizes REST northbound API and Southbound API with OpenFlow 1.0, 1.3, OVSDB and Netconf. It supports Linux platform in CLI mode.
PANE [195]	Distributed Flat	Haskell	It utilizes PANE northbound API, Southbound API with OpenFlow 1.0 and Zookeeper East/Westbound API. It utilizes Linux and Mac OS supporting platform and supports CLI. It supports BSD 3.0 License and has a fair modularity with no consistency.



POF Controller [196]	Centralized	Java	It utilizes Southbound API with OpenFlow 1.0 and POF-FIS. It utilizes Linux supporting platform and supports CLI and GUI.
POX [31]	Centralized	Python	It utilizes ad-hoc northbound API and southbound API with OpenFlow 1.0. It utilizes Linux, MacOS and Windows platform.
Ravel [197]	Centralized	Python	It utilizes ad-hoc northbound API and southbound API with OpenFlow 1.0. It utilizes Linux supporting platform and supports CLI.
Rosemary [198]	Centralized	C	It utilizes ad-hoc northbound API and Southbound API with OpenFlow 1.0, 1.3 and XMPP. It utilizes Linux supporting platform and supports CLI.
Ryu [200]	Centralized	Python	It utilizes REST northbound API and Southbound API with OpenFlow 1.0-1.5. It utilizes Linux and MacOS supporting platform and supports CLI.
SMaRtLight [201]	Distributed	Java	It utilizes REST northbound API, southbound API with OpenFlow 1.3 and East/Westbound API. It supports Linux platform in CLI mode with proprietary license and has no consistency.
TinySDN [202]	Centralized	C	It utilizes southbound API with OpenFlow 1.0 and supports Linux with CLI mode. It supports BSD 3.0 license and has no multithreading functionality and consistency.
Trema [203]	Centralized	C, Ruby	It utilizes ad-hoc northbound API and southbound API with OpenFlow 1.0.

ACADEMIC SOLUTIONS

			It supports Linux platform in CLI mode.
Yanc [204]	Distributed Flat	C, C++	It utilizes REST northbound API, Southbound API with OpenFlow 1.0-1.3. It utilizes platform and supports CLI.
ZeroSDN [205]	Distributed Flat	C++	It uses REST northbound API, Southbound API with OpenFlow 1.0 and 1.3. It supports Linux platform with CLI and Web UI modes.

The summary of SDN Controllers is presented in Table 2.7. These manage flow control to the switches/routers via southbound APIs and the applications/business logic via northbound APIs to deploy intelligent networks. Based on the instructions (flow rules) from the controllers, the data plane devices (switches/routers) perform required functionalities, like forwarding and dropping, etc. There are different kinds of SDN controllers (Centralized and Distributed) which are developed to perform various functionalities in different programming languages (Python, C, C++, Java etc.). Some are discussed in section 2.8 along with their specifications. In our simulation environment, POX SDN controller is used due to its good integration with OpenFlow switches and flow rule installation mechanisms. In addition, it provides a rich collection of libraries to test and implement network policies.

ACADEMIC SOLUTIONS

2.9 Summary

In this chapter, we have discussed and analyzed the existing research work related to the identified research problem in detail along with the limitations. We organized the existing research work into seven categories, and these include network testing and verification, flow rule installation mechanisms, network security and management, memory management, SDN Simulators and Emulators, SDN programming languages and SDN controller platforms. Finally, we have shown each category in tabular form by describing the purpose, tools/techniques, architecture and short description of each technique.

Chapter 3

Problem Statement

3.1 Introduction

In this chapter, the research problem of efficient handling of network policy change in SDN is explained. In section 3.2, network policy change and flow rules installation scenario in SDN is elaborated diagrammatically. Section 3.3 comprises composition and formalization issues of network policies. These issues need to be resolved to increase network efficiency.

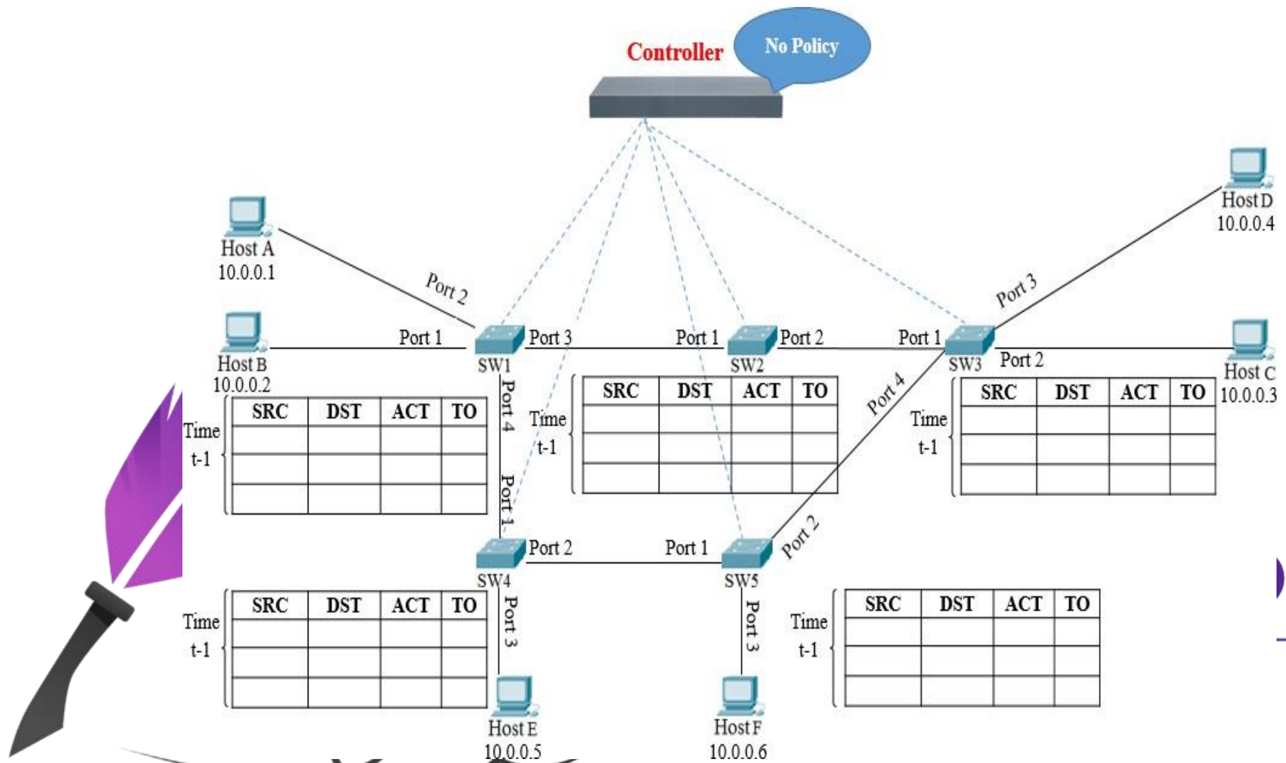
3.2 Network Policy Change at Controller and Flow Rules Installation

Suppose an enterprise network based on SDN architecture as shown in Figure 3.1. The network comprises five switches (SW1, SW2, SW3, SW4, SW5), six hosts (Host A, Host B, Host C, Host D, Host D, Host E) and these are connected via point-to-point links. The hosts are assigned IP address (10.0.0.0, 10.0.0.1, 10.0.0.2, 10.0.0.3, 10.0.0.4, 10.0.0.5). Each switch has flow table which comprises source IP address (SRC), destination IP address (DST), action (ACT) and timeout value (TO) for a certain flow rule. The network also includes a SDN controller which is responsible for installing flow rules at the switches based on the network policies. Initially all flow tables of all switches are empty as shown in Figure 3.1 (a). Assume that Host A wants to communicate with Host C. At a time t_1=0, there is no policy defined at the controller. In this case, Host A transmits the first packet (say pkt1) destined to Host C. After reaching first packet (pkt1) at Switch 1 (SW1), since there is no flow rule in the flow table of SW1 for Host C. Thus, pkt1 is stored at SW1 and a digest packet is forwarded to controller. When controller receives the digest packet, it installs rules for the communication from Host A to all switches (SW1, SW2 and SW3) along the path to Host C. Similarly, if Host A wants to communicate with Host D, and Host B to Host C, then controller installs the flow rules accordingly which is shown in Figure 3.1 (b). Then pkt1 from Host A destined for Host C is forwarded along the path as shown in Figure 3.1 (c). Suppose, Host A sends more packets (pkt2, pkt3) to Host C. All these subsequent packets follow the same path without intervening controller as shown in Figure 3.1 (d).

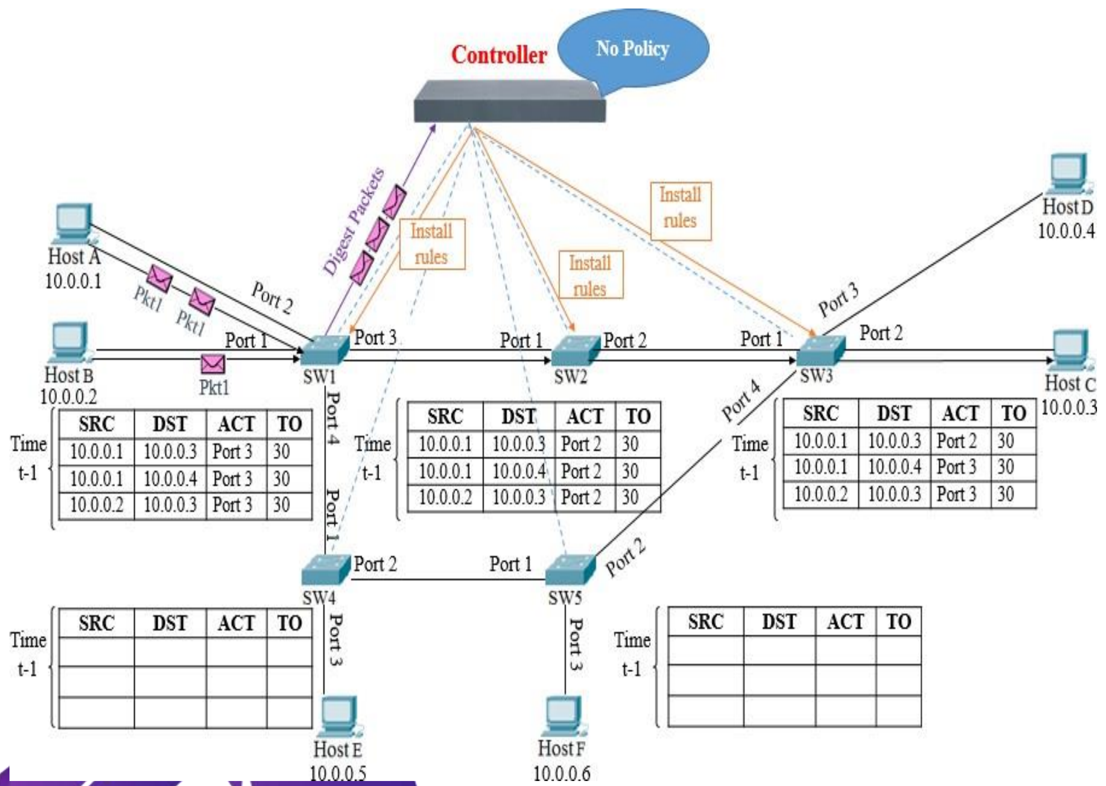
Suppose again that a new policy P_1 is introduced at the controller at a time t_2 (such that $t_2 > t_1$). Let suppose network policy P_1 defines that packet destined to Host C should

ACADEMIC SOLUTIONS

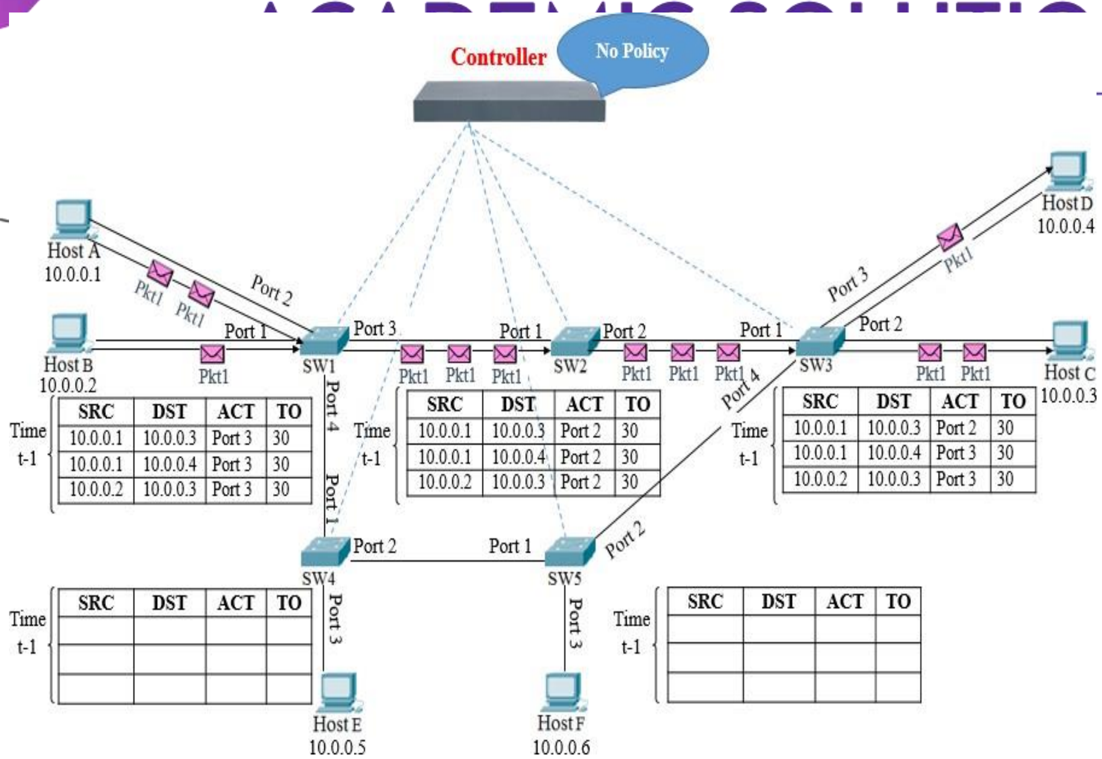
not pass through SW2. After this modification at the controller, Host A transmits more packets (pkt4, pkt5 and pkt6) destined to Host C. These packets (pkt4, pkt5 and pkt6) are forwarded on the existing path (Host A-SW1-SW2-SW3-Host C) due to already installed flow rules at switches along the path as shown in Figure 3.1 (e). This is a problem because these packets (pkt4, pkt5 and pkt6) violate the new policy P_1 (packets destined to Host C should not pass through SW2).



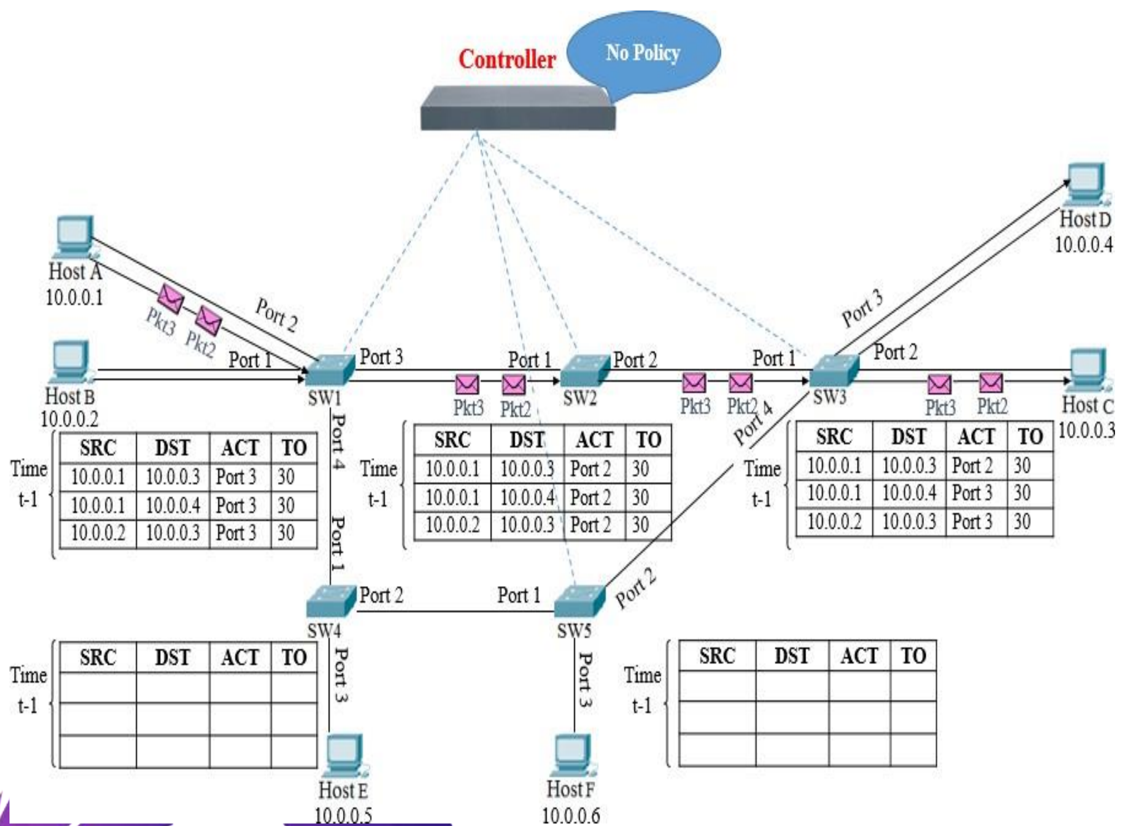
(a) OpenFlow Network with Empty Flow Tables of Switches at Time t_1



(b) OpenFlow Network with Flow Rules Installed in Flow Tables of Switches Along the Path (SW1-SW2-SW3) at Time t_1



(c) OpenFlow Network with Flow of Packets from Host A to Host C and Host D, and from Host B to Host D Along the Path (SW1-SW2-SW3) at Time t_1



SRC	DST	ACT	TO
10.0.0.1	10.0.0.3	Port 3	30
10.0.0.1	10.0.0.4	Port 3	30
10.0.0.2	10.0.0.3	Port 3	30

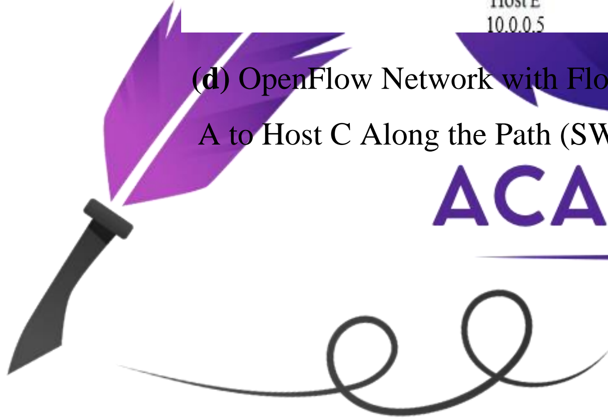
SRC	DST	ACT	TO
10.0.0.1	10.0.0.3	Port 2	30
10.0.0.1	10.0.0.4	Port 2	30
10.0.0.2	10.0.0.3	Port 2	30

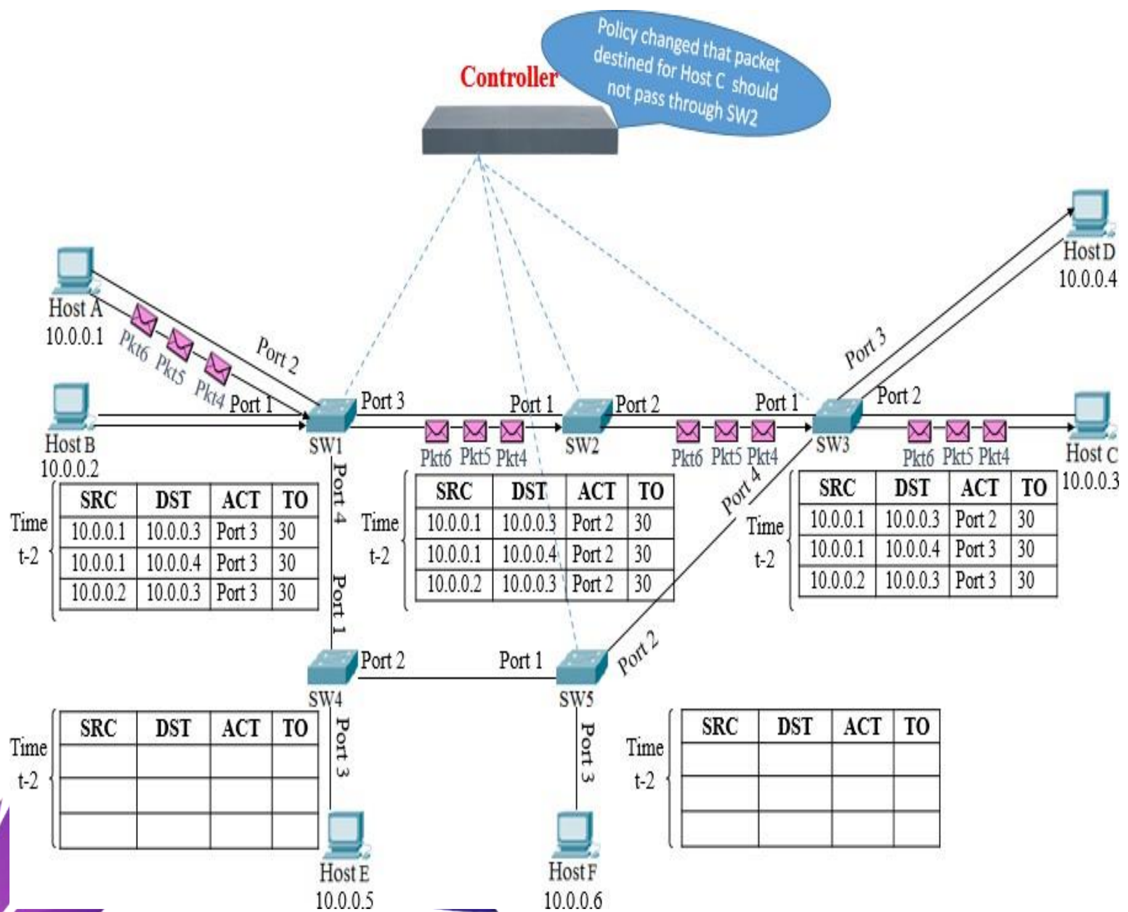
SRC	DST	ACT	TO
10.0.0.1	10.0.0.3	Port 2	30
10.0.0.1	10.0.0.4	Port 3	30
10.0.0.2	10.0.0.3	Port 3	30

SRC	DST	ACT	TO

SRC	DST	ACT	TO

(d) OpenFlow Network with Flow of Subsequent Packets (Pkt2 and Pkt3) from Host A to Host C Along the Path (SW1-SW2-SW3) without Intervention of Controller at Time t_1


ACADEMIC SOLUTIONS



(e) OpenFlow Network in Case of Policy Change at Controller but Packets Delivered to Invalid Interfaces due to Already Installed Flow Rules at Time t_2

ACADEMIC SOLUTIONS

Figure 3.1: OpenFlow Network Scenario of Network Policy Change

3.3 Composition and Formalization Issues of Network Policies

The network policies change with the passage of time due to change in application/user requirements or network topology. The network applications need to access data from various application platforms, servers, clouds etc. which may result in overlapping or conflicting access to network resources due to the implementation of network policies. To overcome these issues, the network policies need to be composed and formalized in a way that conflicts or overlapping issues of access to network resources should not occur. As in computer networks the network policies often change, so implementation of this change is very critical and can lead to network outages.

To resolve the above mentioned issues, our first proposed approach EPE [103] represents network policies via matrices and then compares these matrices to detect the change in network policies. However, this approach is inefficient with respect to access time, cost and space (this is shown by both computing the complexity in Chapter 4 and through simulation results in Chapter 5). It is due to that it requires all network policies to compare one by one in each matrix which takes longer time especially in a networking environment that experience frequent change in policies [206]. As a result, it delays flow rules computation at controller in addition to installation and deletion of these flow rules at data plane. Furthermore, because of this increased delay, the number of packets violating the network policies are also increased which leads towards network inefficiencies. In addition, in EPE [103] network abstractions are not assumed for implementation of network policies which is an important aspect especially in this information age. By introducing abstraction for the network policies, it enables network operators to easily specify, debug and manage network policies based on high level names instead of low-level commands. So, in order to address all these issues there is dire need of an efficient mechanism which can effectively resolve network policies conflicts and overlapping along with the composition issues of network policies. In addition, the proposed mechanism should be efficient in a way that network policy change should not affect network performance and efficiency.

ACADEMIC SOLUTIONS

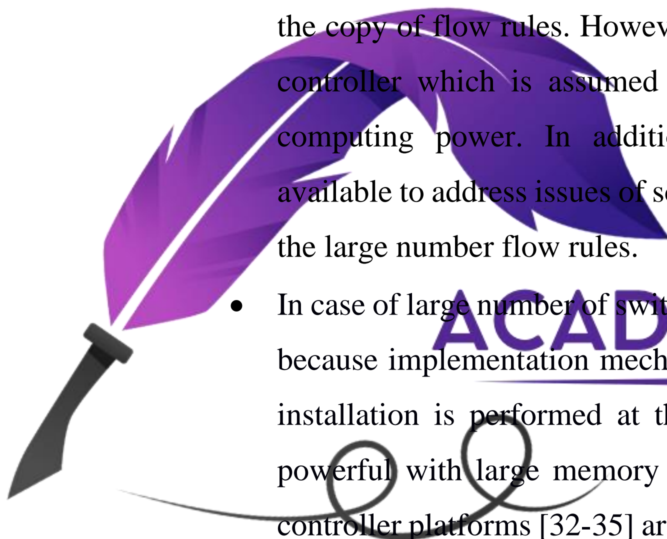
3.4 Discussion on Cost and Limitations of our Research Work

In this section, we have discussed the cost and limitations of our proposed research work as follows:

- In this research work, the focus is on reactive flow rules installation mechanism due to its wide implementation and use in real networks. This is due to that; we do not require prior knowledge of policy change. However, in proactive flow rules installation mechanism the prior knowledge of policy change is desired. Additionally, our proposed approaches can be more beneficial in proactive flow installation because a smaller number of packets are violated during the detection of policy change and the updation of flow rules. However, proactive flow rules installation may be costly and can create congestion in the network as well as at the controller in case of addition and deletion of large number of flow rules. This is because, in proactive flow rules installation, the controller needs to install flows for all nodes at data plane (i.e. at each switches/routers)

in advance, regardless the data is to be sent for every host on the network. This is very expensive in terms of memory usage at data plane, traffic overhead between controller and data plane, and processing overhead at controller. Thus, reactive flow rule installation is preferred as indicated in [37-38], especially in network environments where network policies change frequently.

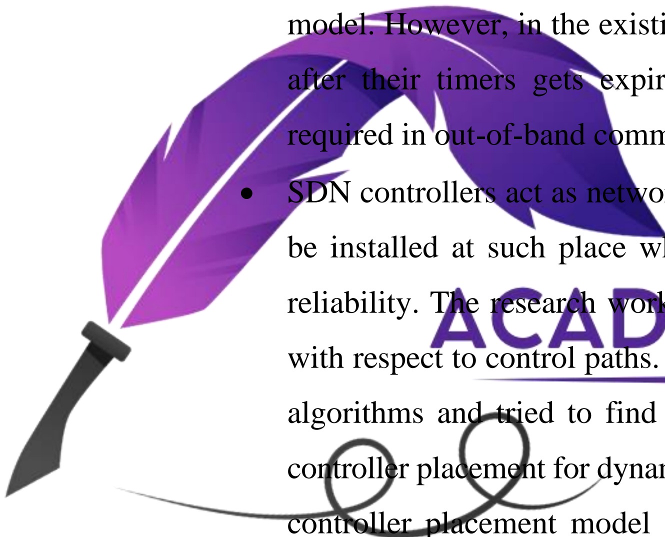
- In this research, hash table data structure is implemented at controller for caching a copy of flow rules (before installing at data plane). The advantage of this data structure is constant speed for lookup and storing flow rules in cache. This becomes more apparent when the number of entries is large. Hash tables are particularly efficient when the maximum number of entries can be predicted in advance, so that the bucket array can be allocated once with the optimum size and is never re-sized. Although, the cache is not infinite at controller to store the copy of flow rules. However, the proposed approaches cache flow rules at controller which is assumed to be powerful with large memory and high computing power. In addition, distributed controller platforms are also available to address issues of scalability which can have more memory to cache the large number flow rules.



- In case of large number of switches and hosts, the proposed solutions work well because implementation mechanism of policy change detection and flow rule installation is performed at the controller. The controller is assumed to be powerful with large memory and computing power. In addition, distributed controller platforms [32-35] are also available to address issues of scalability in which the workload is dynamically shifted to allow the controllers to operate at a load window that is specified in advance. In such systems, with the change of load over time, the pool of controllers expands or shrinks as per requirements.
- SDN has central point of failure problem due to centralized controller. However, the central point of failure in SDN can be handled via implementing distributed controllers, like [33-34]. These approaches deal with the issue of fault-tolerance as follows. If one controller fails, another standby controller can take over the responsibility. The proposed approaches of this work can adopt any mechanism of distributed controller platforms to deal with the central point of failure. Moreover, in [37] SDN multi-controller architectures along with their distribution method and the communication system are

discussed which are helpful in designing such kind of systems which deal with the issue of fault tolerance.

- The implementation of this research is based on out-of-band communication. After detecting the change in network policies and determining the already installed flow rules that need to be deleted, the controller sends the commands to data plane to delete these flow rules using out-of-band communication. Thus, the flow rules are deleted in less time and a smaller number of packets violate the policy in the proposed approaches. However, if in-band communication is used between the controller and data plane, then the proposed approaches will take longer time to remove the already installed flow rules. Thus, in this case the proposed approaches will have a greater number of packets violating the policy as compared to out-of-band communication model. However, in the existing approaches [52,58] the flow rules are purged after their timers gets expired. This duration is longer than the duration required in out-of-band communication to remove the already installed rules.
- SDN controllers act as network control points, so these control points need to be installed at such place which should increase network performance and reliability. The research works in [39-42] discussed reliability and resilience with respect to control paths. The researchers analyzed and compared various algorithms and tried to find the best one for the said problem. In [43] the controller placement for dynamic traffic flows is analyzed based on a combined controller placement model such as placement of controller platforms and communication between switch and controller. The proposed approach optimized the problem by minimizing average flow setup time under dynamic traffic conditions. The results suggest that average flow setup time for low flow densities could reduce up to 50% in case of static placement and at high densities flow setup performance is marginal.


ACADEMIC SOLUTIONS

Chapter 4

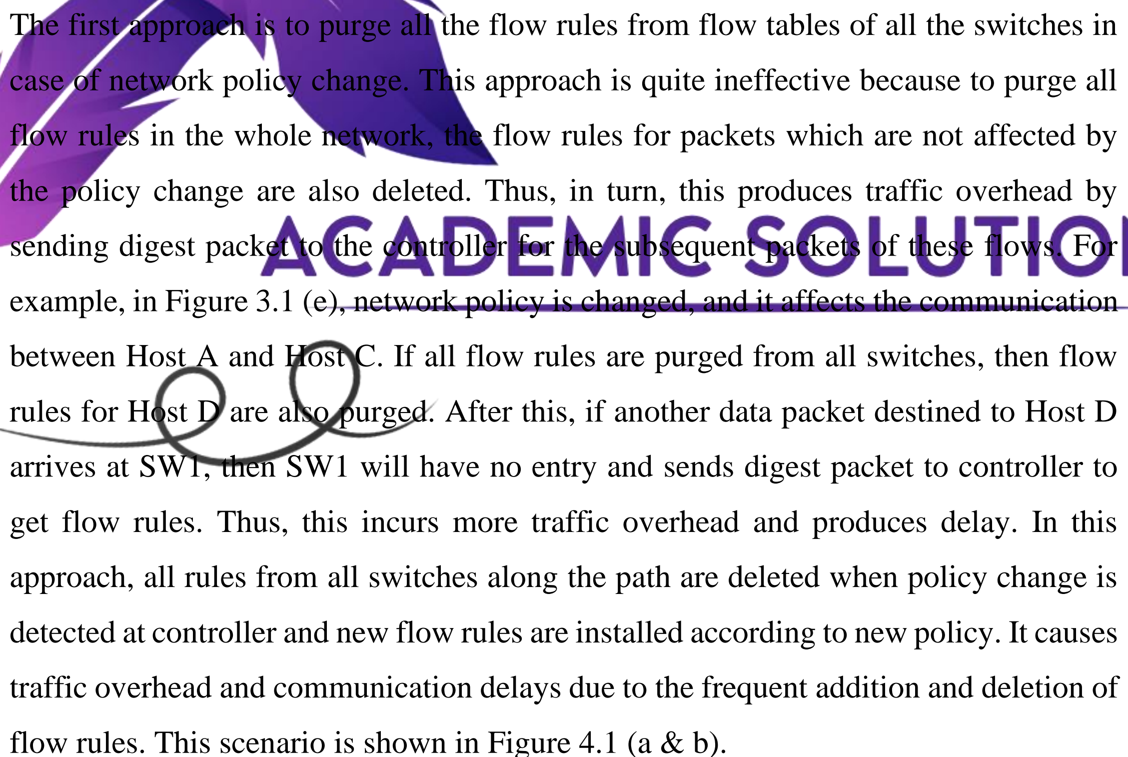
Proposed Solutions for Change Detection in Network Policies

4.1 Introduction

This chapter presents proposed solution of the identified research problem. In section 4.2, solution of purging all flow rules in case of policy change is discussed which is not an efficient solution. Section 4.3 includes deletion of only those flow rules which conflict with the changed policies. Section 4.4 comprises proposed solution based on matrix-based policy change detection and implementation. Section 4.5 includes graph-based policy change detection and implementation mechanism for the efficient handling of policy change at controller. Finally, on detection of policy change, flow rules are deleted from the switches based on old policies and new flow rules based on new policies are installed in addition to caching those flow rules at the controller.

4.2 Deletion of All Flow Rules

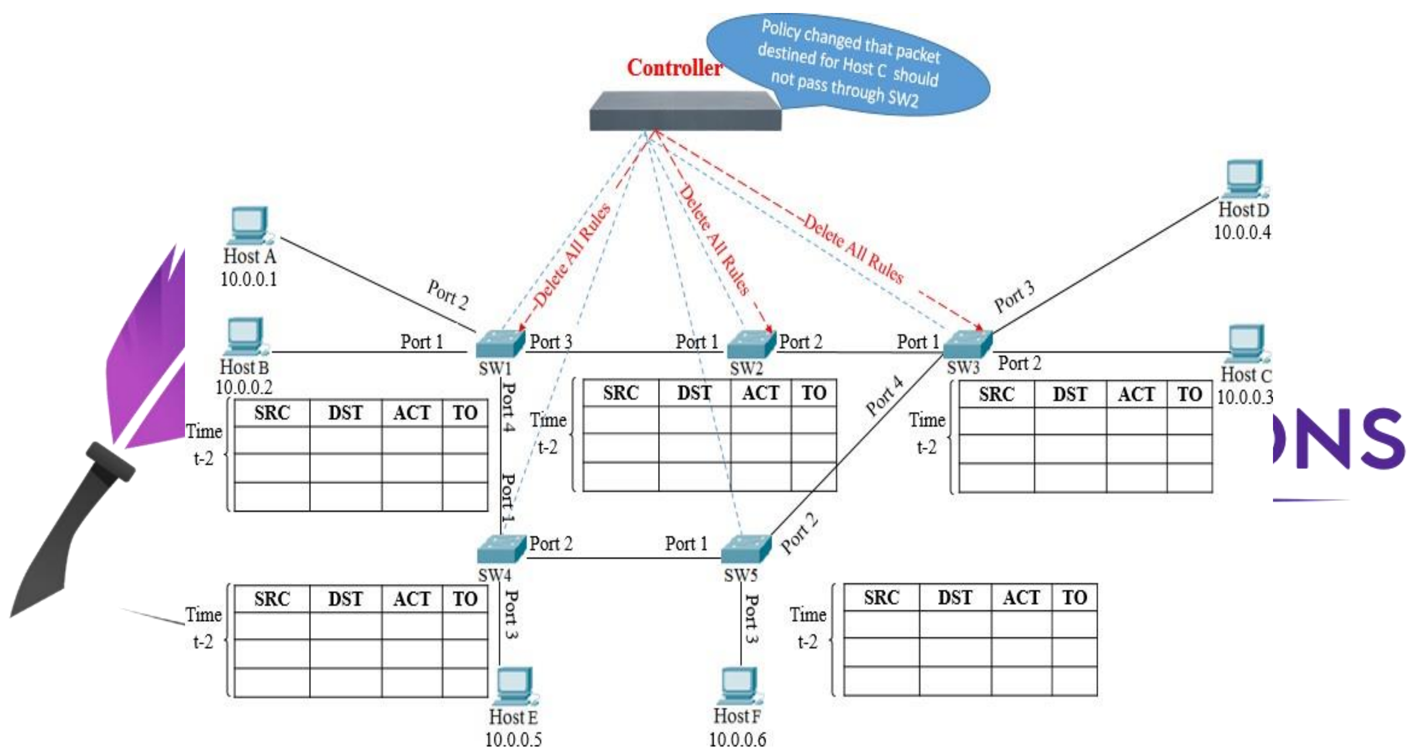
The first approach is to purge all the flow rules from flow tables of all the switches in case of network policy change. This approach is quite ineffective because to purge all flow rules in the whole network, the flow rules for packets which are not affected by the policy change are also deleted. Thus, in turn, this produces traffic overhead by sending digest packet to the controller for the subsequent packets of these flows. For example, in Figure 3.1 (e), network policy is changed, and it affects the communication between Host A and Host C. If all flow rules are purged from all switches, then flow rules for Host D are also purged. After this, if another data packet destined to Host D arrives at SW1, then SW1 will have no entry and sends digest packet to controller to get flow rules. Thus, this incurs more traffic overhead and produces delay. In this approach, all rules from all switches along the path are deleted when policy change is detected at controller and new flow rules are installed according to new policy. It causes traffic overhead and communication delays due to the frequent addition and deletion of flow rules. This scenario is shown in Figure 4.1 (a & b).



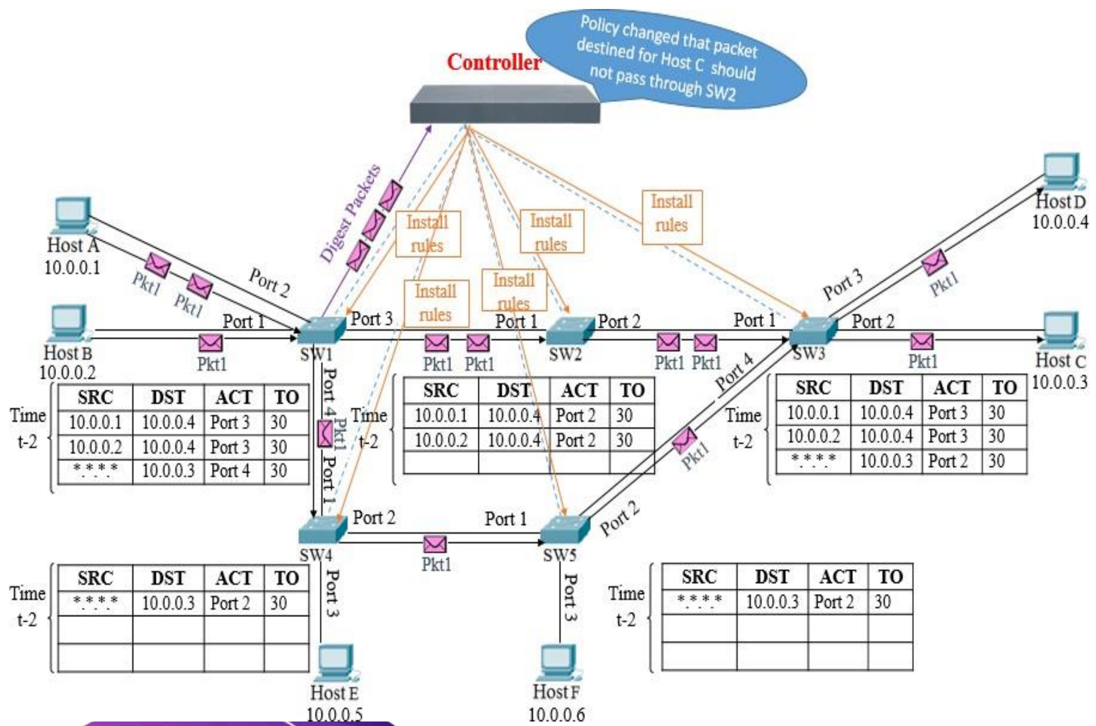
4.3 Deletion of Conflicting Flow Rules

In case of network policy change, the second and proposed approach of this work is to remove only those rules from the flow table of switches that are affected by the new policy P_1. For example, after deleting rules that are affected by P_1, the network configuration is shown in Figure 4.1(c). For this, it is proposed that controller caches a

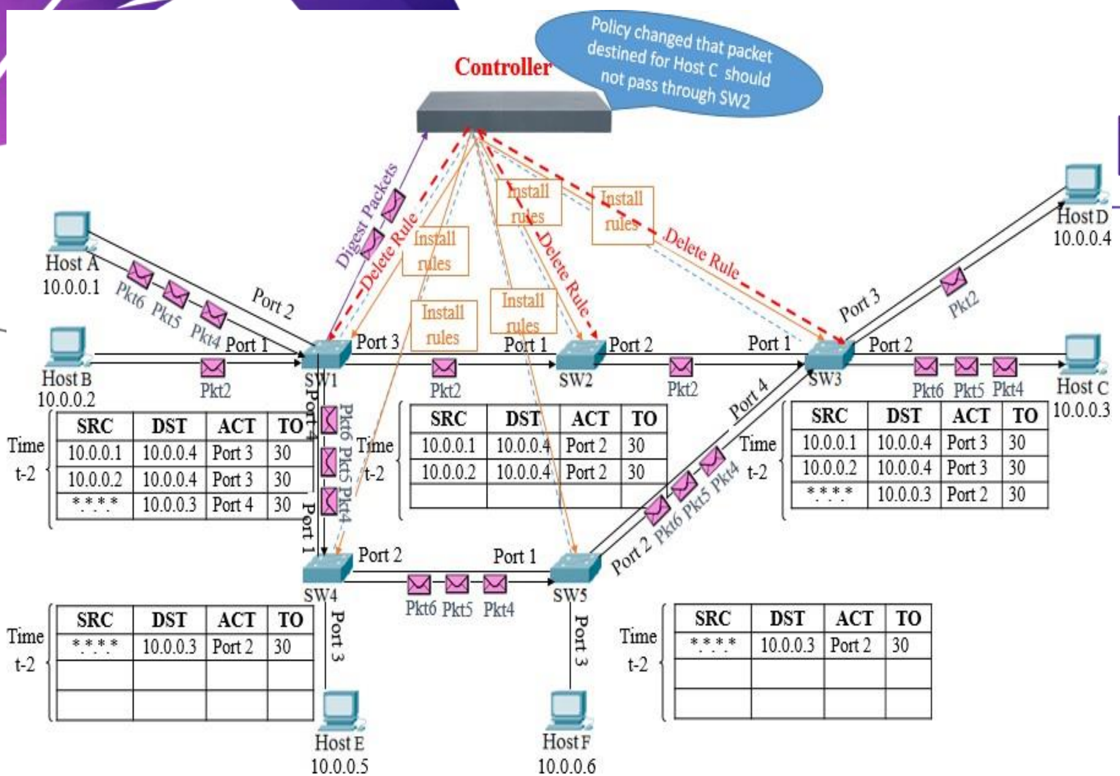
copy of rules before installing at switches. When controller detects policy change then it removes only those installed rules on switches which are affected by the new policy P_1 and installs new rules on the switches along the path based on new policy. This scenario shows that controller deletes only those flows from SW1, SW2 and SW3 which are affected by policy P_1. After this, when packet 4 is transmitted from Host A to Host C, SW1 will have no rules and thus forwards digest packet to controller. Then controller installs rules on SW1, SW3 and SW4 along new path according to new policy. Further, all packets from Host A to Host C will follow the same defined path without intervention of controller till the policy changes.



(a) OpenFlow Network Scenario in case of Policy Change at Controller and All Flow Rules are Deleted



(b) OpenFlow Network Scenario in case of Policy Change on Controller and New Flow Rules Installed on Switches along the Path



(c) OpenFlow Network Scenario in case of Policy Change at Controller and only Affected Flow Rules are Deleted and Installed

Figure 4.1: OpenFlow Network Scenario in Case of Policy Change and Flow Rules Installation

4.4 Matrix-Based Policy Change Detection and Flow Rules Installation

In this research work, matrix-based policy change detection and flow rule installation approach called "Efficient Policy Enforcement" (EPE) is proposed which caches flow rules at controller before installing them at data plane (i.e. forwarding devices). When a policy is updated at the controller, it is triggered to check that whether this new policy affects the rules already installed at the switches/routers. If so, then those flow rules are purged from the corresponding forwarding devices. For proposed scenario, it is assumed that network policies are defined only based on destination IP address. Flow rules generated by the controller may consist of many fields. For simplicity, flow rules are represented in 5 Tuple <Source IP, Destination IP, Action, Timeout, SwitchID>. Source IP means the IP address of Source Host, Destination IP means IP address of Destination Host, Action means that what action (forward to the specified port or drop) is applied to the destination Host, Timeout means after how many seconds flow rule will be removed from the flow table and SwitchID means which switch has respective flow rule. The proposed approach has following components.

4.4.1 Policy Representation

ACADEMIC SOLUTIONS

In this research, pyretic language [11] is used to define network policies via high-level abstractions rather than low-level open flow mechanisms. Pyretic helps to implement physical rules on multiple switches via a pyretic policy based on abstract functions that takes packet as input and returns a set of new packets. This return function depends upon the below given functions:

- Return empty packet based on the drop
- Return single packet based on forwarding to new single location
- Return multiple packets based on multicasting, etc. [156]

In current presented scenario, flow rules are generated by the controller in 5 tuple <Source IP, Destination IP, Action, Timeout, SwitchID> and installed at all switches along the path. Based on policy representation file, after traversing a policy, say (match (switch = 'switchid') & match (dstip = 'ip') >> policy, search for "switchid" in SWITCH matrix and there can be following two cases:

i. If the matching does not exist, then "switchid" is appended in a new row in SWITCH, say "rowi", and "ip" is appended in "rowi" in DESTINATION_IP matrix and policy is appended in "rowi" in FORWARDING_POLICY matrix

ii. Otherwise, "switchid" exists in SWITCH at a row, say "rowj".

Then "ip" is appended in a new column at "row_j" in DESTINATION_IP matrix and policy is added in a new column at "row_j" in FORWARDING_POLICY matrix.

```
(match(switch=SW1) & match(dstip='10.0.0.3')>>fwd(2))
(match(switch=SW2) & match(dstip='10.0.0.3')>>fwd(3))
(match(switch=SW3) & match(dstip='10.0.0.3')>>fwd(2))
(match(switch=SW1) & match(dstip='10.4.1.1')>>drop)
(match(switch=SW1) & match(dstip='10.4.1.1')>>fwd(3))
(match(switch=SW4) & match(dstip='*.*.*.*')>>fwd(3))
(match(switch=SW5) & match(dstip='11.5.1.2')>>fwd(1))
```

Figure 4.2: Policy Representation File at Time t_1

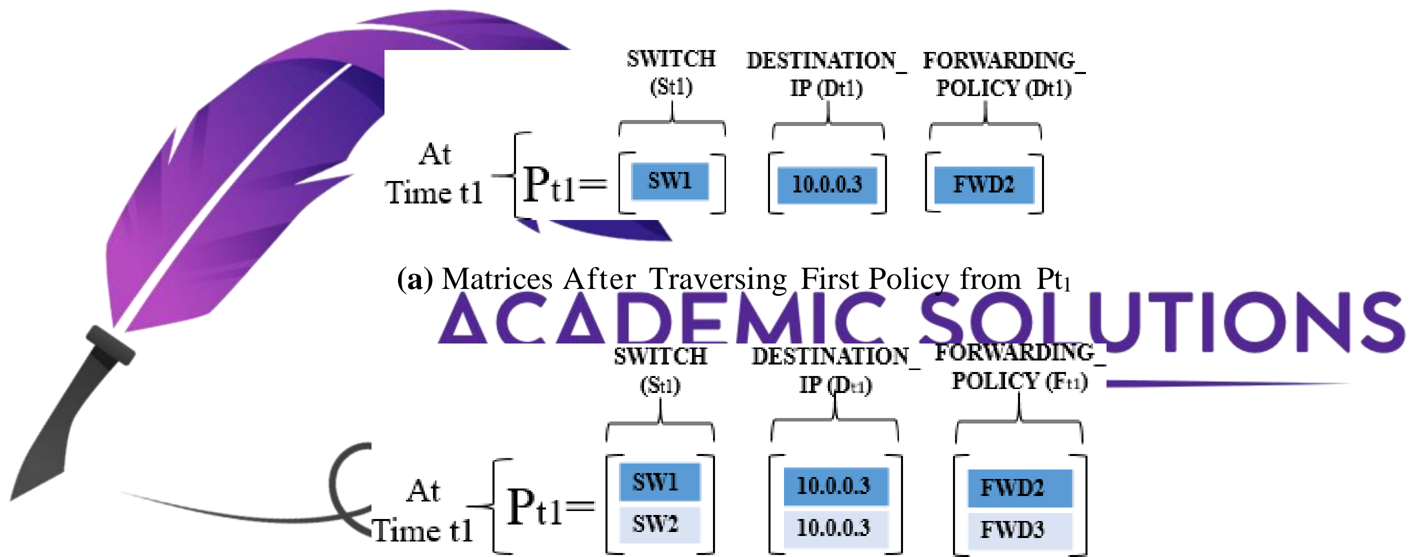
```
(match(switch=SW1) & match(dstip='10.0.0.3')>>fwd(2))
(match(switch=SW2) & match(dstip='10.0.0.3')>>drop)
(match(switch=SW3) & match(dstip='10.0.0.3')>>fwd(2))
(match(switch=SW1) & match(dstip='10.4.1.1')>>fwd(3))
(match(switch=SW4) & match(dstip='*.*.*.*')>>fwd(3))
(match(switch=SW4) & match(dstip='10.5.1.1')>>fwd(2))
```

Figure 4.3: Policy Representation File at Time t_2

For example, after traversing first network policy from the policy representation file at time instance t_1, as shown in Figure 4.2, say, (match (switch = SW1) & match (dstip = '10.0.0.3') >> fwd (2)), the matrices are populated and shown in Figure 4.4(a). After traversing next policy in Figure 4.2, say (match (switch = SW2) & match (dstip = '10.0.0.3') >> fwd (3)), it is found that SW2 does not exist in SWITCH matrix, so SW2 is appended in new row, i.e. row_2, in SWITCH matrix and destination ip is appended in

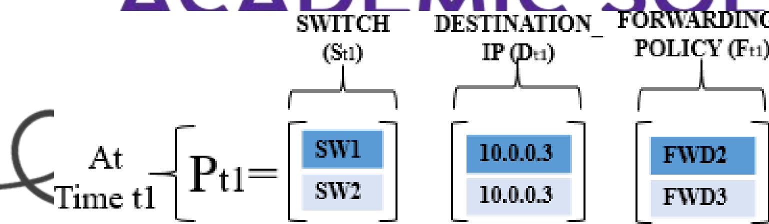
DESTINATION_IP at "row2". In addition, respective policy, fwd (3), is appended in "row2" of FORWORDING_POLICY matrix. In this case, matrices are shown in Figure 4.4(b). Similarly, after traversing next network policies in Figure 4.2, matrices are populated which are shown in Figure 4.4(c–g).

Suppose at time instance t_2 (such that $t_2 > t_1$) policy pt_2 is implemented which states that packets destined to Host "10.0.0.3" should not pass through Switch "SW2". This policy is represented in pyretic language as: (match (switch = SW2) & match (dstip = '10.0.0.3') >> drop). At time t_2, the set of network policies are added in policy representation file as shown in Figure 4.3. Then, the proposed approach traverses the policy representation file, the three matrices are populated according to policies which are shown in Figure 4.4(h).

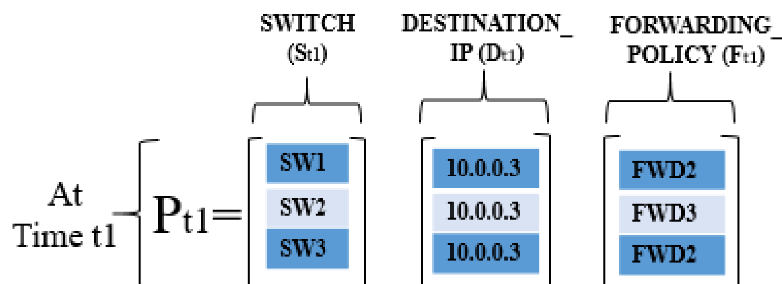


(a) Matrices After Traversing First Policy from Pt_1

ACADEMIC SOLUTIONS



(b) Matrices After Traversing Second Policy from Pt_1



(c) Matrices After Traversing Third Policy from Pt_1

At Time t1 P_{t1} =

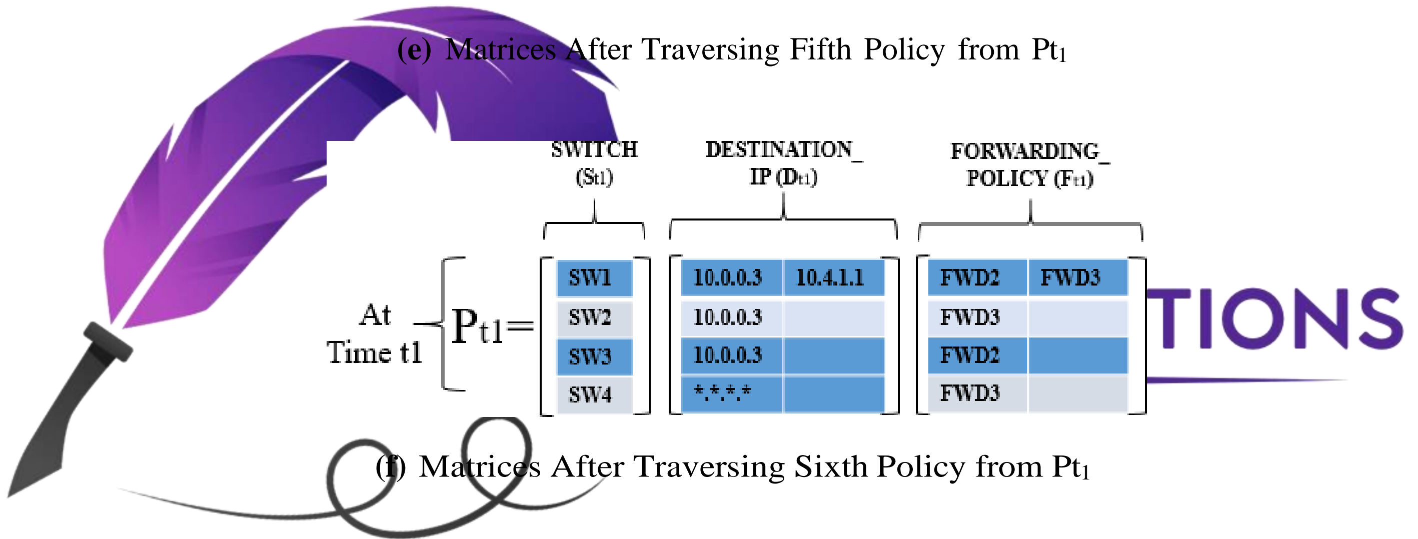
SWITCH (S_{t1})	DESTINATION_IP (D_{t1})		FORWARDING_POLICY (F_{t1})	
SW1	10.0.0.3	10.4.1.1	FWD2	DROP
SW2	10.0.0.3		FWD3	
SW3	10.0.0.3		FWD2	

(d) Matrices After Traversing Fourth Policy from Pt_1

At Time t1 P_{t1} =

SWITCH (S_{t1})	DESTINATION_IP (D_{t1})		FORWARDING_POLICY (F_{t1})	
SW1	10.0.0.3	10.4.1.1	FWD2	FWD3
SW2	10.0.0.3		FWD3	
SW3	10.0.0.3		FWD2	

(e) Matrices After Traversing Fifth Policy from Pt_1

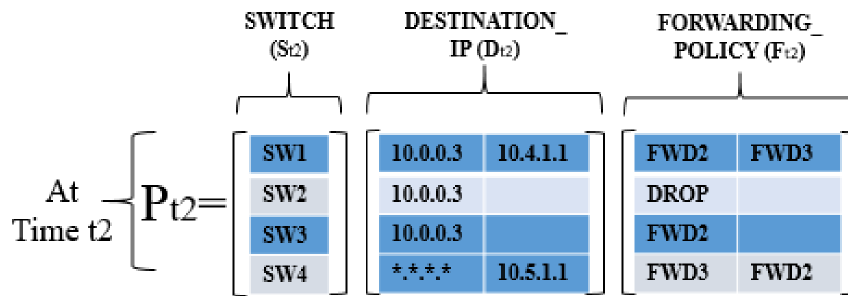


(f) Matrices After Traversing Sixth Policy from Pt_1

At Time t1 P_{t1} =

SWITCH (S_{t1})	DESTINATION_IP (D_{t1})		FORWARDING_POLICY (F_{t1})	
SW1	10.0.0.3	10.4.1.1	FWD2	FWD3
SW2	10.0.0.3		FWD3	
SW3	10.0.0.3		FWD2	
SW4	*.*.*.*		FWD3	
SW5	11.5.1.2		FWD1	

(g) Matrices After Traversing Seventh Policy from Pt_1



(h) Matrices After Traversing All Policies from Pt_2

Figure 4.4: Matrices for Detection of Policy Change

4.4.2 Matrix-Based Network Policy Change Detection

In order to detect network policy change, two matrices Pt_1 and Pt_2 are compared as shown in Figure 4.4 (g,h). More specifically, it is done as follows:

The network policy change is detected via "Algorithm 4.1". In this algorithm, we have taken two set of policy matrices, Pt_1 and Pt_2 at times t_1 and t_2 respectively. These sets of matrices consist of SWITCH, DESTINATION_IP and FORWARDING_POLICY. At first, 'switch id' is selected from the SWITCH matrix of Pt_2 and respective entry is checked from Pt_1. If respective entry is found, then both Pt_1 and Pt_2 are compared. If both are equal, then the result is "Matrices are Matched". After that, respective entry is removed from the matrix Pt_1. Otherwise, the result is "Matrices are Not Matched" and entry is noted and removed from the matrix Pt_1. On the other hand, if there is no entry found then result is "Matrices are Not Matched" and respective entry is noted. This process is repeated until all matrices are traversed. Finally, entries are searched from Pt_1 and noted in a new matrix. If the resultant Pt_1 is equal to Pt_2 then it means that policy is not changed otherwise policy is changed.

ACADEMIC SOLUTIONS

Algorithm 4.1: Matrix-Based Policy Change Detection

Input: Pt_1 and Pt_2 are set of matrices at time t_1 and t_2 respectively, say,

SWITCH = (SW_1, SW_2, … ,SW_n), DESTINATION_IP = (IP_1, IP_2, … , IP_n),

FORWARDING_POLICY = (R_1, R_2, … , R_n)

Output: Pt_2 = Pt_1 (Network Policy is Not Changed) OR Pt_2 != Pt_1 (Network Policy Changed)

1: **Bool** Status = False

2: Matrix_Index = 0

3: Matrix_Hashable_Struct = Dictionary ()

4: Function Matrix_Log (*matlog)

5: **For Each** Matrix in matlog **Do**

6: Matrix_Index = Matrix_Index + 1

7: Matix_Hashable_Struct [Matrix_Index] = Matrix

8: Function Delete (*matvalue)

9: **End For Loop**

10: **For Each** Mat_Value in matvalue **Do**

11: Mat_Value.Delete ()

12: **End For Loop**

13: **For Each** Mat_Index_2 **From** Pt_2.SWITCH **Do**

14: **For Each** Mat_Index_1 **From** Pt_1.SWITCH **Do**

15: **If** (Pt_1.SWITCH[Mat_Index_2][Mat_Index_1] = Pt_2.SWITCH[Mat_Index_2][Mat_Index_1]) **Then**

16: *Status = True*

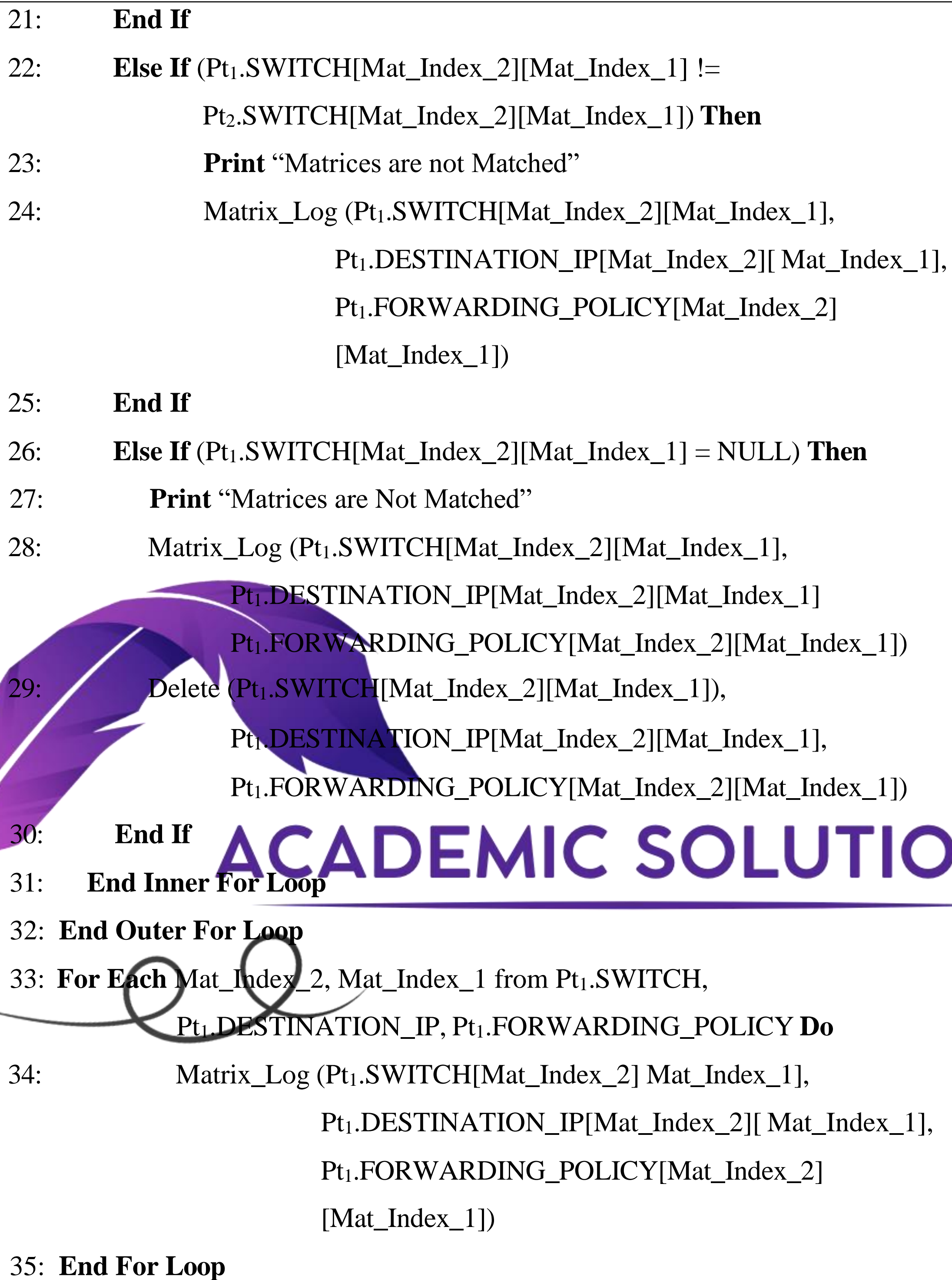
17: **If** ((Pt_1.DESTINATION_IP[Mat_Index_2][Mat_Index_1] = Pt_2.DESTINATION_IP[Mat_Index_2][Mat_Index_1]) & (Pt_1.FORWARDING_POLICY[Mat_Index_2][Mat_Index_1] = Pt_2. FORWARDING_POLICY[Mat_Index_2][Mat_Index_1]))

18: **Print** "Matrices are Matched"

19: Delete (Pt_1.SWITCH[Mat_Index_2] [Mat_Index_1], Pt_1.DESTINATION_IP[Mat_Index_2] [Mat_Index_1], Pt_1. FORWARDING_POLICY [Mat_Index_2] [Mat_Index_1])

20: **End If**

ACADEMIC SOLUTIONS


ACADEMIC SOLUTIONS

```
21:         End If
22:         Else If (Pt1.SWITCH[Mat_Index_2][Mat_Index_1] !=
                     Pt2.SWITCH[Mat_Index_2][Mat_Index_1]) Then
23:                  Print "Matrices are not Matched"
24:                  Matrix_Log (Pt1.SWITCH[Mat_Index_2][Mat_Index_1],
                                   Pt1.DESTINATION_IP[Mat_Index_2][ Mat_Index_1],
                                   Pt1.FORWARDING_POLICY[Mat_Index_2]
                                   [Mat_Index_1])
25:         End If
26:         Else If (Pt1.SWITCH[Mat_Index_2][Mat_Index_1] = NULL) Then
27:             Print "Matrices are Not Matched"
28:             Matrix_Log (Pt1.SWITCH[Mat_Index_2][Mat_Index_1],
                        Pt1.DESTINATION_IP[Mat_Index_2][Mat_Index_1]
                        Pt1.FORWARDING_POLICY[Mat_Index_2][Mat_Index_1])
29:             Delete (Pt1.SWITCH[Mat_Index_2][Mat_Index_1]),
                        Pt1.DESTINATION_IP[Mat_Index_2][Mat_Index_1],
                        Pt1.FORWARDING_POLICY[Mat_Index_2][Mat_Index_1])
30:         End If
31:     End Inner For Loop
32:  End Outer For Loop
33:  For Each Mat_Index_2, Mat_Index_1 from Pt1.SWITCH,
               Pt1.DESTINATION_IP, Pt1.FORWARDING_POLICY Do
34:               Matrix_Log (Pt1.SWITCH[Mat_Index_2] Mat_Index_1],
                                 Pt1.DESTINATION_IP[Mat_Index_2][ Mat_Index_1],
                                 Pt1.FORWARDING_POLICY[Mat_Index_2]
                                 [Mat_Index_1])
35:  End For Loop
```

4.4.3 Flow Rules Caching

The flow rules generated by controller are cached in a hash table [207] data structure. In this proposed approach, Hash Function maps each destination IP address to a unique bucket/slot in hash table and rules are stored at that bucket/slot. The Hash Table ranges is set from 0 to 500 slots due to the limited number of Hosts/Switches which is kind of

static table as the number of Hosts/Switches are known. In addition, Consistent Hash Function [208] is also used due to the fact that every time when function is executed it produces same output for an input, therefore, it helps to accurately find a certain flow rule on a specific switch. The flow rules are cached in buckets/slots in a linked list data structure which consists of 5-tuple form [source address, destination address, Timeout, action, SwitchID]. When a policy change is detected at controller via Algorithm 4.1, then based on the value of key (destination IP address), Hash Function is applied which points to the specific slot/bucket. Based on the result of Hash Function, the flow rules are deleted from switches via Modify-State message [209] and new flow rules are computed and added in the bucket/slot accordingly.

For example, suppose Hash Function takes destination IP address as its parameter and makes the sum of its decimal numbers. After that, the it takes the modulus of that number by the table length to get an index number. The table length is assumed to 500. Figure 4.5 shows Hash Table based on the destination IP addresses for SW_1, SW_2, SW_3 and SW_4. Similarly, for other keys, flow rules are also added at their respective locations. In this way, flow rules are cached at controller and in case of policy change, these are deleted/added from the flow tables of respective switches and controller cache.

ACADEMIC SOLUTIONS

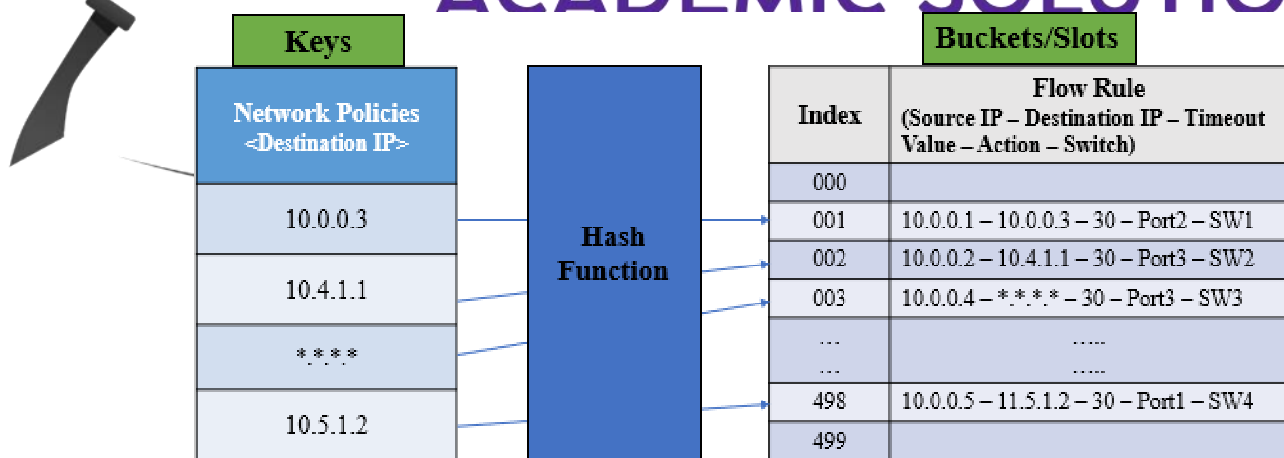


Figure 4.5: Hash Table Implementation in Case of Matrix-Based Approach

4.4.4 Complexity of Algorithm 4.1

The Matrix-Based Policy Change Detection Algorithm "Algorithm 4.1" is analyzed based on number of times instructions are executed. It consists of loops and statements which check the detection of policy change via matrices. The matrices are "SWITCH",

"DESTINATION_IP" and "FORWARDING_POLICY". Suppose "n" represents the number of times a statement is executed, then the execution frequency of each statement is given in Table 4.1.

Table 4.1: Complexity of Matrix-Based Policy Change Detection

Algorithm 4.1	**No. of Times Instructions Executed**
Input: Pt_1 and Pt_2 are set of matrices at time t_1 and t_2 respectively, say, SWITCH = (SW_1, SW_2, … ,SW_n), DESTINATION_IP = (IP_1, IP_2, … , IP_n), FORWARDING_POLICY = (R_1, R_2, … , R_n)	
Output: $Pt_2 = Pt_1$ (Network Policy is Not Changed) OR $Pt_2 != Pt_1$ (Network Policy is Changed)	
1: **Bool** Status = False	1
2: Matrix_Index = 0	1
3: Matrix_Hashable_Struct = Dictionary ()	1
4: Function Matrix_Log (*matlog)	1
5: **For Each** Matrix in mat **Do**	n+2
6: Matrix_Index = Matrix_Index + 1	n+1
7: Matix_Hashable_Struct [Matrix_Index] = Matrix	n+1
8: Function Delete (*matvalue)	n+1
9: **End For Loop**	
10: **For Each** Mat_Value in matvalue **Do**	n+2
11: Mat_Value.Delete ()	n+1
12: **End For Loop**	
13: **For Each** Mat_Index_2 **From** Pt_2.SWITCH **Do**	n+2
14: **For Each** Mat_Index_1 **From** Pt_1.SWITCH **Do**	(n+1)(n+2)
15: **If** (Pt_1.SWITCH[Mat_Index_2] [Mat_Index_1] = Pt_2.SWITCH[Mat_Index_2][Mat_Index_1]) **Then**	(n+1)(n+1)
16: *Status = True*	(n+1)(n+1)
17: **If** ((Pt_1.DESTINATION_IP[Mat_Index_2][Mat_Index_1] = Pt_2.DESTINATION_IP[Mat_Index_2][Mat_Index_1]) & Pt_1.FORWARDING_POLICY[Mat_Index_2][Mat_Index_1]	(n+1)(n+1)

ACADEMIC SOLUTIONS

Pt_2.FORWARDING_POLICY[Mat_Index_2][Mat_Index_1]))	
18: **Print** "Matrices are Matched"	(n+1)(n+1)
19: Delete (Pt_1.SWITCH[Mat_Index_2] [Mat_Index_1], Pt_1.DESTINATION_IP[Mat_Index_2] [Mat_Index_1], Pt_1. FORWARDING_POLICY[Mat_Index_2][Mat_Index_1])	(n+1)(n+1)
20: **End If**	
21: **End If**	
22: **Else If** (Pt_1.SWITCH[Mat_Index_2][Mat_Index_1] != Pt_2.SWITCH[Mat_Index_2][Mat_Index_1]) **Then**	(n+1)(n+1)
23: **Print** "Matrices are not Matched"	(n+1)(n+1)
24: Matrix_Log(Pt_1.SWITCH[Mat_Index_2][Mat_Index_1], Pt_1.DESTINATION_IP[Mat_Index_2][Mat_Index_1], Pt_1.FORWARDING_POLICY[Mat_Index_2][Mat_Index_1])	(n+1)(n+1)
25: **End If**	
26: **Else If** (Pt_1.SWITCH[Mat_Index_2][Mat_Index_1] = NULL) **Then**	(n+1)(n+1)
27: **Print** "Matrices are Not Matched"	(n+1)(n+1)
28: Matrix_Log (Pt_1.SWITCH[Mat_Index_2][Mat_Index_1], Pt_1.DESTINATION_IP[Mat_Index_2][Mat_Index_1] Pt_1.FORWARDING_POLICY[Mat_Index_2][Mat_Index_1])	(n+1)(n+1)
29: Delete(Pt_1.SWITCH[Mat_Index_2][Mat_Index_1]), Pt_1.DESTINATION_IP[Mat_Index_2][Mat_Index_1], Pt_1.FORWARDING_POLICY[Mat_Index_2][Mat_Index_1])	(n+1)(n+1)
30: **End If**	
31: **End Inner For Loop**	
32: **End Outer For Loop**	
33: **For Each** Mat_Index_2, Mat_Index_1 from Pt_1.SWITCH, Pt_1.DESTINATION_IP, Pt_1.FORWARDING_POLICY **Do**	n+2
34: Matrix_Log(Pt_1.SWITCH[Mat_Index_2] Mat_Index_1], Pt_1.DESTINATION_IP[Mat_Index_2][Mat_Index_1], Pt_1.FORWARDING_POLICY[Mat_Index_2] [Mat_Index_1])	n+1
35: **End For Loop**	
Total Complexity = $13n^2 + 36n + 31$ **Worst Case Complexity** = $O(n^2)$	

ACADEMIC SOLUTIONS

4.5 Graph-Based Policy Change Detection and Flow Rules Installation

In this research work, the network policies are represented in 6 tuple <Source, Destination, Protocol, Ports, Service Function Chain, Action>. These policies are traversed into the policy implementation file, which is then passed to the controller. The controller computes a multi-attributed graph, G = (V, E, α, β).

Here:

- V is finite set of vertices which consists of three types, that is, Source, Destination and Service Function Chain Sequence.
- An edge in G is represented by E which consists of a communication link (v_1, v_2) between two vertices ($v_1, v_2 \in V$).
- α consists of two types of attributes of vertices, namely, Endpoint Group (E_g) and Service Function Chain sequence (S).

  Here, $\alpha = E_g \cup S$, where E_g denotes an Endpoint Group (EPG) and S denotes service function chain sequence. An EPG comprises collection of Endpoints (EPs). An EP is a minimum abstraction unit for which a policy is implemented, for example, server, subnet, network, or end-user. An EPG consists of all EPs which satisfies all conditions of membership predicate. Each membership predicate is assigned a label, e.g., FCL, STF, LMS, WBS as shown in Figure 4.9. The membership predicate is the Boolean Expression over all labels. In addition, there is S sequences for the network function boxes, for example, FW and LB to handle network communication. The composed graph shows which communication is allowed between network endpoints and what S sequence is required for that communication.
- β consists of two attributes of an edge, namely, $\beta_{Protocol}$ and β_{Ports}. The $\beta_{Protocol}$ represents the attribute of an edge that comprises values of TCP or UDP or its value can be ANY (which consists of both TCP and UDP traffic). The β_{Ports} represents another type of edge attribute that comprises a set of destination port numbers of applications, that is, β_{Ports} represents set of integer values or ANY for a policy.

The system model of this research work is whitelisting, it means that by default no communication is allowed between the EPs. Moreover, in this work, reactive mode of

ACADEMIC SOLUTIONS

flow installation mechanism is assumed in which the controller computes flow rules in on-demand fashion. More specifically, when a flow is generated, the controller computes the flow rules for that flow by using the network topology and network policies. It then installs flow rules at the switches along the shortest computed path and caches a copy of flow rules in a Hash Table data structure. Suppose, later on, policy change is triggered, then the controller computes the multi-attributed graph and policy change is detected via our proposed graph matching algorithm (that is Algorithm 4.2). If there is no change, then no action is required, it means that network is running as per up-to-date policies. Otherwise, the proposed approach finds out that whether changed policies are conflicting with the flow rules cached at the controller or not. If matching is found, then these are called conflicting flow rules. Otherwise, the flow rules are called non-conflicting. In this case, we can say that new policy is added for the Source and Destination other than the existing ones. Thus, if there is no conflict, it means that new policy is added for hosts other than the existing ones. Otherwise, the proposed approach deletes all those flow rules from the flow tables of the switches along the shortest path and from the controller cache that are conflicting with the changed policies. Finally, new flow rules and shortest path are computed as per new policy and these newly computed flow rules are installed along the shortest path at switches and cached at controller for further processing.

ACADEMIC SOLUTIONS

Let us discuss the proposed solution stepwise which is represented in Figure 4.9. In first step, we list down the network policies as per our network scenario in a policy file, say at time t_1 and pass the policy file to the controller. We represent these policies via a multi-attributed graph (say G_1), where, $G_1 = (V_1, E_1, \alpha_1, \beta_1)$.

Here:

- V_1 is finite set of vertices at time t_1, such that, $V_1 \subseteq V$. These vertices consist of $Source_1$, $Destination_1$ and S_1 sequence.
- An Edge in G_1 is represented by E_1 which consists of a communication link $(v1_1, v2_1)$ between two vertices $(v1_1, v2_1 \in V_1)$.
- α_1 consists of two types of attributes, namely, EndPoint Group (E_{g_1}) and S_1 sequence. Here, $\alpha_1 = E_{g_1} \cup S_1$, where E_{g1} denotes an EPG_1 and S_1 denotes service function chain sequence to handle network traffic by network function boxes (FW and LB). The composed graph G_1 at time t_1 shows that which

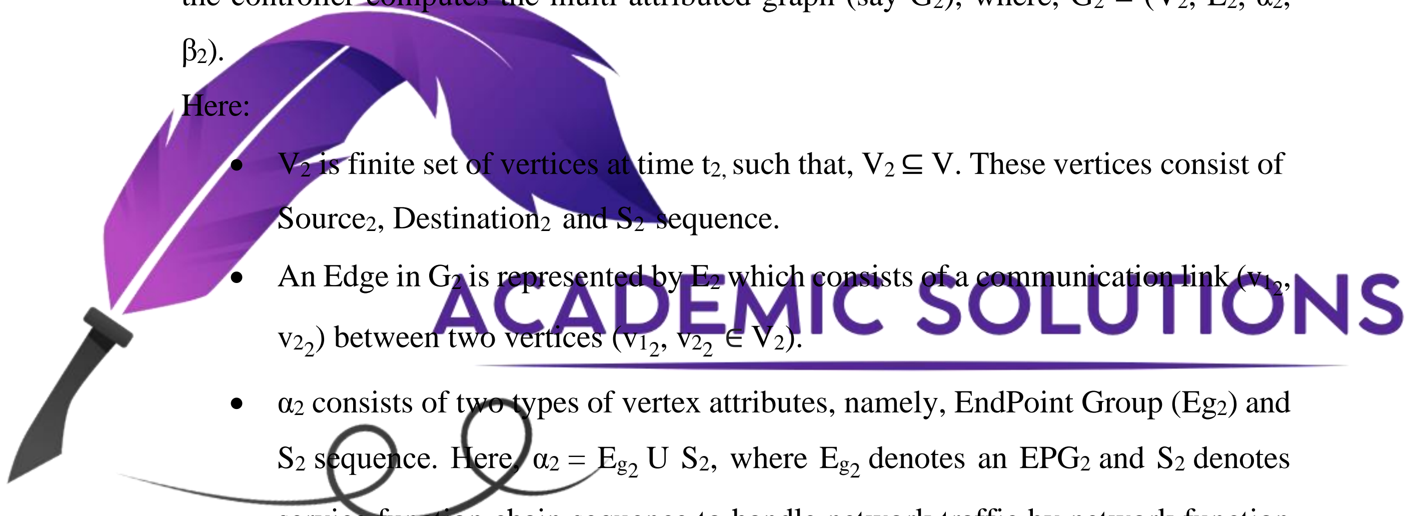
communication is allowed between network EP_1 and what S_1 sequence is required for the communication between Source and Destination.

- β_1 consists of two attributes of an edge, namely, $\beta_{Protocol_1}$ and β_{Ports_1}. The $\beta_{Protocol_1}$ represents the attribute of an edge that comprises values of TCP or UDP or ANY, where ANY represents both TCP and UDP traffic. Moreover, the β_{Ports_1} represents another type of attribute of the edge that comprises a set of destination port numbers of applications. Here, β_{Ports_1} consists of a set of integer values or ANY for a policy.

In second step, it installs flow rules in flow tables of the switches along the shortest computed path. In third step, it caches a copy of flow rules in a hash table data structure at controller. In fourth step, suppose, later, say at time t_2, policy change triggers, then the controller computes the multi-attributed graph (say G_2), where, $G_2 = (V_2, E_2, \alpha_2, \beta_2)$.

Here:

- V_2 is finite set of vertices at time t_2, such that, $V_2 \subseteq V$. These vertices consist of $Source_2$, $Destination_2$ and S_2 sequence.
- An Edge in G_2 is represented by E_2 which consists of a communication link (v_{1_2}, v_{2_2}) between two vertices (v_{1_2}, $v_{2_2} \in V_2$).



- α_2 consists of two types of vertex attributes, namely, EndPoint Group (Eg_2) and S_2 sequence. Here, $\alpha_2 = E_{g_2} \cup S_2$, where E_{g_2} denotes an EPG_2 and S_2 denotes service function chain sequence to handle network traffic by network function boxes. The composed graph G_2 at time t_2 shows that which communication is allowed between network EP_2 and what S_2 sequence is required for that network communication.
- β_2 consists of two attributes of an edge, namely, $\beta_{Protocol_2}$ and β_{Ports_2}. The $\beta_{Protocol_2}$ represents the attribute of an edge that comprises values of TCP or UDP or ANY, where ANY represents both TCP and UDP traffic. Moreover, the β_{Ports_2} represents another type of an edge attribute that consists of a set of destination port numbers of applications. Here, β_{Ports_2} consists of a set of integer values or ANY for a policy.

In fifth step, new flow rules and shortest path are computed between $Source_2$ and $Destination_2$ as per new network policies. Moreover, the newly computed flow rules are installed along the path in the flow tables of switches and cached at controller for further processing. The overall proposed solution is presented in Figure 4.6.

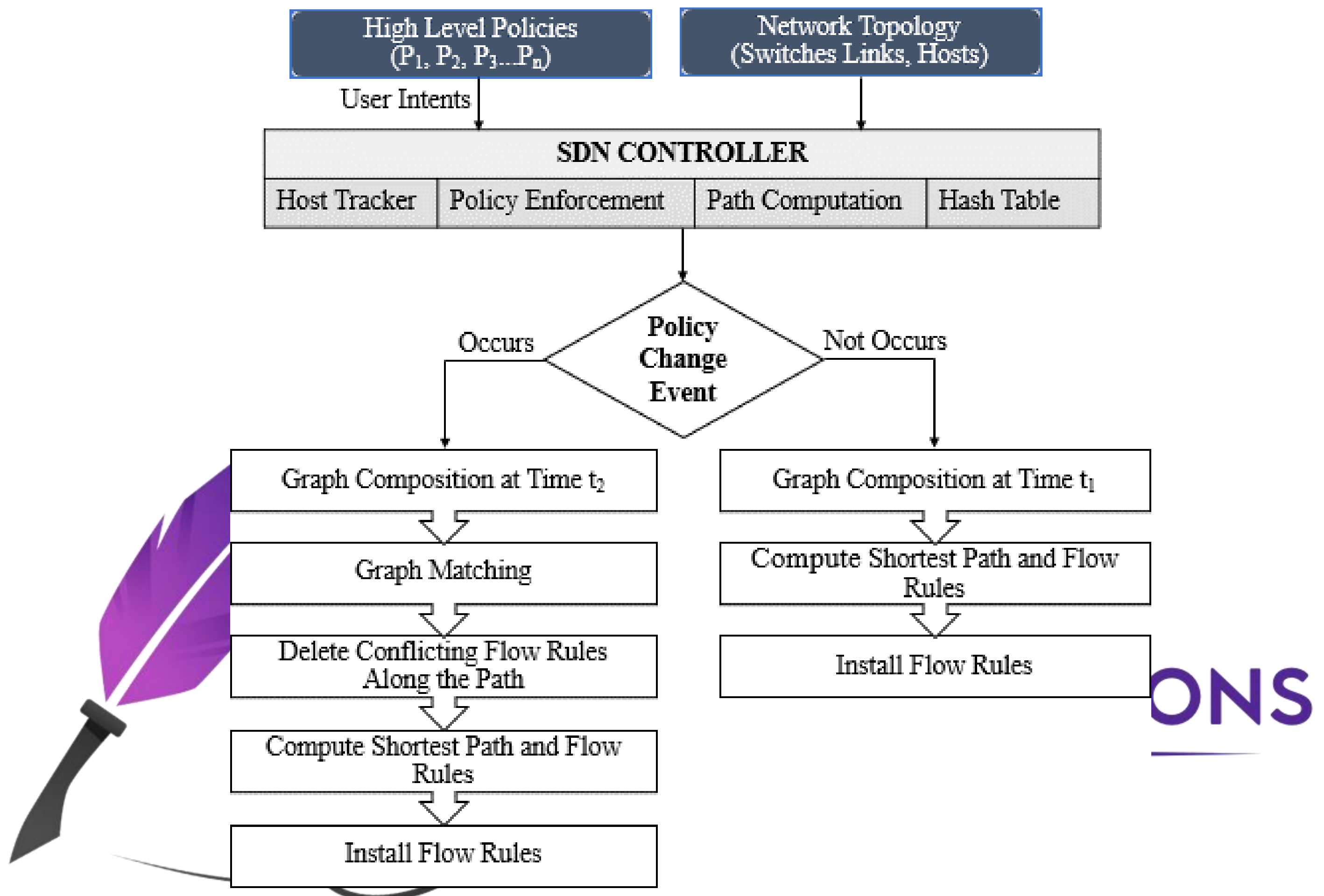


Figure 4.6: Graph-Based Proposed Solution

4.5.1 Graph-Based Policy Representation

In graph-based proposed solution, the network policies are represented in 6 tuple < Source, Destination, Protocol, Ports, Service Function Chain, Action > and multi-attributed graph is constructed with the help of PGA for the composition of these policies. In traditional networking environment, the network policy implementation process is largely manual and network administrator translates high level policies into low level commands to implement these policies on network devices. Due to this distributed and manual policy implementation process, policy change takes lot of time and suffers inefficiencies, like, policy overlapping, policy conflicts etc. These issues

become worse in automated network environments, for example, SDN applications in Enterprise networking environments, Network Functions Virtualization (NFV), cloud infrastructures etc.

Consider, an example of three network policies to understand the above-mentioned policy composition issues. Suppose we have three network policies to be defined at controller as follows:

Policy-1 (P_1):

Faculty has access to LMS server via TCP Port 20 through FW and LB Service Function Chain

Policy-2 (P_2):

Employees (Faculty and Staff) have access to the servers via TCP Ports 20, 25, 80 through FW Service Function Chain

Policy-3 (P_3):

Faculty can communicate with Staff via TCP Ports 20, 25, 80, 587, 993 through FW Service Function Chain

The above three policies (P_1, P_2, P_3) are represented in 6 tuple as:

P_1 = <Faculty, LMS, TCP, 20, (FW, LB), Permit>

ACADEMIC SOLUTIONS

P_2 = <Employees, Servers, TCP, (20, 25, 80), FW, Permit>

P_3 = <Faculty, Staff, TCP, (20, 25, 80, 587, 993), FW, Permit>

There is conflict between policies P_1 and P_2, because through P_1 only Faculty can access LMS Server and Staff cannot access it. However, through P_2 Staff can access all Servers including LMS. So, there is policy conflict between P_1 and P_2, as, through P_1 access is denied and through P_2 access is permitted. Similarly, both Policies P_1 and P_2 also overlap, because by using P_1 Faculty access is Permitted to LMS via TCP Port 20 and in P_2 Employees access is permitted to LMS including all Servers via TCP Ports 20, 25, 80. Here, Employees also include Faculty. The third policy P_3 is conflict free and non-overlapping, because it has no conflict and overlapping with other two policies. To implement these policies correctly, we need human operator who manually composes these policies into a composite policy via IF-THEN-ELSE in a program. This is shown below in an SDN program:

- -

If_match (SRC = Faculty, DEST = LMS, PROT = TCP, PORT = 20)

Permit FW>>LB Service Function Chain

Else If_match (DEST = LMS)

Deny

Else If_match (SRC = Employee, DEST = Servers, PROT = TCP, PORT = 20,25,80)

Permit FW Service Function Chain

Else If_match (SRC = Faculty, DEST =Staff, PROT = TCP, PORT = 20,25,80,587,993)

Permit Bidirectional

- -

These network policies can also be sketched with the help of Policy Whiteboarding and Set Operator [210] as shown in Figure 4.7. It shows that Faculty is part of Employees and LMS is part of Servers. So, Faculty is subset of Employees and LMS is subset of Servers.

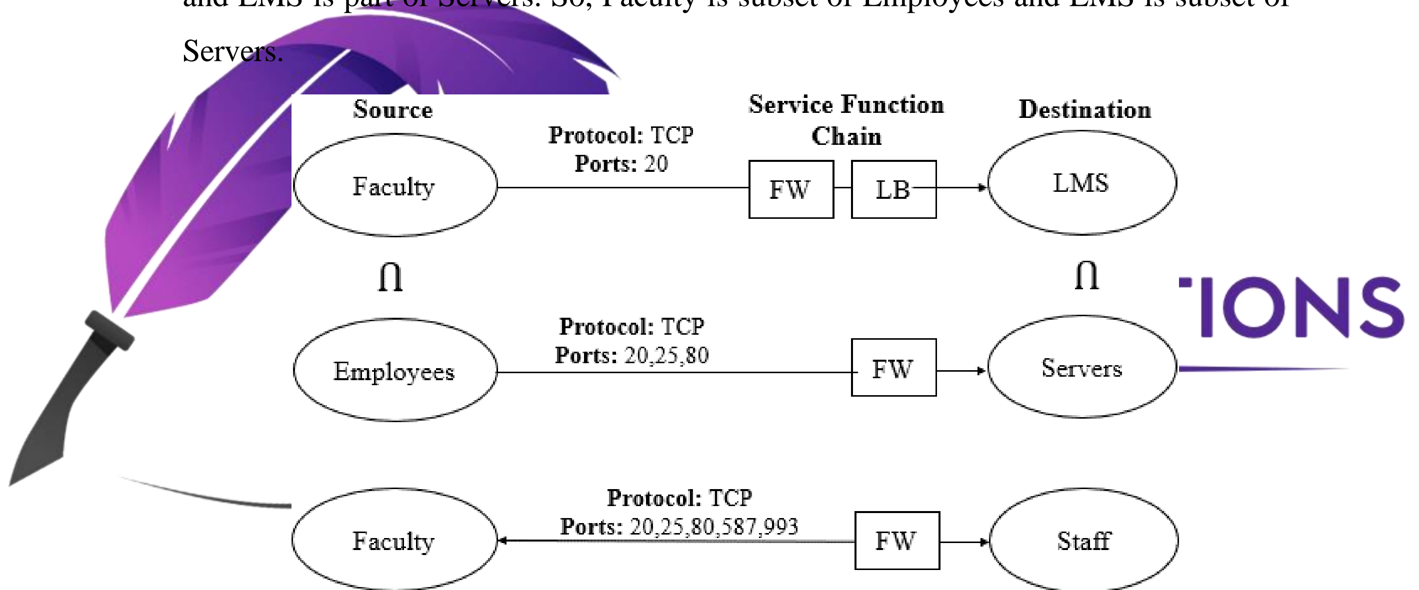


Figure 4.7: Policy Whiteboarding

The above three policies are expressed correctly via graph composition as shown in Figure 4.8. It shows that faculty can access LMS via two function boxes forming correct service function chains required by both policies. As there is no relationship between Staff and LMS, so this is basically correctly implementing exclusive access to LMS which is required by P_1. The other Servers excluding LMS are denoted by Set Operator *Difference ('-'), which* allows communication with all Employees along with Faculty.

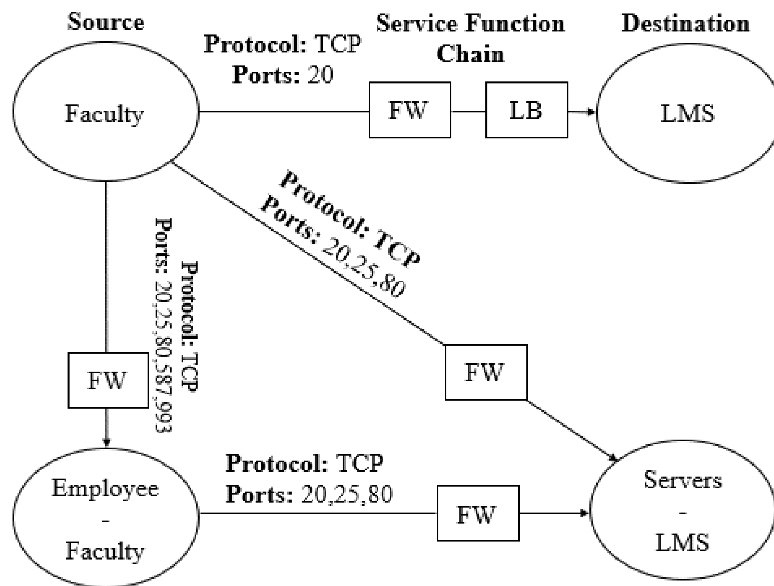


Figure 4.8: Graph Composition

ACADEMIC SOLUTIONS

It means that graph abstraction is very useful to express policies naturally and leverage of graph structure can be taken to do automatic composition. PGA [58, 210] has implemented this concept which enables users to easily specify and naturally draw some diagrams to capture network policies. The general PGA system architecture expresses policies via graph-based abstractions. Different stakeholders (Users/Tenants/Admins) in addition to SDN Apps, produce network policies as graphs. These graph-based policies are further sent to the Graph Composer through a PGA User Interface (UI) which gets additional information from external sources for the efficient policy formulation. After that, the Graph Composer provides conflict free graph after fixing all Errors/Conflicts. Figure 4.9 shows that each membership predicate is specified as a label, for example, LMS, Faculty, Staff, etc. which can be a Boolean Expression for all Labels.

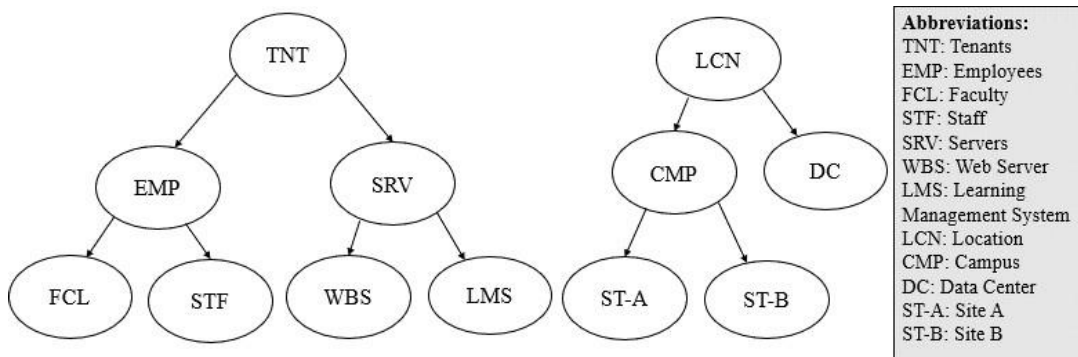


Figure 4.9: Input Label Namespace Hierarchy

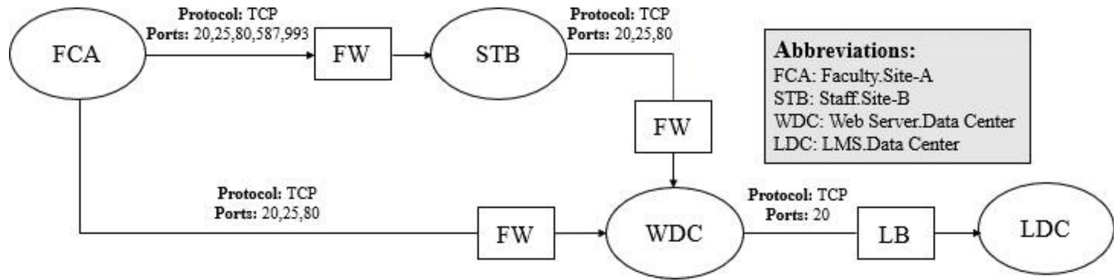


Figure 4.10(a) Composed Graph based on Label Namespace at Time t_1

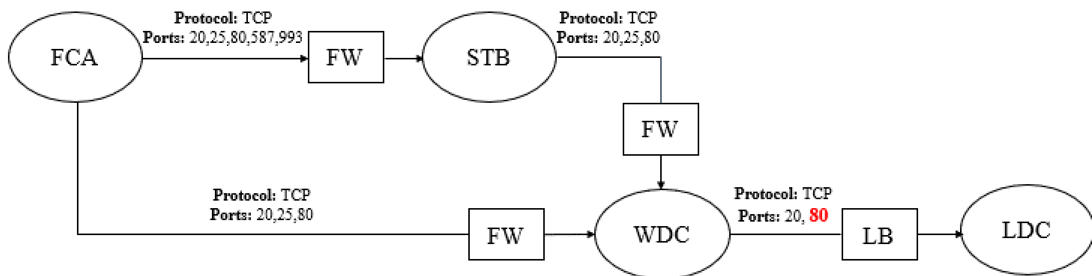


Figure 4.10(b) Composed Graph based on Label Namespace at Time t_2

Figure 4.10: Composed Graph based on Label Namespace at Different Time Instances

ACADEMIC SOLUTIONS

4.5.2 Flow Rules Caching

As we are using reactive mode for flow installation, therefore, controller installs the flow rules at the data plane when a flow is generated in on-demand fashion. When controller receives request from data plane to compute the flow rules for the flow, controller computes flow rules based on network topology, network policies, and other application requirements. After this, as per proposed approach, it caches one copy of the flow rules at controller and installs the flow rules at the corresponding switches along the shortest path.

We have used Hash Table data structure [207] for caching flow rules at controller which maps keys to unique buckets/slots. It uses Hash Function to compute an index into an array of buckets/slots which helps to retrieve the desired flow rule. In this proposed approach, we express keys and buckets/slots as tuples. The key tuple consists of network policies in 6 tuple <Source, Destination, Protocol, Ports, Service Function Chain, Action> while the bucket/slot tuple comprises of Flow Rules in 5 tuple, that is,

<Source, Destination, Shortest path between Source and Destination, Action, Timeout>. In simulation environment, we choose Hash Table length of 10000 (0 to 9999) slots due to the limited number of nodes (Hosts/Switches). Moreover, we have used Consistent Hash Function [208] as it produces the same output for a specific input every time when Hash Function is executed. It in turn helps to track specific flow rule against a defined network policy more accurately. Whenever any change in policy is detected via Algorithm 4.2, then as per key tuple, Hash Function is executed which points to specific bucket/slot. Based on Hash Function result, respective flow rule is deleted from the Hash Table of controller. Moreover, as per new policy, new flow rules are computed and installed along the shortest path in addition to caching these flow rules at controller. Figure 4.11 shows that hash function is applied based on key tuple, which points to specific bucket/slot for caching flow rules at specific index in hash table.

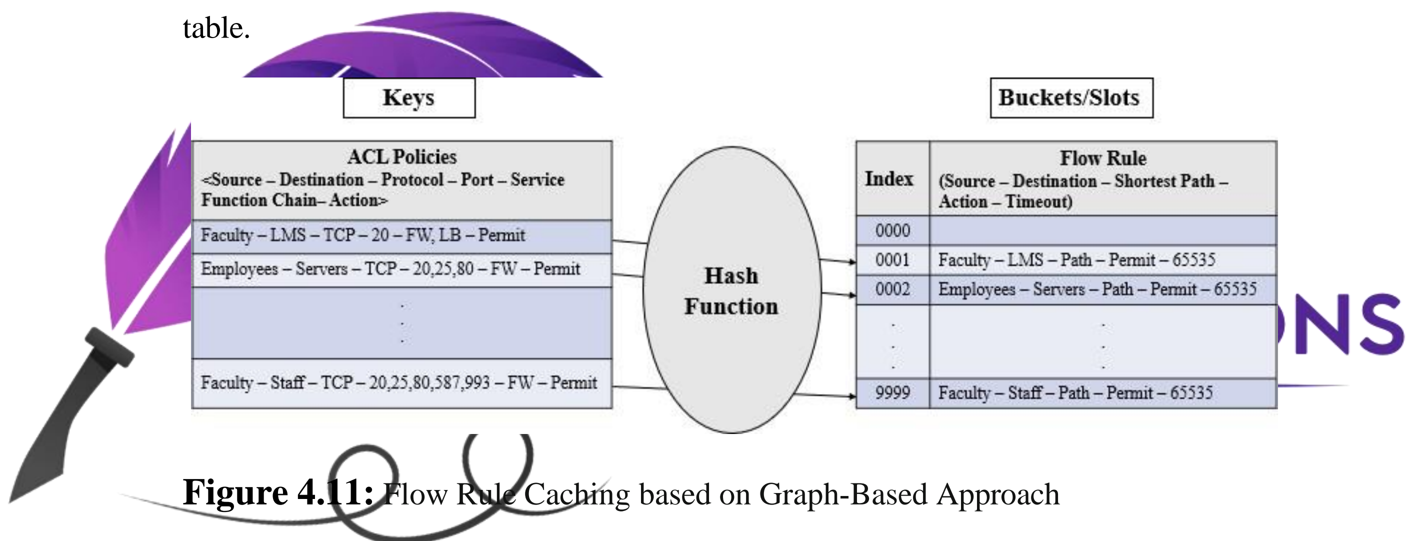


Figure 4.11: Flow Rule Caching based on Graph-Based Approach

4.5.3 Detecting Policy Change via Graph Matching

Network policies change with the passage of time as per requirements of tenants or change in application environment. The multi-attributed graphs for the network policies at two different time instances, say, at time t_1 and t_2 are computed respectively. The whole network policy composition and computing process is shown in Figure 4.6. The graph at time t_1 is shown in Figure 4.10(a) which is composed on basis of three network policies (P_1, P_2, P_3). At time t_2 policy changes which states that LMS is also accessible via port 80, then the composed graph is shown in Figure 4.10(b). After this, these two graphs are compared by using graph matching algorithm 4.2 in order to detect that whether the network policies are changed/modified or not. By change it means whether

the policies are modified, deleted or new policies are added. The matching is performed as follows.

Suppose a directed multi attributed graph $G = (V, E, R_v, R_e)$ which is defined by 4 tuple. Here, V is a finite set of vertices and E is finite set of edges. These vertices and edges are associated with attributes that define their properties. In graph G, R_v is set of vertex attributes and R_e is set of edge attributes. Here:

- V is finite set of vertices which consist of three types, that is, Source, Destination and Service Function Chain
- $E \subseteq V \times V$ is a set of directed edges. Here, an edge E consists of a communication link (v_1, v_2) between two vertices $(v_1, v_2 \in V)$
- $R_v \subseteq V \times A_v$ is a relation associating attributes to vertices, that is, R_v is the set of couples (v_i, A) such that vertex v_i is attributed by A
- $R_e \subseteq V \times V \times A_e$ is a relation associating attributed to edges, that is, R_e is the set of triples (v_i, v_j, A) such that edge (v_i, v_j) is attributed by A

The graph similarity is defined for two multi-attributed graphs $G_1 = (V_1, R_{v_1}, R_{e_1})$ and $G_2 = (V_2, R_{v_2}, R_{e_2})$ over the same sets of vertices and edges attributes A_v and A_e, such that $V_1 \cap V_2 = \emptyset$ and $E_1 \cap E_2 = \emptyset$. At first, vertices matching is checked to measure the graph similarity via matching each vertex with an empty set of vertices of the other graph. After that, a collection of common features (vertices, edges, vertices attributes, edges attributes) of both graphs are identified to measure the similarities between both graphs [211, 212]. The following definitions are used in graph matching via graph isomorphism with the help of exact graph matching which is used in detection of change in network policies. In scenario of this work, the graphs are generated at time t_1 and at time t_2 based on network policies. Researchers intend to find instances of graph at time t_1 within graph at time t_2 allowing additional edges which is called *graph monomorphism.* Finding instances with no additional edges is called *subgraph isomorphism* and finding one-to-one mapping for all nodes and edges is called *graph isomorphism.* The intention is to exploit concept of *graph isomorphism* for detection of policy change at different instances of time.

Definition 1: Suppose $G = (V, E, \alpha, \beta)$ which represents directed and multi-attributed graph for communication at time t_1 where:

ACADEMIC SOLUTIONS

- V is a finite set of vertices
- $E \subseteq V \times V$ is set of edges and $e(i, j)$ shows a directed edge which is communicating from vertex i to j
- $\alpha: v \rightarrow A_v$ is a function which assigns unique attributes to each vertex in G such that $\alpha(i) \neq \alpha(j)$
- $\beta: E \rightarrow A_e$ is a function which assigns unique attributes to each edge in G.

Definition 2: A subgraph of G with $V_1 \subseteq V$ is represented by $S(V_1) = (V_1, E \cap V_1 \times V_1, \alpha, \beta)$

Definition 3: A function $f: V_1 \rightarrow V_2$ is a *graph monomorphism* from graphs $G_1 = (V_1, E_1, \alpha_1, \beta_1)$ to $G_2 = (V_2, E_2, \alpha_2, \beta_2)$, iff:

- $\alpha(v) = \alpha(f(v))$, for all $v \in V$
- $\beta(e) = \beta(e_1)$, for all edges $e = (v_1, v_j) \in E$ and $e_1 = (f(v_i), f(v_j)) \in E_1$

Definition 4: A function $f: V_1 \rightarrow V_2$ is a *subgraph isomorphism* from graphs G_1 to G_2, if it satisfies definition 3 and additionally for every edge following conditions hold.

$e_2 = (v_{2i}, v_{2j}) \in E_2 \cap f(V) \times f(V)$ *exists an edge*

$e_1 = f^{-1}(v_{2i}) \times f^{-1}(v_{2j}) \in E$ with $v(e_2) = v(e_1)$

Definition 5: A function $f: V_1 \rightarrow V_2$ is a *graph isomorphism* from graphs G_1 to G_2, if, f is bijective function and it satisfies definition 4.

In this research work, Graph Isomorphism Approach is adopted which is based on divide and conquer principle. At first, two Graphs G_1 (input graph) and G_2 (model graph) are taken which are generated on the basis of network policies that are implemented at times t_1 and t_2 respectively. These policies are represented with the help of graphs in Figure 4.10. The model graph is partitioned into disjoint subgraphs and subgraph isomorphism is checked for those disjoint subgraphs within the input graph. These graphs are merged to provide subgraph isomorphism of the full graph. Starting from full model graph G_2, set V is divided into two separate sets, such that, $V = V_1 \cup V_2$. The matching module is applied on both subsets $S(V_1)$ and $S(V_2)$. The outputs of partial matching modules are sent to merge module for subgraph isomorphism, if

ACADEMIC SOLUTIONS

possible. The V_1 and V_2 are further subdivided till single vertex is reached. In this way, hierarchy of partial matches are formed with respect to binary tree.

The overall matching algorithm works based on four categories of nodes as given in Algorithm 4.2. At topmost of hierarchy in nodes is the *input-node*, the 2nd highest node is *A-vertex-checker*, 3rd one is *E-subgraph-checker* and the 4th one is *m-model-nodes*. The *input-node* is the entry point of matching algorithm. There are multiple edges from the *input-node* to the *A-vertex-checker*. At the time of execution, the input graph is forwarded to *A-vertex-checkers,* which is quite simple matching mechanism. Every *A-vertex-checker* characterizes a specific attribute of a vertex in model graph which is attributed by *A-vertex checker,* if $\alpha(v) = A$. *A-vertex-checker* takes vertices of input graph from *input-node.* If the input vertex has attribute 'A' then it is locally stored and forwarded to all descendant nodes. Each *A-vertex-checker* has a single or multiple outgoing edges which lead to *m-model-nodes* or *E-subgraph-checkers.* The 3rd type of nodes are E-subgraph-checkers that contain minimum of two vertices and one edge. These nodes have two parents of either A-vertex-checker or E-subgraph-checker. Each *E-subgraph checker* has two parent nodes which are either A-vertex-checker or E-subgraph-checker types.

ACADEMIC SOLUTIONS

The graph denoted by E-subgraph-checker consists of two graphs of its parent nodes (n_i, n_j), which are represented by graphs $g_i (V_i, E_i, \alpha_i, \beta_i)$ and $g_j (V_j, E_j, \alpha_j, \beta_j)$ then the graph denoted in *E-subgraph-checker* n_k is given by $g_k (V_k, E_k, \alpha_k, \beta_k)$ with $V_k = V_i \cup V_j$ and $E_k = E_i \cup E_j \cup E$. Here, E is the collection of edges linked to n_k. any edge $e \in E$ is either an edge from g_i to g_j or from g_j to g_i. At execution time, the *E-subgraph checker* n_k will receive instances t_i of g_i and instances t_j of g_j from its parent node and tries to merge them into an instance t_k for g_k. The two instances can be combined if they are disjointed and the edges specified in E exist between t_i and t_j. In addition, no edge exists in the input that is not specified by E. If both conditions are satisfied, then t_i and t_j will be concatenated and t_k is an instance of g_k. Each new instance is locally stored and forwarded to all descendant nodes. The m-model-nodes are the last type of node which is connected to just one parent node. The graph represented in parent node is identical to model m. So, any instance found in parent node is sent to the *m-model-node* where it is stored as an instance of model m. The exact graph matching is shown in Algorithm 4.2.

Algorithm 4.2: Graph-Based Policy Change Detection

Input: Input Graph and Model Graph at times t_1 and t_2 respectively

Output: Graphs are Matched OR Graphs are NOT Matched

1: **Procedure** GraphMatch (G= (V, E, α, β))

2: GE = E ▷E is the edges of the Input-graph which are accessible globally

Module-1

3: **For Each** *Attribute-Vertex* v ∈ V **Do** ▷Input nodes of the graph

4: **For Each** *Attribute-Vertex-Monitor* **Do**

5: Call *Attribute-Vertex-Monitor (v)*

6: **End Inner For Loop**

7: **End Outer For Loop**

Module-2

8: **Procedure** *Attribute-Vertex-Monitor (v)* ▷Checking vertices for matching

9: **If** α(v) = A **Then**

10: $A_{rr}[0] = v$ ▷A_{rr} represents instance of graph G

11: store A_{rr} in the Local-Memory

12: **For Each** Descendent Node n

13: **If** n is *Edge-Sgraph* **Then**

14: Call *Edge-Sgraph-Monitor* (A_{rr}) **Then**

15: **End If**

16: **If** n is *M-Model-Node* **Then**

17: Call *M-Model-Node* (A_{rr})

18: **End If**

19: **End For**

20: **End If**

Module-3

21: **Procedure** *Edge-Sgraph-Monitor* (A_{rr}) ▷Checking subgraphs for matching

22: **If** called by *Left-Parent-Node* **Then**

23: **For Each** instances A_{rr_R} ∈ Local-Memory of Right-Parent

24: **If** A_{rr} and A_{rr_R} are disjoint **And** ∀ (x, y, A) ∈ E edge 'e' exists, such that e =(A_{rr}[x], A_{rr_R} [y] ∈ GE & v(e) = A **And** there exists no additional edge **Then**

ACADEMIC SOLUTIONS

25: $A_{new} = A_{rr} + A_{rr_R}$ ▷The instances of G are concatenated

26: Save A_{new} in Local-Memory

27: **For Each** Descendant-Node n

28: **If** n is *Edge-Sgraph-Monitor* **Then**

29: Call *Edge-Sgraph-Monitor* (A_{new})

30: **End If**

31: **If** n is *M-Model-Node* **Then**

32: **Call** *M-Model-Node* (A_{new})

33: **End If**

34: **End For**

35: **End If**

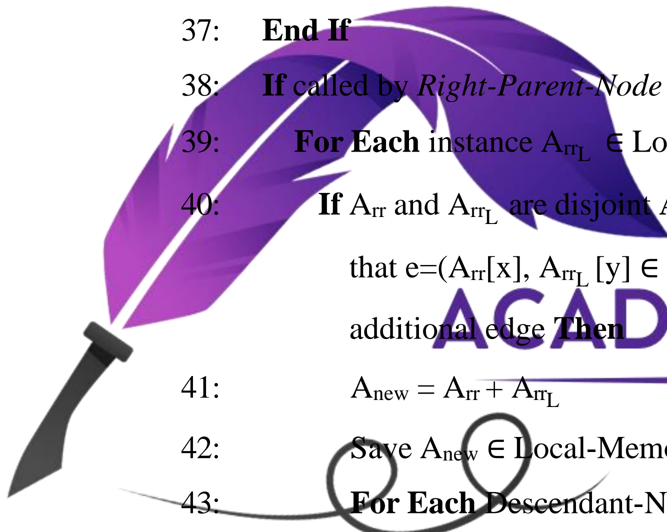
36: **End For**

37: **End If**

38: **If** called by *Right-Parent-Node* **Then**

39: **For Each** instance A_{rr_L} ∈ Local-Memory of Left-Parent

40: **If** A_{rr} and A_{rr_L} are disjoint **And** ∀ (x, y, A) ∈ E edge 'e' exists, such that e=(A_{rr}[x], A_{rr_L} [y] ∈ GE & v(e) = A **And** there exists no additional edge **Then**



41: $A_{new} = A_{rr} + A_{rr_L}$ ▷The instances are concatenated

42: Save A_{new} ∈ Local-Memory

43: **For Each** Descendant-Node n

44: **If** n is *Edge-Sgraph-Monitor* **Then**

45: **Call** *Edge-Sgraph-Monitor* (A_{new})

46: **End If**

47: **If** n is *M-Model-Node* **Then**

48: **Call** *M-Model-Node* (A_{new})

49: **End If**

50: **End For**

51: **End If**

52: **End For**

53: **End If**

Module-4

54: **Procedure** *M-Model-Node (instance Arr)* ▷Storing instance of Model Graph

55: Save Arr in Local-Memory

56: Print the new instance A_{rr} of the *Model M*

Table 4.2: Complexity of Graph-Based Policy Change Detection

Algorithm 4.2	No. of Times Instructions Executed
Input: Input Graph and Model Graph at times t_1 and t_2 respectively	
Output: Graphs are Matched OR Graphs are NOT Matched	
1: **Procedure** GraphMatch (G= (V, E, α, β))	1
2: GE = E ▷E is the edges of the Input-graph which are accessible globally	1
Module-1	
3: **For Each** *Attribute-Vertex* v ∈ V **Do** ▷Input nodes of the graph	n+2 (
4: **For Each** *Attribute-Vertex-Monitor* **Do**	n+1)(n+2)
5: Call *Attribute-Vertex-Monitor (v)*	(n+1)(n+1)
6: **End Inner For Loop**	
7: **End Outer For Loop**	
Module-2	
8: **Procedure** *Attribute-Vertex-Monitor (v)* ▷Checking vertices for matching	1
9: **If** α(v) = A **Then**	1
10: A_{rr}[0] = v ▷ A_{rr} represents instance of graph G	1
11: store A_{rr} in the Local-Memory	1
12: **For Each** Descendent Node n	n+2
13: **If** n is *Edge-Sgraph* **Then**	n+1
14: Call *Edge-Sgraph-Monitor* (A_{rr}) **Then**	n+1
15: **End If**	
16: **If** n is *M-Model-Node* **Then**	n+1

ACADEMIC SOLUTIONS

17: Call *M-Model-Node* (A_{rr})	n+1
18: **End If**	
19: **End For Loop**	
20: **End If**	
Module-3	
21: **Procedure** *Edge-Sgraph-Monitor* (A_{rr}) ▷Checking subgraphs matching	1
22: **If** called by *Left-Parent-Node* **Then**	1
23: **For Each** instances A_{rr_R} ∈ Local-Memory of Right-Parent	n+2
24: **If** A_{rr} and A_{rr_R} are disjoint **And** ∀ (x, y, A) ∈ E edge 'e' exists, such that e =(A_{rr}[x], A_{rr_R} [y] ∈ GE & v(e) = A **And** there exists no additional edge **Then**	n+1
25: $A_{new} = A_{rr} + A_{rr_R}$ ▷The instances of G are concatenated	n+1
26: Save A_{new} in Local-Memory	n+1
27: **For Each** Descendant-Node n	(n+1)(n+2)
28: **If** n is *Edge-Sgraph-Monitor* **Then**	(n+1)(n+1)
29: Call *Edge-Sgraph-Monitor* (A_{new})	(n+1)(n+1)
30: **End If**	
31: **If** n is *M-Model-Node* **Then**	(n+1)(n+1)
32: **Call** *M-Model-Node* (A_{new})	(n+1)(n+1)
33: **End If**	
34: **End Inner For Loop**	
35: **End If**	
36: **End Outer For Loop**	
37: **End If**	
38: **If** called by *Right-Parent-Node* **Then**	1
39: **For Each** instance A_{rr_L} ∈ Local-Memory of Left-Parent	n+2
40: **If** A_{rr} and A_{rr_L} are disjoint **And** ∀ (x, y, A) ∈ E edge 'e' exists, such that e=(A_{rr}[x], A_{rr_L} [y] ∈ GE & v(e) = A **And** there exists no additional edge **Then**	n+1
41: $A_{new} = A_{rr} + A_{rr_L}$ ▷The instances are concatenated	n+1

ACADEMIC SOLUTIONS

42: Save A_{new} ∈ Local-Memory	n+1
43: **For Each** Descendant-Node n	(n+1)(n+2)
44: **If** n is *Edge-Sgraph-Monitor* **Then**	(n+1)(n+1)
45: **Call** *Edge-Sgraph-Monitor* (A_{new})	(n+1)(n+1)
46: **End If**	
47: **If** n is *M-Model-Node* **Then**	(n+1)(n+1)
48: **Call** *M-Model-Node* (A_{new})	(n+1)(n+1)
49: **End If**	
50: **End Inner For Loop**	
51: **End If**	
52: **End Outer For Loop**	
53: **End If**	
Module-4	
54: **Procedure** *M-Model-Node (instance Arr)* ▷Storing Model Graph instance	1
55: Save Arr in Local-Memory	1
56: Print the new instance A_{rr} of the *Model M*	1
Total Complexity = $12n^2 + 41n + 45$ Worst Case Complexity = $O(n^2)$	

ACADEMIC SOLUTIONS

4.5.4 Comparison of Algorithms Complexities

In order to theoretically show the comparison of both Matrix-Based and Graph-Based policy change detection algorithms, the complexities are computed and shown in Table 4.1 and Table 4.2, respectively. Both tables show that complexity in constant numbers of Graph-Based approach is lesser as compared to Matrix-Based approach. It is also noted that Graph-Based approach utilizes complex network policies based on real time network traces using high level names instead of low-level destination IP addresses and add more policy constructs like, source, destination, protocol and ports. It then represents and formalizes these policies via multi-attributed graphs along with removing conflicts and overlapping for efficient handling of network policy change mechanism. However, the worst case complexity still remains same even by formalizing complex network policies and more policy constructs which is good.

4.5.5 Checking the Flow Rules that Violate New Policies

In case of detection of policy change at time t_2, the controller searches all flow rules that are installed by the policy at time t_1 from all the switches on the basis of shortest path between source and destination. Controller keeps policies in Hash Table Keys and forwarding rule along with path in respective bucket/slot. When bucket/slot is traversed to check flow rules installed by the conflicting policy, it returns all paths along with flow rules. If source and destination are matched with a flow rule in a switch flow table then respective flow rule is deleted as per section 4.5.6.

4.5.6 Deleting Flow Rules that Violate New Policies

As per policy violation mechanism in section 4.5.5, the controller starts flow rules deletion process. It iterates over switches in the path and forwards a flow deletion command via flow table modification message (OFPFC_DELETE) [213] and appends egress switch port number for flow rule deletion from switch flow table.

4.5.7 Installing Flow Rules as Per New Policies

The controller installs new flow rules as per new policies after deletion of conflicting flow rules. It computes shortest path between source and destination as per policy and installs flow rules on switches along the path via flow table modification message (OFPFC_ADD) [213]. This mechanism helps to minimize packet violation percentage and increases network efficiency by increasing successful packet delivery and network throughput.

ACADEMIC SOLUTIONS

4.6 Summary

In this chapter, we have discussed proposed solutions of the identified research problem with the help of diagrams. The proposed solutions consist of two approaches and these are matrix-based policy change detection and implementation, called ‘EPE’ and graph-based policy change detection and implementation, called ‘GPE’ for the efficient handling of network policy change at controller. In addition, we have represented network policies in 5 tuple based on EPE approach and traverse these policies via matrices for policy change detection. Furthermore, we have taken complex network policies from a campus area network, represented these policies in 6 tuple and

constructed multi-attributed graphs with the help of PGA for policy composition and change detection process. Moreover, we have shown hash table data structure implementation for caching flow rules at controller in both proposed approaches based on destination IP addresses and complex network policies. Additionally, we have presented algorithms for network policy change detection and calculated complexity of each algorithm. Finally, we have presented the mechanism to detect and delete the conflicting flow rules and installing new flow rules along the shortest path between source and destination based on changed network policies.

Chapter 5

Results and Discussion

5.1 Introduction

Chapter 5 comprises results and discussion of the proposed approaches for policy change detection and implementation. Section 5.2 consists of matrix based proposed solution by analyzing experimental results as compared to the existing approaches [52,58]. In section 5.3, a graph based approach is proposed to handle network policy change at controller and flow rules management at data plane. Table 5.1 shows list of software used during simulation modeling.

Table 5.1: Software used in Simulation and Modeling

Software	Operation	Version
Mininet [30]	Network Emulator	2.2.0
POX [31]	Controller	Beta
Windows 10	Host OS	Version 10, 64 Bit
Ubuntu	Guest OS	16.01 LTS
Open vSwitch [213]	SDN Switch	OVS 2.8.0
Oracle Virtual Box [105]	Virtualization Platform	Version 10, 64 Bit
Python [214]	Programming Language	3.7

ACADEMIC SOLUTIONS

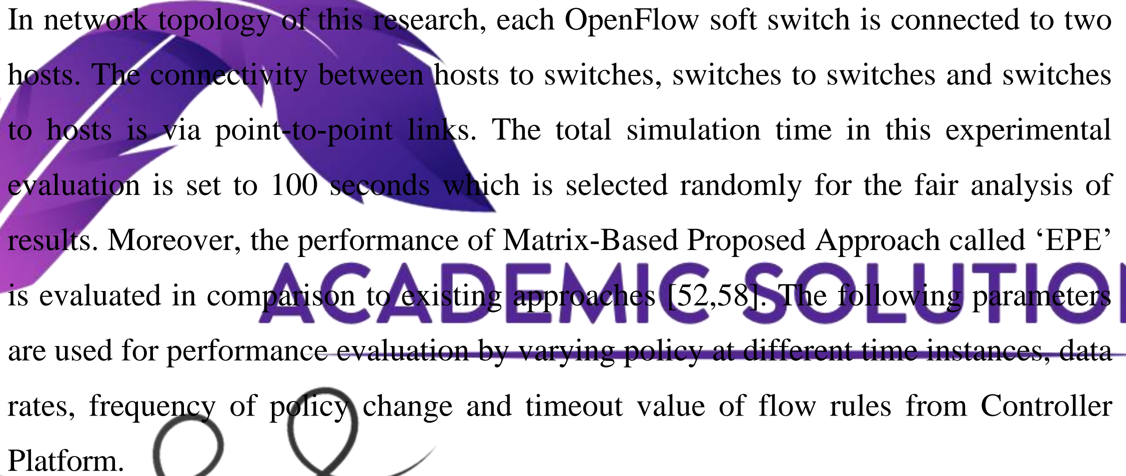
5.2 Experimental Results of Matrix-Based Proposed Approach

For this research work, Mininet Network Emulator [30] and POX [31] SDN Controller are chosen for the simulation of the network scenario which is set up in python on HP ProBook 4540s machine with an Intel Core i3-3110M, 2.4 GHz speed and 8 GB of RAM, running 64-bit Ubuntu Linux 16.01 LTS. A Network Topology is created in Mininet 2.2.0 by adding and configuring hosts, links, OpenFlow Soft Switches (OVS 2.8.0) which supports OpenFlow 1.3 and higher versions, and POX SDN controller. This topology is randomly generated which is calculated via uniform random function. Network Topology is created with the help of this program which consists of 20 hosts, 10 OpenFlow Soft Switches and POX SDN Controller. During this research, 1,000 to 80,000 packets are transmitted based on different network scenarios for performance evaluation. The details of simulation parameters are enlisted in Table 5.2.

Table 5.2: Simulation Parameters of Proposed Approaches

Parameter	Value
Network topology	Custom and random topology based on number of links and nodes
Number of switches	3-30
Number of links	10-300
Number of Policies	1-20
Number of Packets	1,000-80,000
Packet Size	64 KB
Bandwidth	10 Gbps
Simulation time	100 Seconds

In network topology of this research, each OpenFlow soft switch is connected to two hosts. The connectivity between hosts to switches, switches to switches and switches to hosts is via point-to-point links. The total simulation time in this experimental evaluation is set to 100 seconds which is selected randomly for the fair analysis of results. Moreover, the performance of Matrix-Based Proposed Approach called 'EPE' is evaluated in comparison to existing approaches [52,58]. The following parameters are used for performance evaluation by varying policy at different time instances, data rates, frequency of policy change and timeout value of flow rules from Controller Platform.



a) Packet Violation Percentage

Packet Violation Percentage (PVP) is the total number of packets that violate the policy to the total number of packets initiated at source nodes.

b) Normalized Overhead

Normalized Overhead (NOH) is the total number of transmissions in the network divided by total number of packets received successfully at destination nodes as per policy.

c) Successful Packet Delivery

Successful Packet Delivery (SPD) is the percentage of total number of packets that are delivered as per policy to the total number of packets initiated from source nodes.

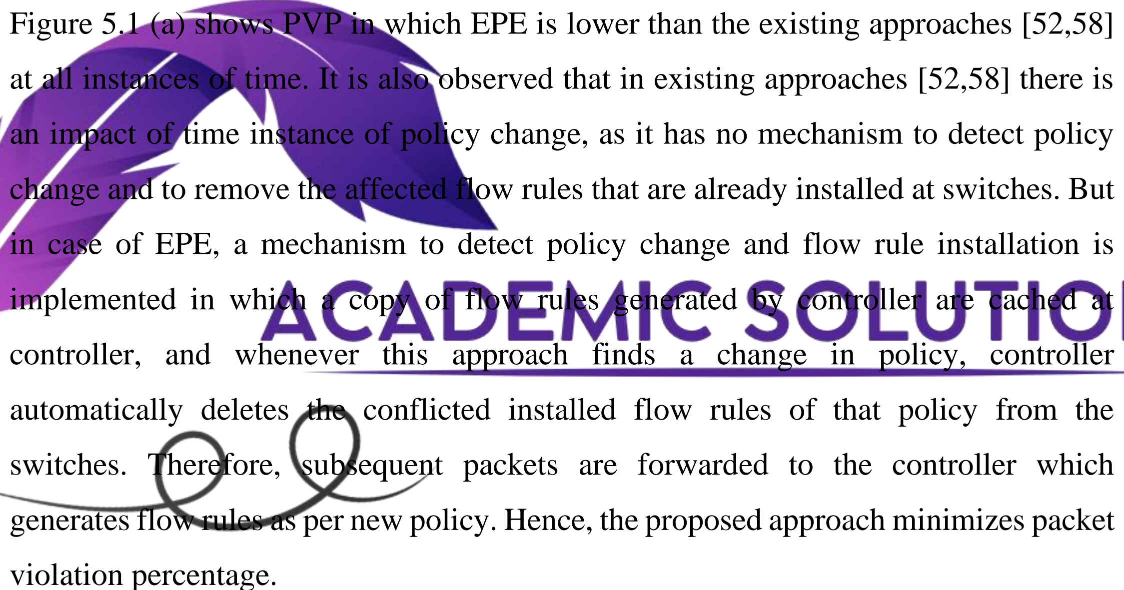
d) Network Throughput

Network Throughput (NTP) is the number of packets per second delivered successfully to the destination as per policy.

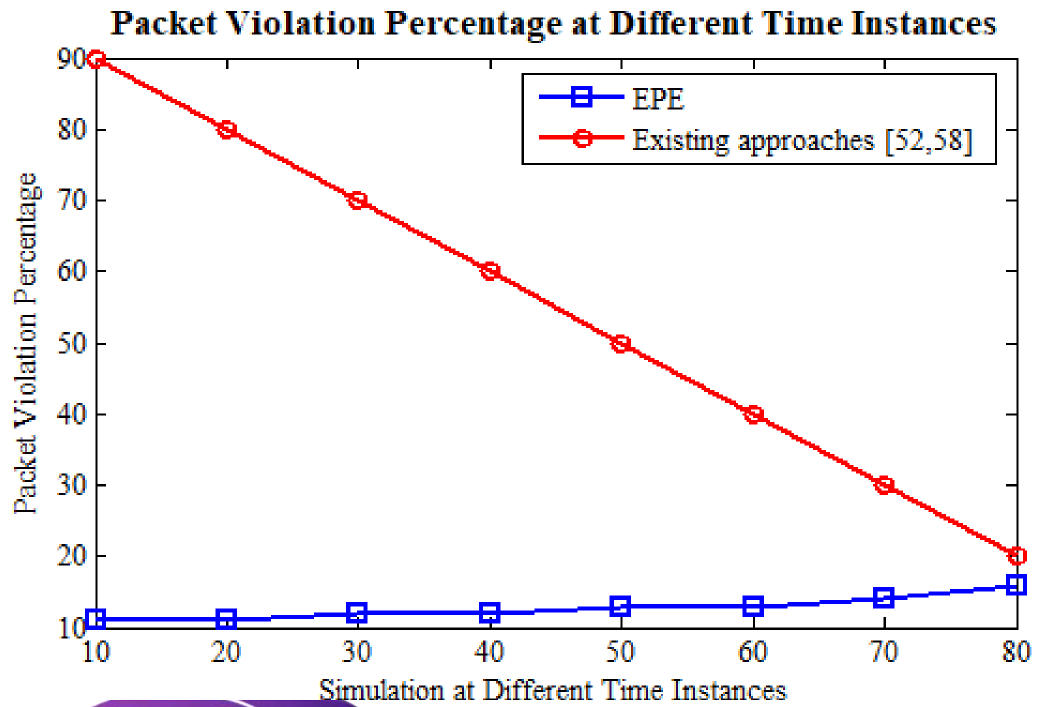
5.2.1 Simulation Results by Varying Network Policy at Different Time Instances

The simulation results are based on static and dynamic parameters. The static parameters include frequency of policy change, which is chosen as 4, data rate is chosen as 4 packets per second and hard timeout is set to 30 seconds. The dynamic parameter is chosen as time instance of policy change and it is randomly changed at 10 seconds(s), 20s, 30s, 40s, 50s, 60s, 70s and 80s. Based on the above parameters, simulation was run, and results are shown in Figure 5.1.

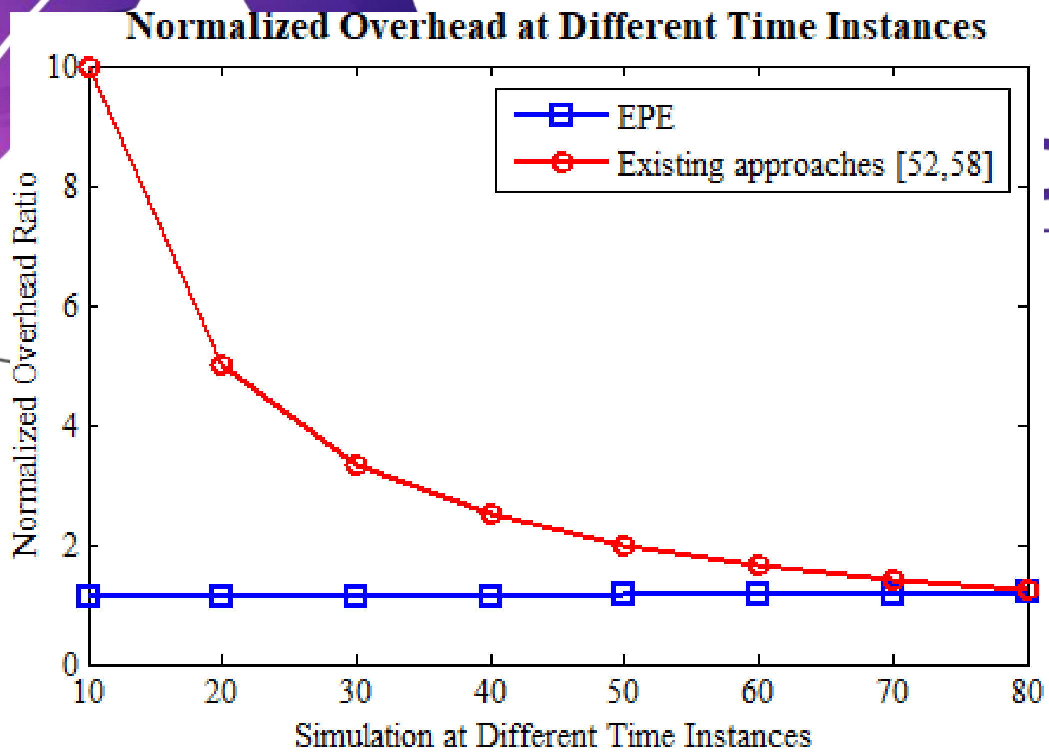
Figure 5.1 (a) shows PVP in which EPE is lower than the existing approaches [52,58] at all instances of time. It is also observed that in existing approaches [52,58] there is an impact of time instance of policy change, as it has no mechanism to detect policy change and to remove the affected flow rules that are already installed at switches. But in case of EPE, a mechanism to detect policy change and flow rule installation is implemented in which a copy of flow rules generated by controller are cached at controller, and whenever this approach finds a change in policy, controller automatically deletes the conflicted installed flow rules of that policy from the switches. Therefore, subsequent packets are forwarded to the controller which generates flow rules as per new policy. Hence, the proposed approach minimizes packet violation percentage.



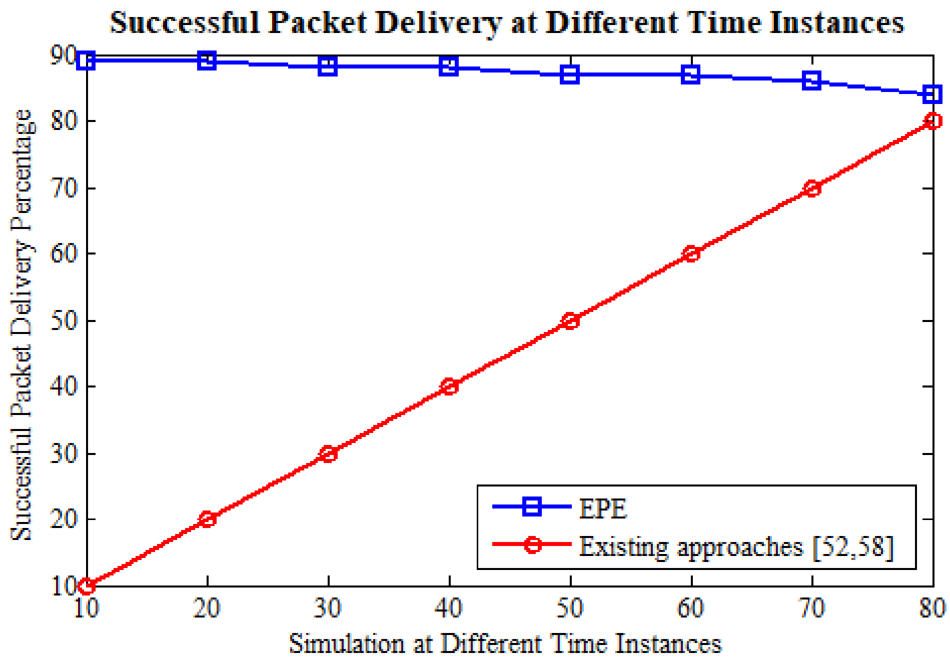
The EPE approach incurs some traffic overheads due to the addition and deletion of flow rules from control plane to data plane in case of network policy change. However, this traffic overhead is minimized in proposed approach by adding and deleting only those flow rules that are affected by the changed policy. Figure 5.1 (b) shows normalized overhead ratio is quite low in EPE as compared to the existing approaches [52,58]. This is due to the fact that number of packet violations are decreased due to the efficient detection of policy change and automatic flow rules installation according to the changed policy. So, normalized overhead ratio also decreases in case of EPE as compared to existing approaches [52,58].



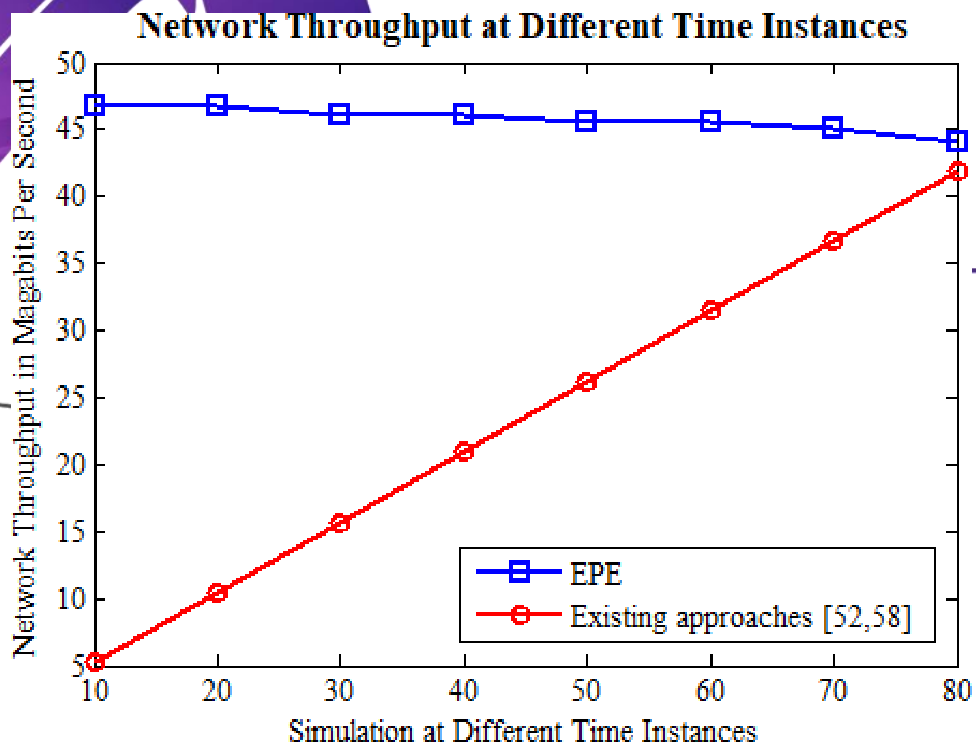
(a) Packet Violation Percentage



(b) Normalized Overhead



(c) Successful Packet Delivery



(d) Network Throughput

Figure 5.1: Simulation Results by Varying Network Policy at Different Time Instances

Simulation results in Figure 5.1 (c) show that EPE provides better SPD percentage because of policy change detection mechanism which auto detects policy change, deletes conflicting flow rules and installs new flow rules as per changed policies. However, no such policy change detection mechanism exists in existing approaches [52,58] due to which successful packet delivery suffers. The results show that SPD remains consistent in case of EPE, however, it is not consistent in case of existing approaches [52,58] and depends on time instance of policy change. Similarly, the network throughput is much better in case of our proposed approach as compared to the existing approaches [52,58] at all time instances of policy change as shown in Figure 5.1 (d).

5.2.2 Simulation Results by Varying Data Rate

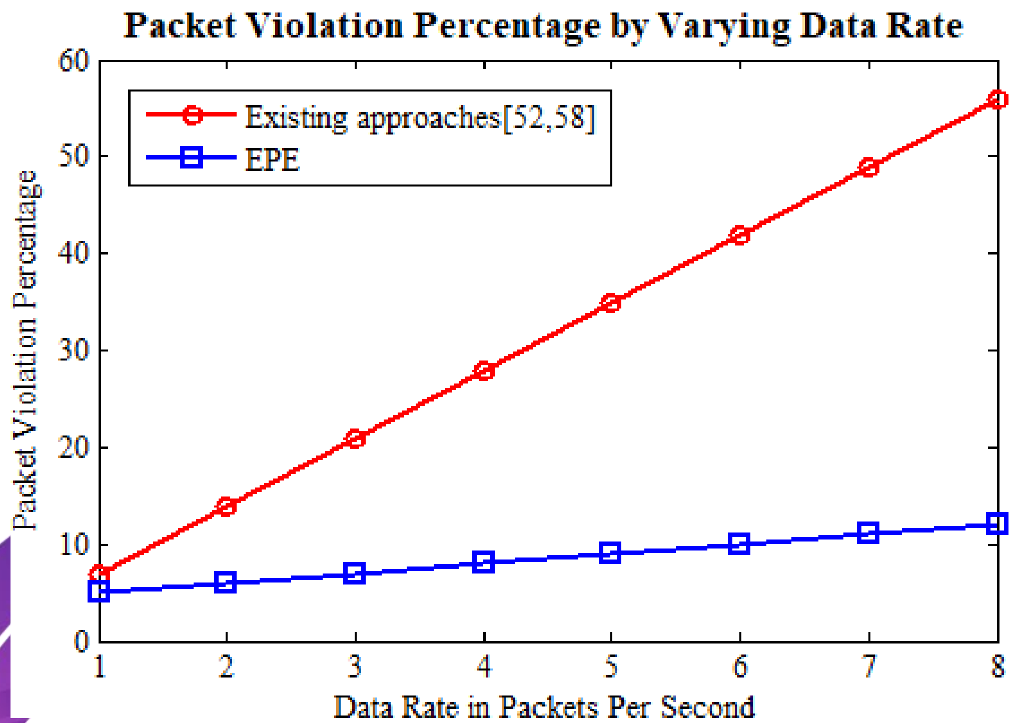
In this simulation, static parameters are set as: time instance of policy change to 50 seconds, frequency of policy change to 4, hard Timeout to 30 seconds and dynamic parameter as data rate to 1, 2, 3, 4, 5, 6, 7 and 8 packets per second.

Simulation results as shown in Figure 5.2 (a) indicate that packet violation percentage in EPE is increasing with the increase in data rate due to the fact that more packets are delivered to the invalid interfaces during the deletion and addition of new flow rules at data plane from the controller on network policy change. Moreover, in existing approaches [52,58] packet violation percentage also increases with the increase in data rate. The results show that EPE provides better results as compared to the existing approaches [52,58] with respect to packet violation percentage. Figure 5.2 (b) shows that normalized overhead ratio increases with the increase in data rate because more packet violations occur due to the increase in data rate in both approaches. However, in case of EPE normalized overhead ratio is quite low due to the better successful delivery percentage.

Simulation results in Figure 5.2 (c) show that SPD in case of existing approaches [52,58] decreases with the increase in data rate because greater number of packets violate the policy. However, in case of EPE, SPD is much better and remains consistent due to the efficient detection of policy change implementation process. In addition, it is also noted that by varying data rate, SPD always remains better in case of EPE as

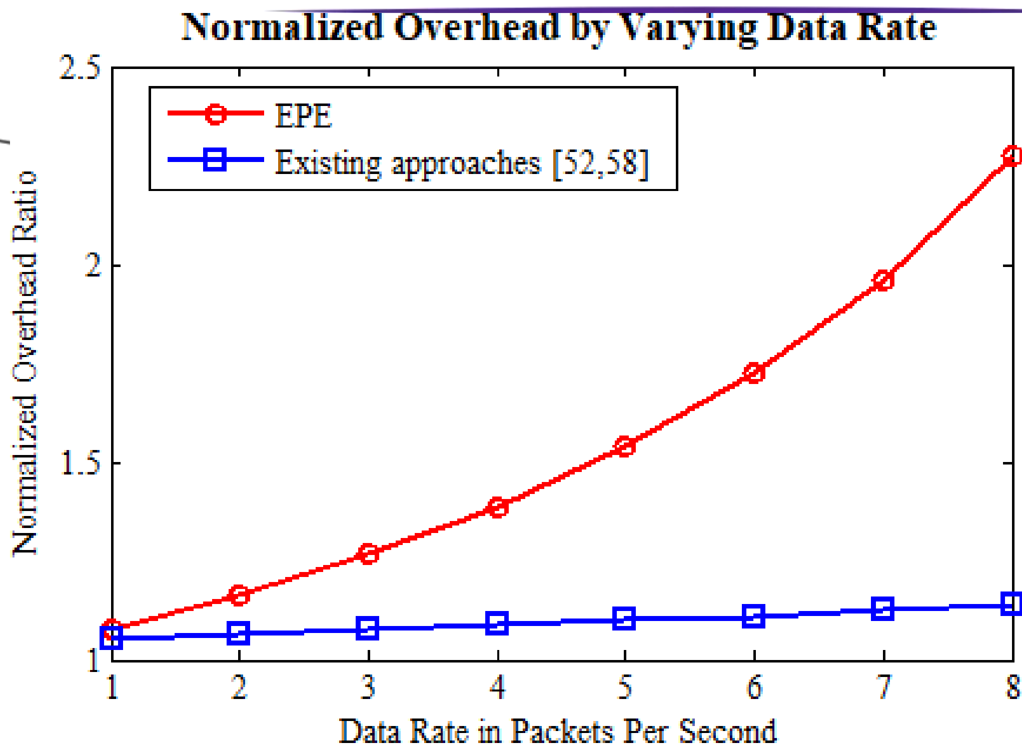
ACADEMIC SOLUTIONS

compared to the existing approaches [52,58]. Similarly, the results in Figure 5.2 (d) show that network throughput is much better as compared to the existing approaches [52,58] by varying data rates.

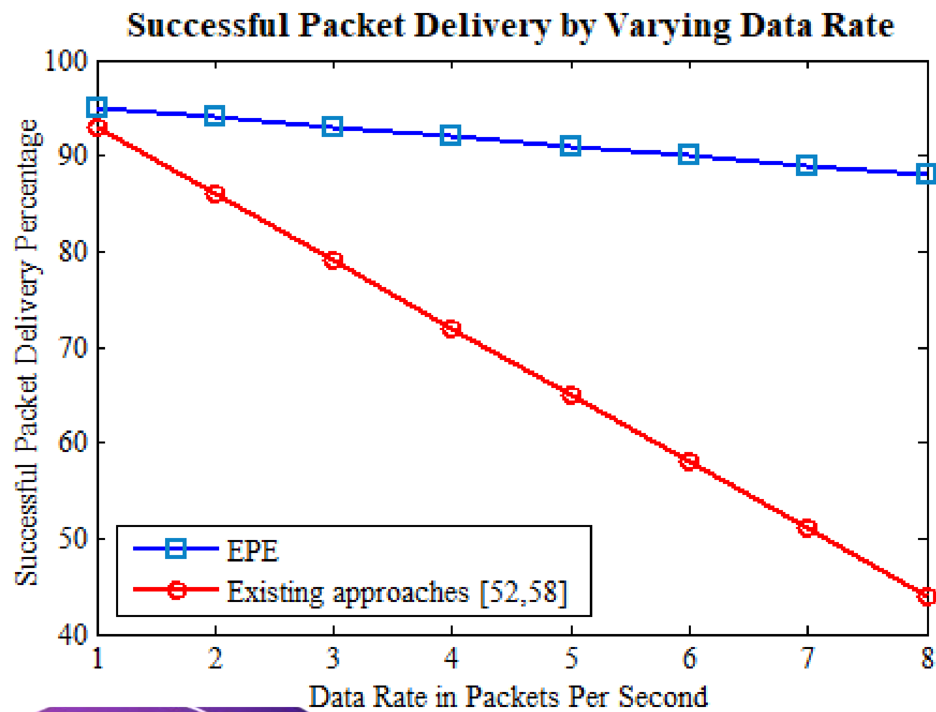


(a) Packet Violation Percentage

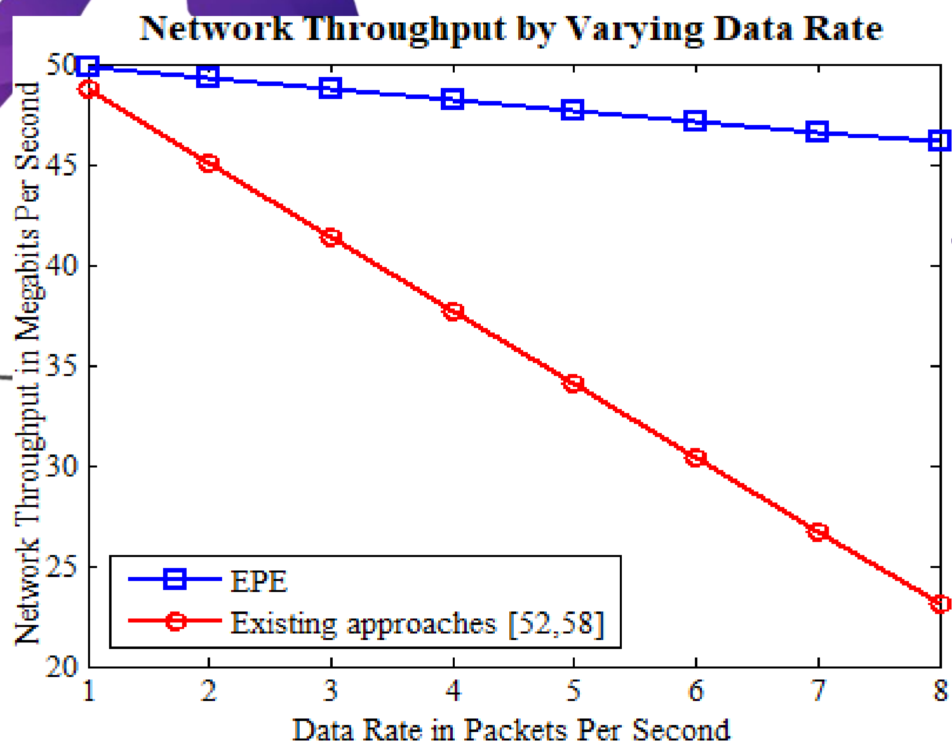
ACADEMIC SOLUTIONS



(b) Normalized overhead



(c) Successful Packet Delivery



(d) Network Throughput

Figure 5.2: Simulation Results by Varying Data Rates

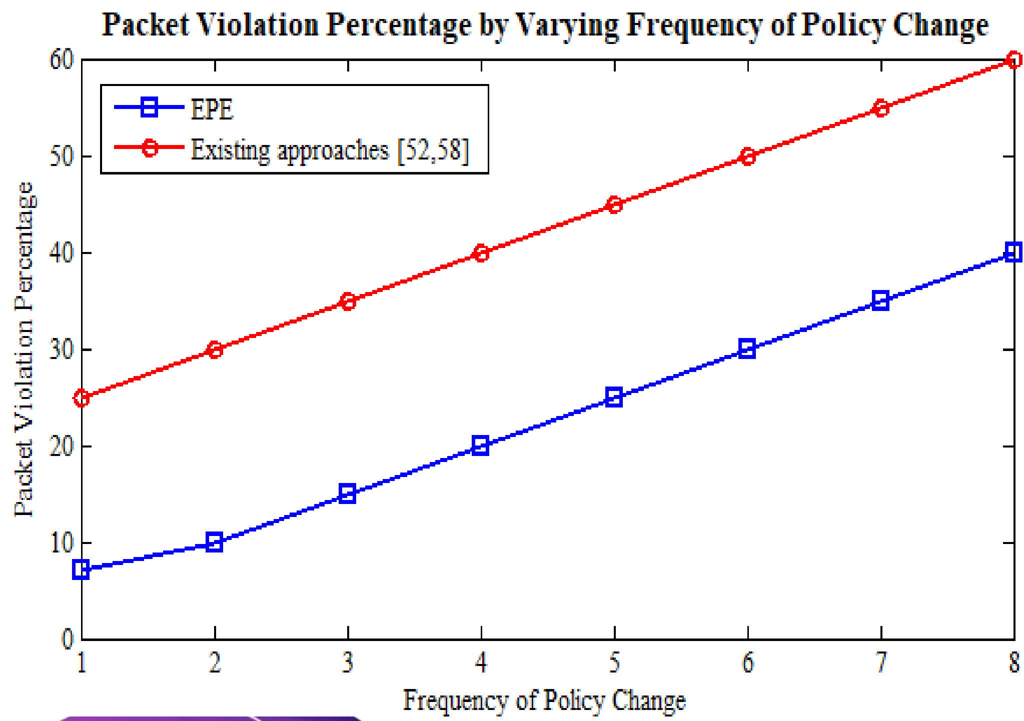
5.2.3 Simulation Results by Varying Frequency of Policy Change

In this simulation, static parameters are set as: hard timeout to 30 seconds, data rate to 4 packets per second, time instance of policy change to 50 seconds and variable parameter is frequency of policy change to 1, 2, 3, 4, 5, 6, 7 and 8.

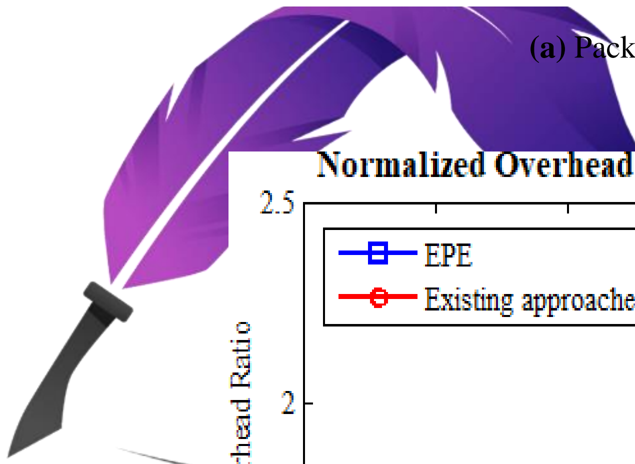
The results in Figure 5.3 (a) show that packet violation percentage increases when the policy changes frequently in both approaches (i.e. in EPE and existing approaches [52,58]). The packet violation percentage of existing approaches [52,58] is higher if policy change occurs at earlier instance of time and is lower in case of policy change at later instance of time. However, in case of EPE there is no impact of earlier or later change of policy. It is also noted that there is linear impact of frequency of policy change in case of EPE but in case of existing approaches [52,58] it is totally dependent upon the first instance of policy change.

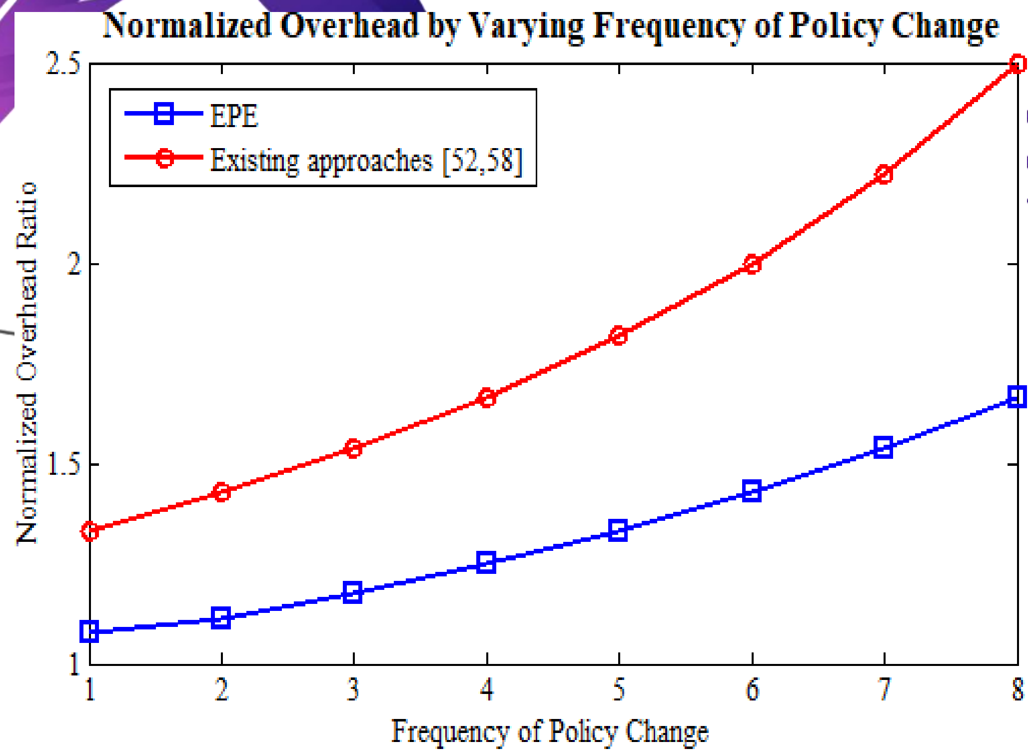
Normalized overhead ratio increases with the increase in frequency of policy change due to the higher packet violation ratio in EPE approach as shown in Figure 5.3 (b). However, in case of existing approaches [52,58] it depends upon time instance of policy change, that is, it is higher at earlier and lower at later time instance of policy change. Simulation results in Figure 5.3 (c) show that SPD percentage in both existing approaches [52,58] and EPE decreases with the increase in frequency of policy change. However, in case of EPE, the SPD percentage remains consistently greater due to the efficient detection of policy change implementation process. Similarly, it is also observed from simulation results that network throughput is much better in case of EPE as compared to existing approaches [52,58] as shown in Figure 5.3 (d). It is due to the fact that less packet violations occur due to the efficient handling of network policy change mechanism which results in increasing SPD percentage and ultimately network throughput increases. It clearly indicates that it greatly enhances network efficiency and correctness.

ACADEMIC SOLUTIONS

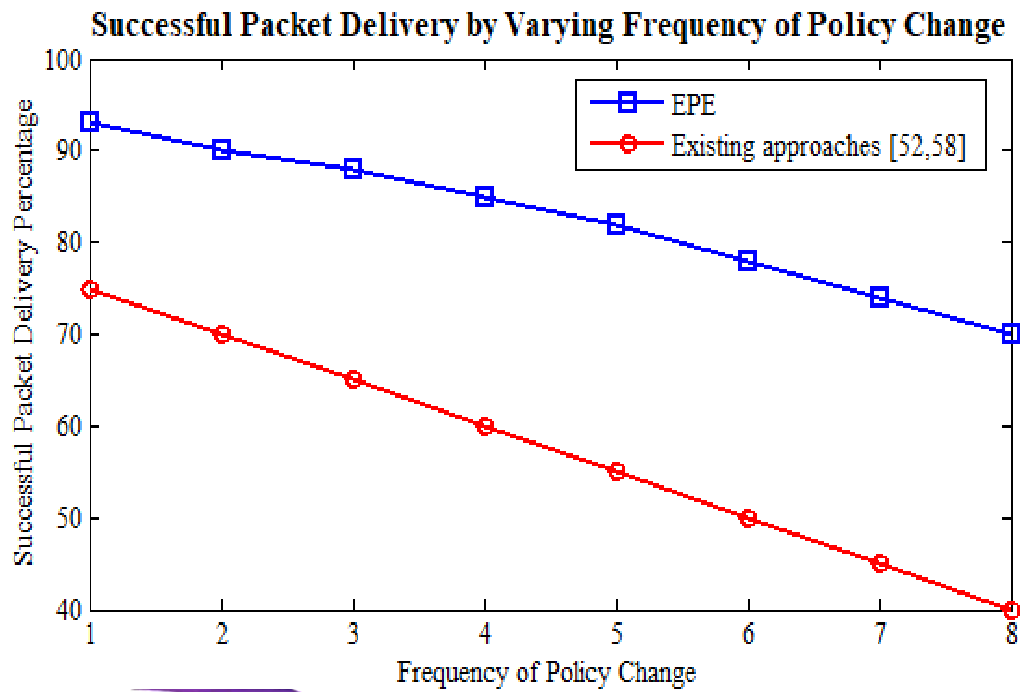


(a) Packet Violation Percentage

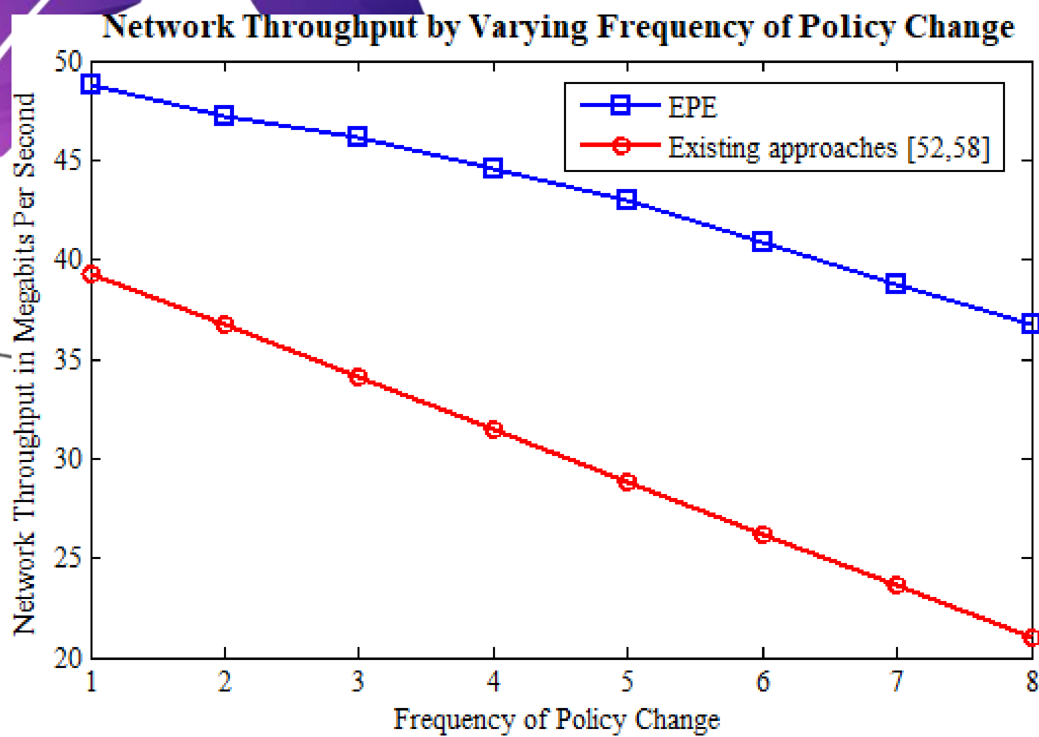




(b) Normalized Overhead



(c) Successful Packet Delivery



(d) Network Throughput

Figure 5.3: Simulation Results by Varying Frequency of Policy Change

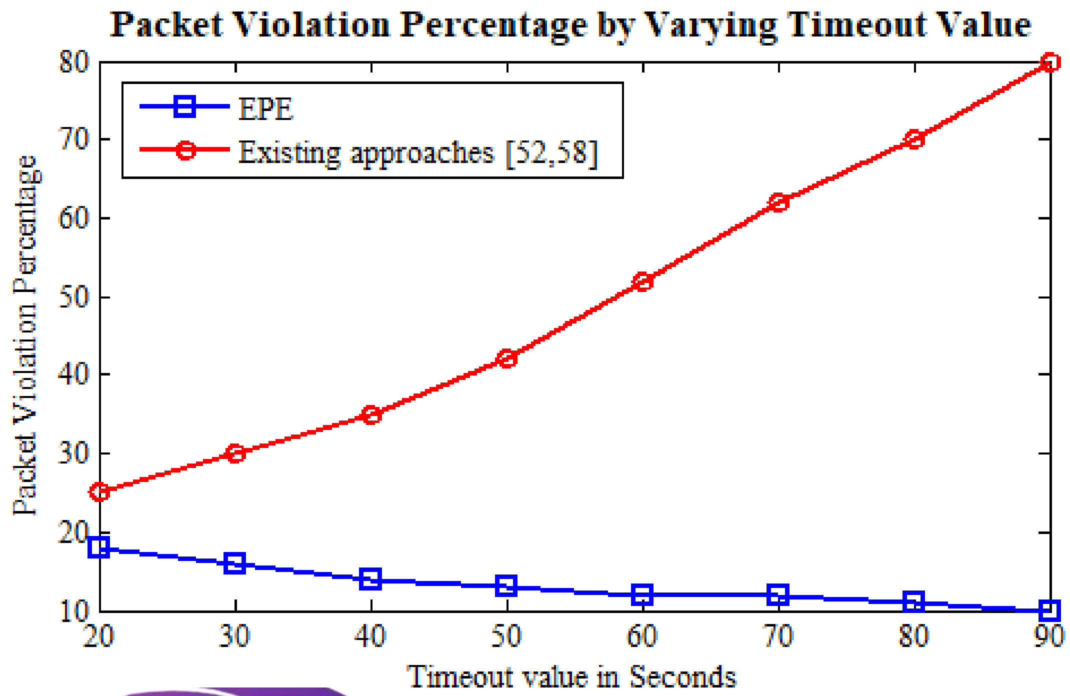
5.2.4 Simulation Results by Varying Timeout Value

The static parameters in this simulation are: time instance of policy change which is set to 50 seconds, frequency of policy change is set to 4, data rate is set to 4 packets per seconds, and dynamic parameter is flow timeout value which is set to 20, 30, 40, 50, 60, 70 and 80 seconds.

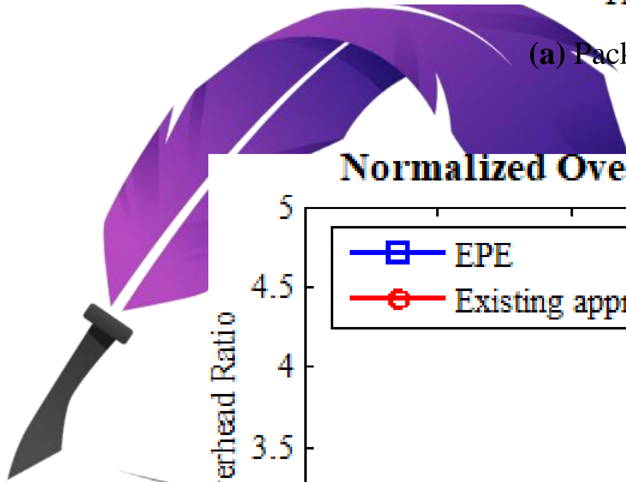
Figure 5.4 (a) demonstrates that simulation results based on varying timeout values which show that packet violation percentage in existing approaches [52,58] increases with the increase of timeout value. This is due to the reason that more packets are delivered to invalid hosts due to already installed flow rules on the switches along the path for the longer period of time. Therefore, with the increase in timeout value packet violations are also increased. However, in case of EPE, packet violation percentage remains consistent. This is because EPE detects policy change and then purges the already installed flow rules which could be affected by the change in policy. Thus, EPE provides better results in the sense that timeout value is set to optimum level to avoid much violations. In addition, it also helps to avoid congestion over the network due to less installation of flow rules at the switches. The normalized overhead increases with the increase in value of timeout in existing approaches [52,58] because more packets violate policy at higher timeout value. However, in case of EPE it does not matter whether timeout value is smaller or greater, that is the normalized overhead remains consistent as shown in Figure 5.3 (b). There are minor effects of normalized overhead due to handling more packet-in messages at lower flow timeout values.

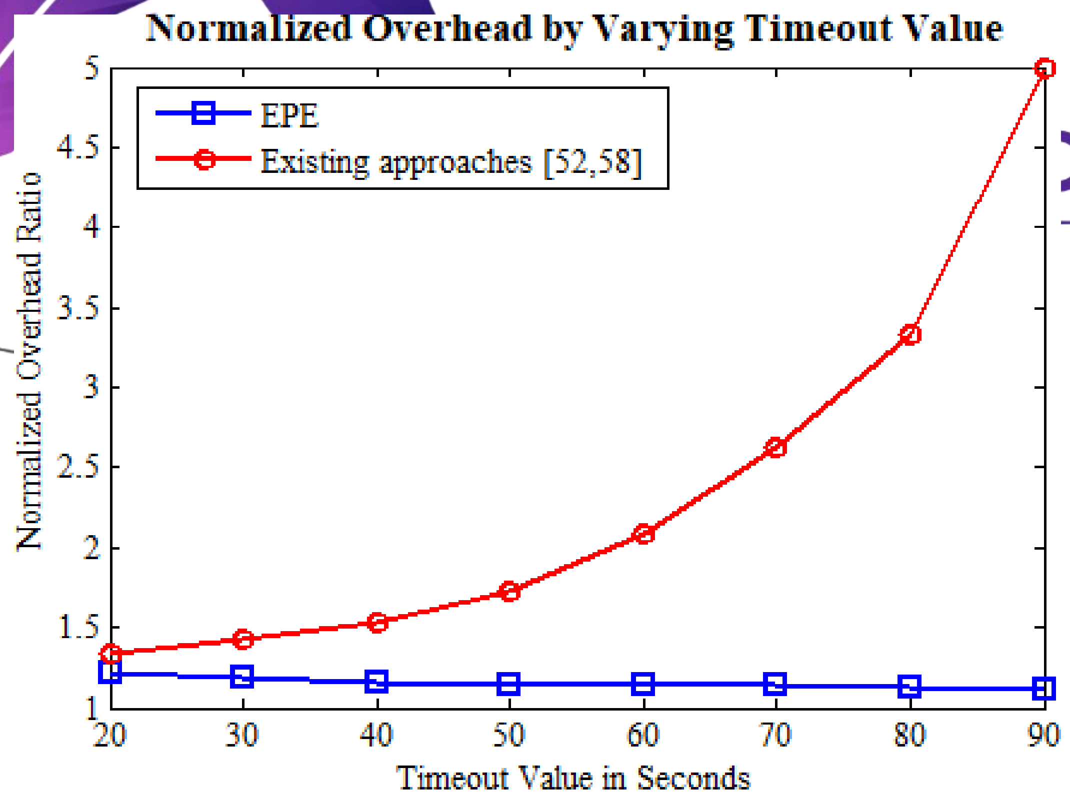
ACADEMIC SOLUTIONS

Simulation results in Figure 5.4 (c) show that SPD increases with the increase in timeout value and remains consistent in case of EPE. However, in case of existing approaches [52,58] SPD continuously decreases with the increase in timeout value due to more packet violations. This is because with the increase in timeout value, the flow rules remain in flow table of switches for more time which results in more violations in case of policy change. Similarly, the simulation results by varying timeout values show that network throughput in case of EPE remains consistent and better as compared to the existing approaches [52,58] as shown in Figure 5.4 (d) due to effective implementation of policy change mechanism.

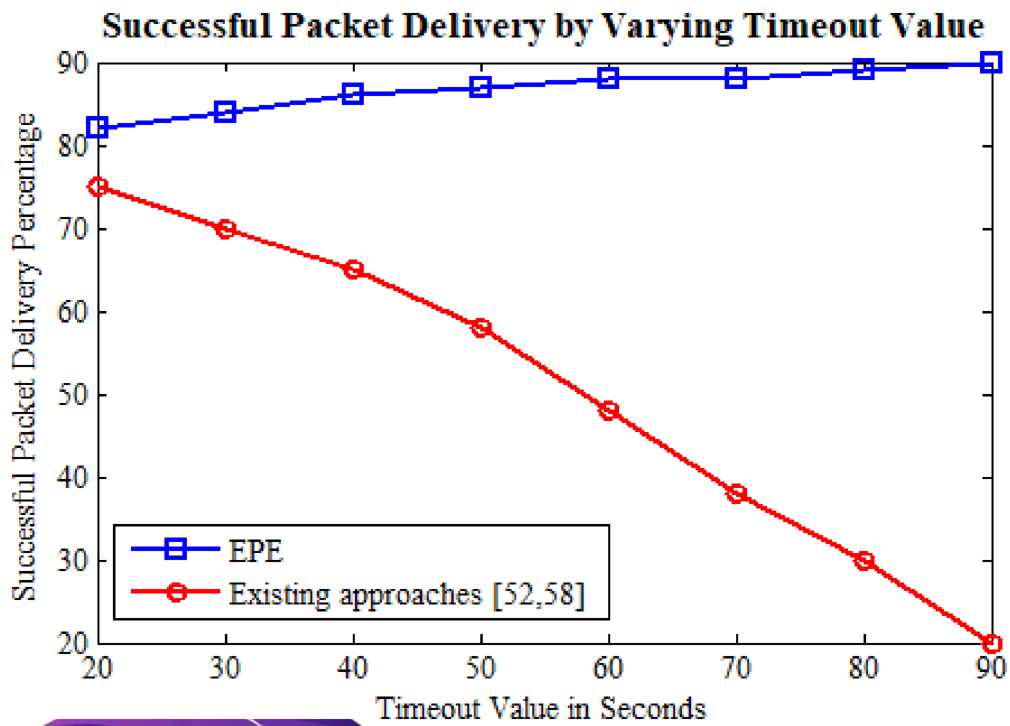


(a) Packet Violation Percentage

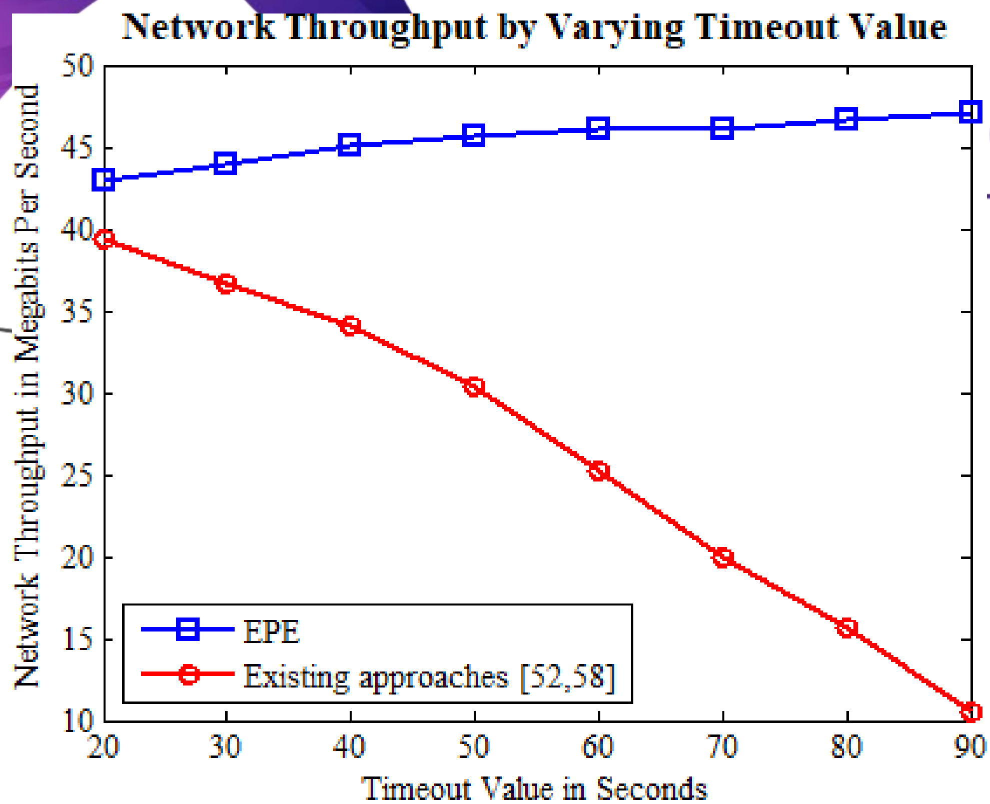




(b) Normalized Overhead



(c) Successful Packet Delivery



(d) Network Throughput

Figure 5.4: Simulation Results Based on Variation in Timeout Value

For experimentation and analysis of proposed solutions, different performance metrics are utilized. The results based on first proposed approach EPE show that the Packet Violation Percentage is reduced up to 99.8%, Successful Packet Delivery Percentage is increased up to 98%, Normalized Overhead Ratio is decreased 88.8% and Network Throughput is increased up to 99.8% as compared to existing approaches [52,58]. The simulation results show that the proposed approach helps to implement network policy change mechanism in an effective way which results in increasing the network performance and efficiency.

5.3 Experimental Results of Graph-Based Proposed Approach

For this research, Mininet [30] SDN Network Emulator and POX SDN controller [31] are chosen for the implementation and analysis of proposed network scenario. This network scenario is setup on HP Probook 450 G5 with Intel Core i5-8250U CPU@ 1.60 Ghz (8 CPUs), 8GB RAM and 1TB Sata HDD, running Linux operating system, Ubuntu 16.04. The proposed application is written in python to be executed on POX controller for the analysis of the proposed solution. For the analysis of this research, network topology is developed in Mininet 2.2 EEL by adding and configuring hosts, links, open-flow soft switches (OVS 2.5.2) which supports open flow 1.3 and higher versions, and POX SDN controller.

ACADEMIC SOLUTIONS

For fair analysis of the proposed solution, network topology of an educational institute is opted. For this purpose, 9 OpenFlow soft switches and 30 hosts are added along with one SDN POX controller. Moreover, 1,000 to 80,000 packets are transmitted randomly from different sources to various destinations based on the network topology and network policies. The total simulation time of experimental evaluation is set to 100 seconds which is selected randomly for the fair analysis of results. The proposed approach 'GPE' is analyzed with 'EPE' approach with the help of following parameters, by varying frequency of policy change, packet transmission rate and timeout value.

a) **Policy Change Detection Time**

Policy Change Detection Time (PCDT) is the time taken by the controller to detect change in network policies and to delete the flow rules from the data plane that

conflict with new policy. This is because the proposed approach attempts to reduce the computation time of policy change detection.

b) Packet Violation Percentage

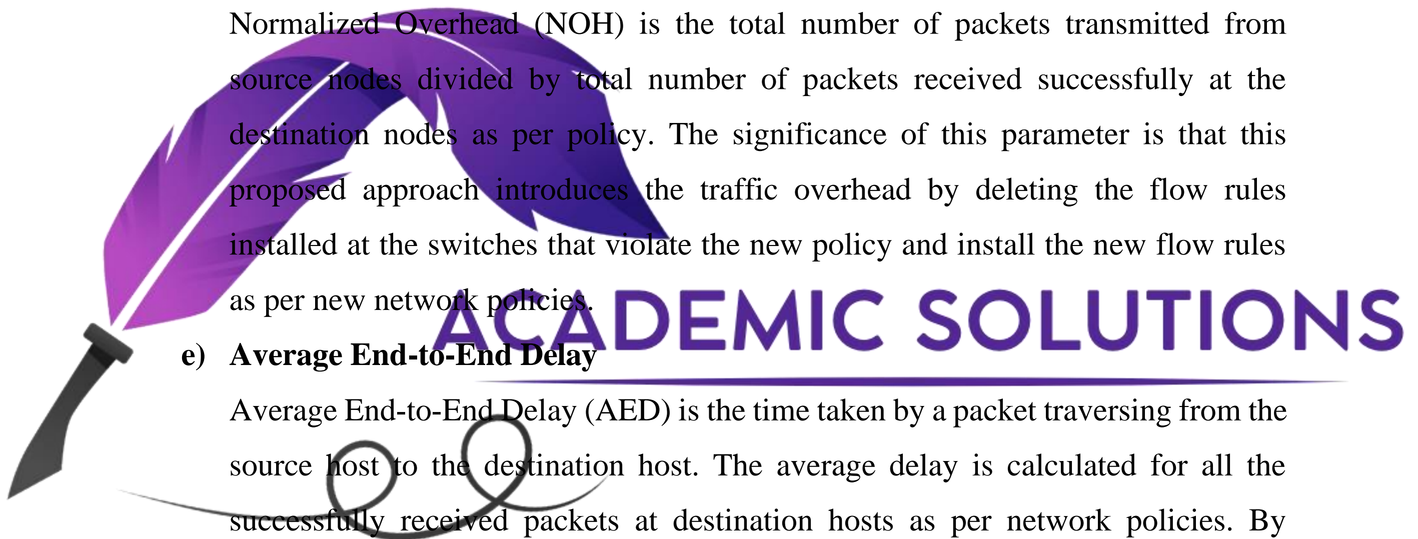
Packet Violation Percentage (PVP) is the percentage of total number of packets that violate the policy to the total number of packets initiated from source nodes. This parameter will show that how the reduced computation time for policy change detection significantly impacts the number of packets violating the policies.

c) Successful Packet Delivery

Successful Packet Delivery (SPD) is the percentage of total number of packets that are delivered as per policy to the total number of packets initiated from source nodes.

d) Normalized Overhead

Normalized Overhead (NOH) is the total number of packets transmitted from source nodes divided by total number of packets received successfully at the destination nodes as per policy. The significance of this parameter is that this proposed approach introduces the traffic overhead by deleting the flow rules installed at the switches that violate the new policy and install the new flow rules as per new network policies.


ACADEMIC SOLUTIONS

e) Average End-to-End Delay

Average End-to-End Delay (AED) is the time taken by a packet traversing from the source host to the destination host. The average delay is calculated for all the successfully received packets at destination hosts as per network policies. By presenting the policies in multi-attributed graph and detecting the policy change through graph matching, the significance improvement of proposed approach is shown by analyzing the average end-to-end delay.

f) Average Verification Time

Average Verification Time (AVT) is the time taken by the controller to interpret and detect change in network policies.

g) Network Throughput

Network Throughput (NTP) is the number of packets per second delivered successfully to the destination as per policy.

5.3.1 Simulation Results by Varying Frequency of Policy Change

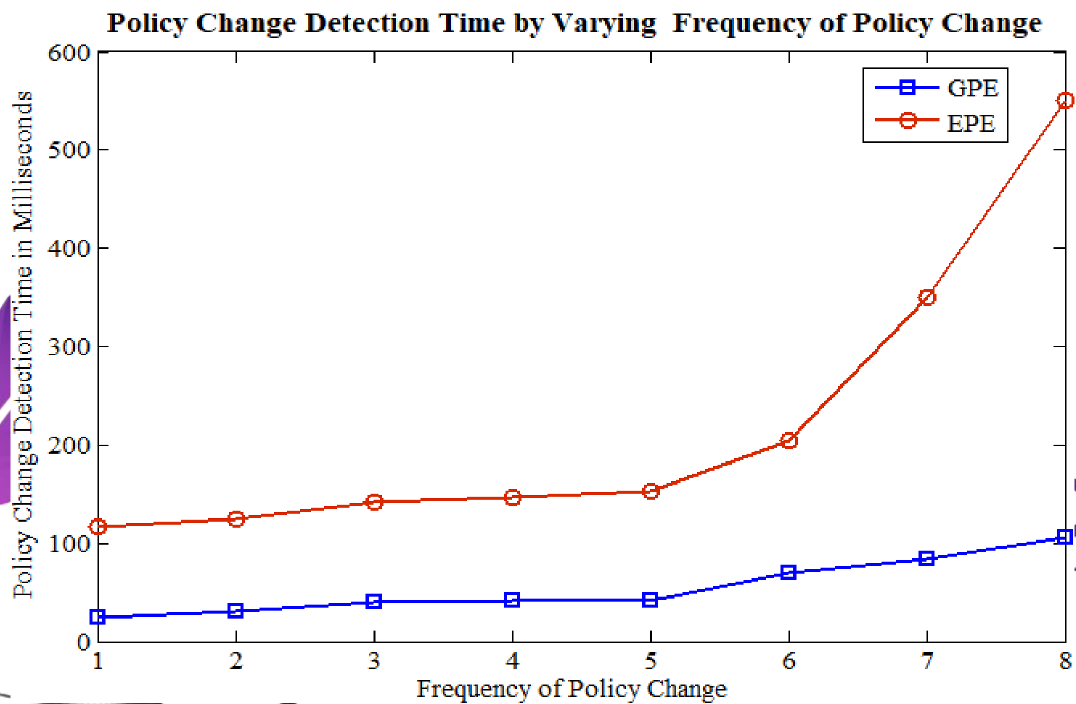
The simulation results are based on static and dynamic parameters. The static parameters include packet transmission rate that is set to 0.6 milliseconds (ms) and default time-out value of flow rules are set. The dynamic parameter is chosen as frequency of policy change, that is chosen as 1, 2, 3, 4, 5, 6, 7 and 8 policies, which changes randomly. Based on the above parameters, simulation was run, and results are shown in Figure 5.5.

The simulation results by varying frequency of policy change are shown in Figure 5.5 (a) which indicate that PCDT in case of EPE is always greater than GPE, especially at higher frequency of policy change. It is also observed that in EPE, PCDT increases abruptly, while in GPE, it increases gradually at higher rate of policy change. It is due to that GPE detects and implements policy change mechanism earlier than EPE. Figure 5.5 (b) shows that PVP in case of GPE is always less than its competitor EPE due to the smaller number of packet violations during policy change implementation process. The results indicate that GPE performs well at all frequencies of policy change and are much better at its higher rates. It reflects that GPE can be more beneficial in data center and campus networks where policy change frequencies are quite higher and frequent.

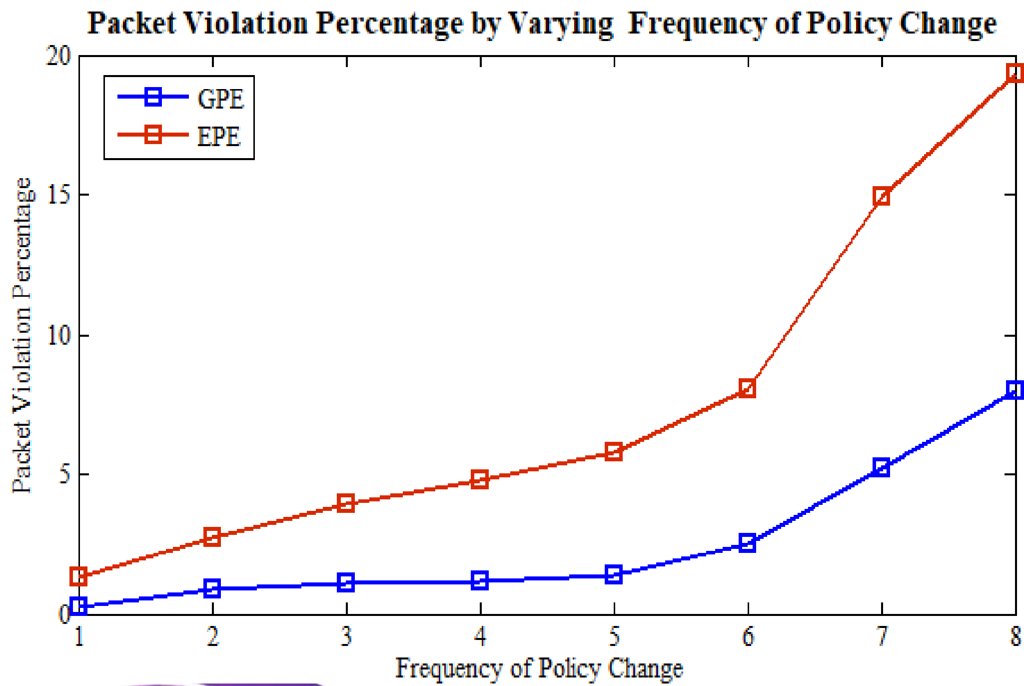
ACADEMIC SOLUTIONS

Simulation results in Figure 5.5 (c) show that GPE always provides better performance with respect to SPD percentage in the network. However, its competitor EPE provides lower SPD percentage at all frequencies of policy change. This is due to that the detection of policy change mechanism is quite efficient in case of GPE and accordingly old flow rules are deleted and new flow rules are computed/installed at data plane in a very effective manner. So, GPE can be implemented in a network where higher reliable communication is desired. Figure 5.5 (d) shows that NOH is increasing with the increase in policy change frequency in both approaches. However, the GPE provides lower NOH throughout the simulation due to higher SPD percentage. Figure 5.5 (e) reflects that AED in case of GPE is always less than EPE. This is due to the fast detection of policy change with the help of multi-attributed graphs. Simulation results show that GPE performs well at all frequencies of policy change. However, AED at higher frequencies of policy change is higher due to the greater policy change detection time of multiple policies in addition to installation and deletion of more flow rules.

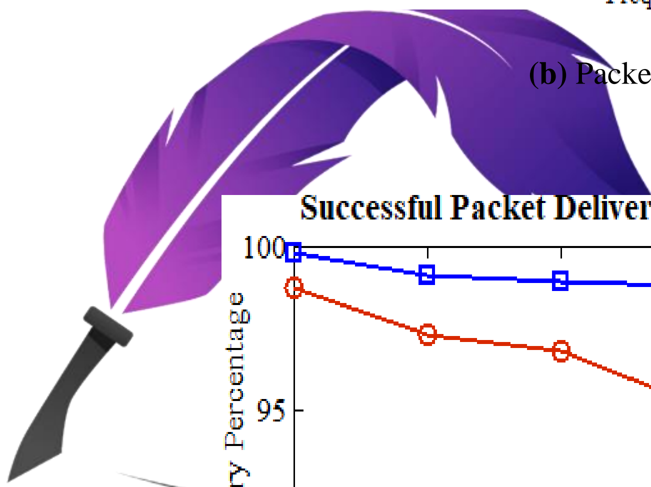
Figure 5.5 (f) represents that AVT increases by varying greater number of policies due to the verification of greater number of policies. However, it is clear that GPE detects and verifies policy change in less amount of time as compared to its competitor EPE. Simulation results based on varying frequency of policy change as shown in Figure 5.5 (g) indicate that GPE provides much better network throughput as compared to EPE. This is due to the better handling of network policy change via graph-based abstractions and fast detection of policy change via multi-attributed graphs.

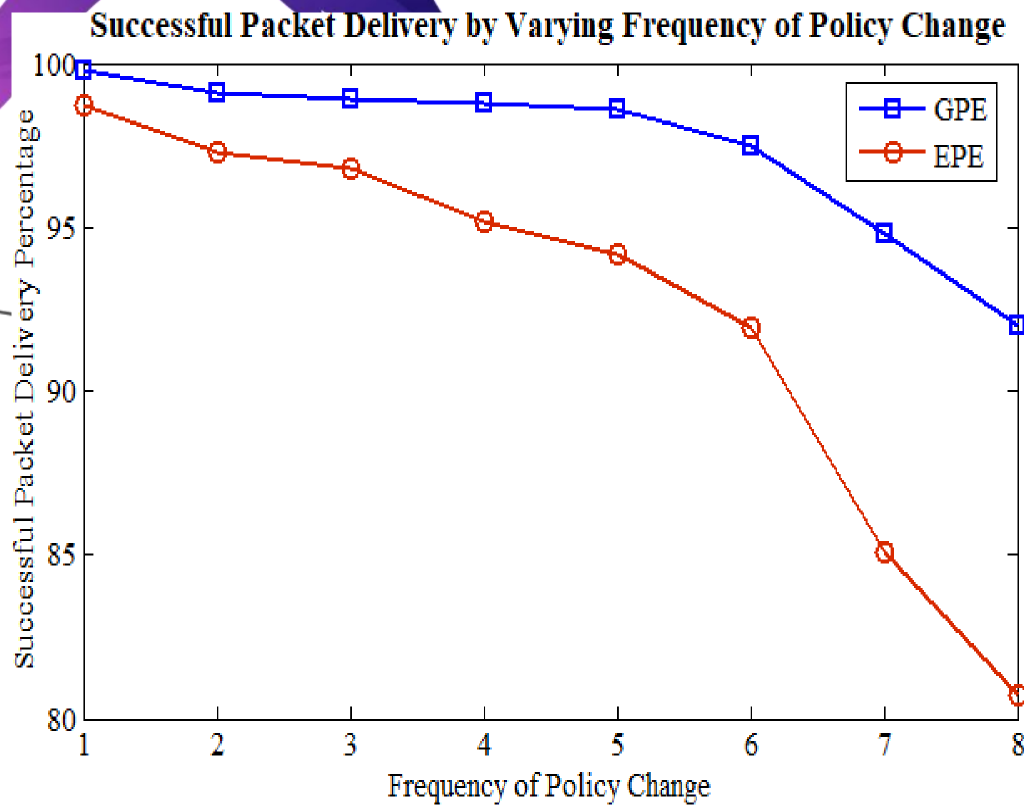


(a) Policy Change Detection Time

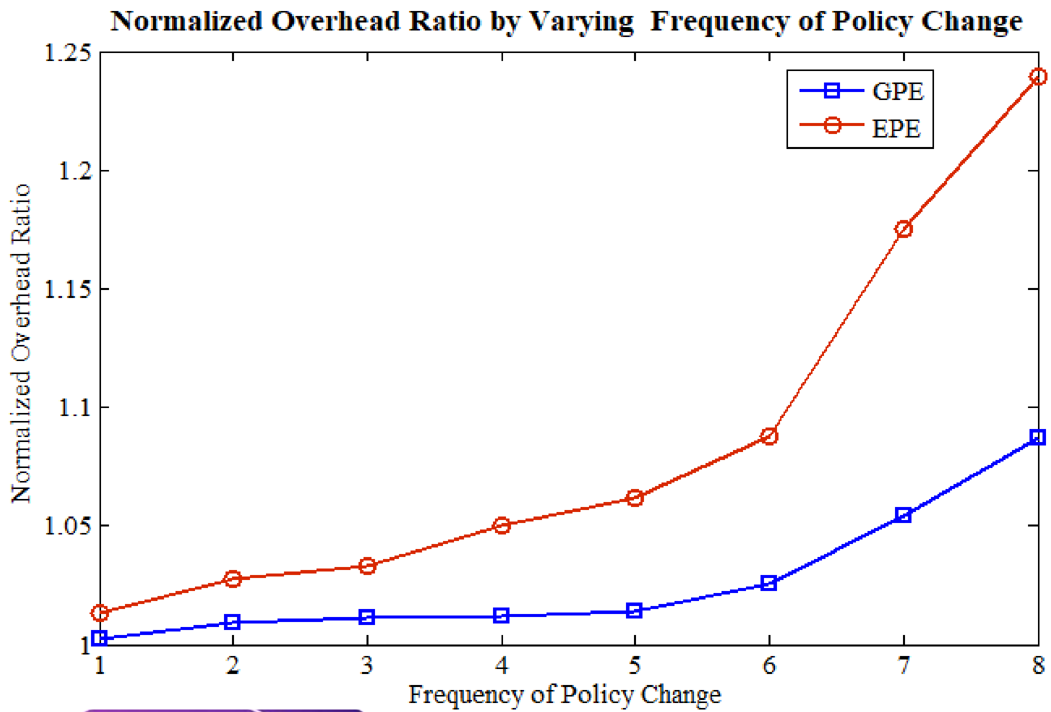


(b) Packet Violation Percentage

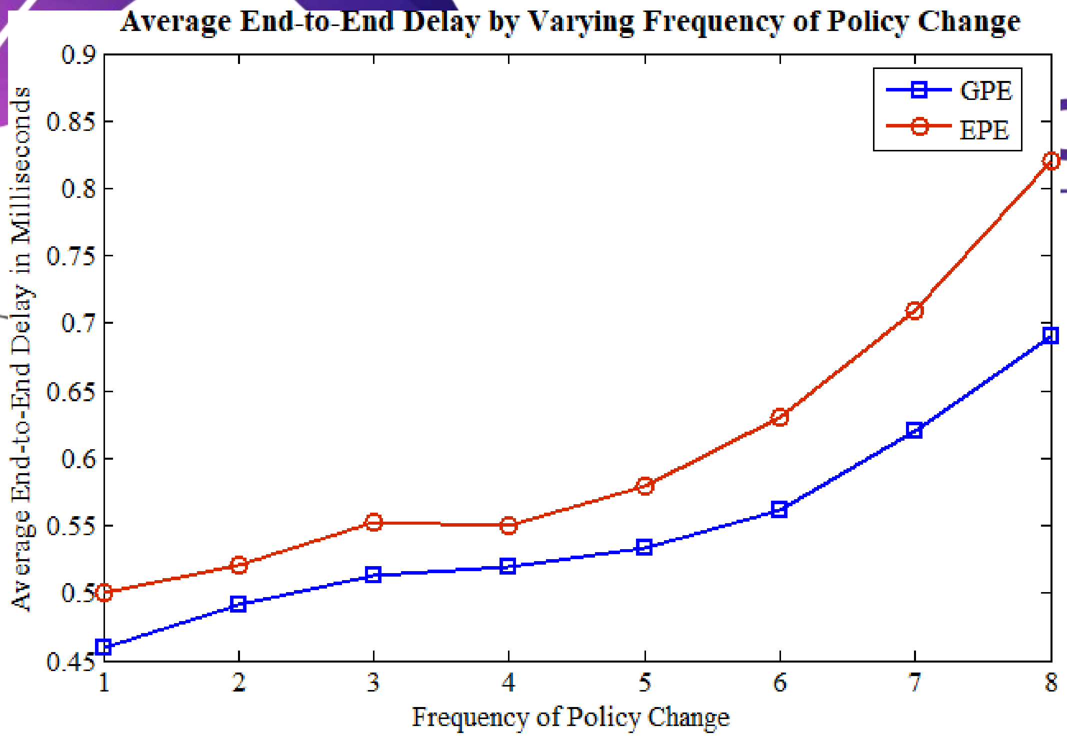




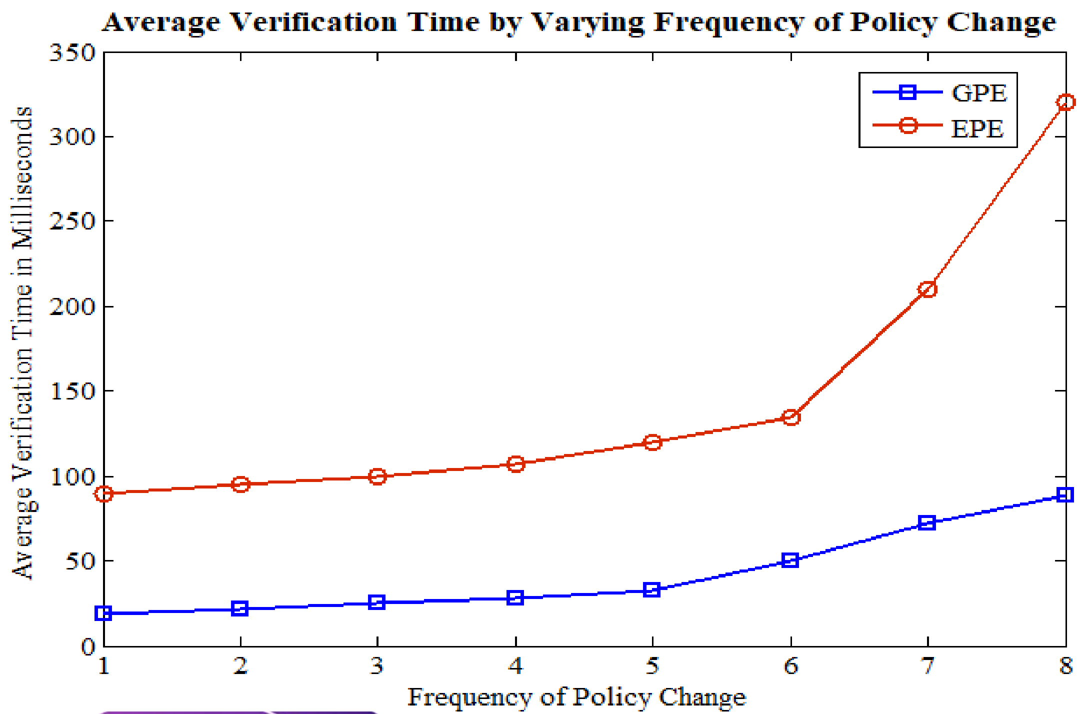
(c) Successful Packet Delivery



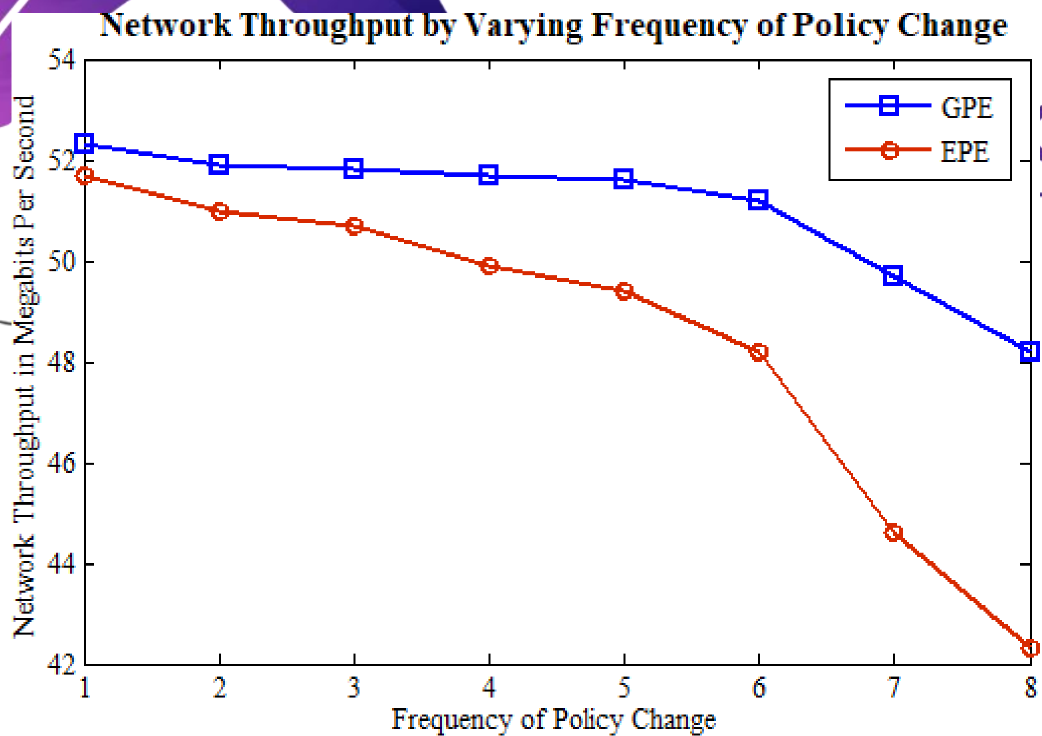
(d) Normalized Overhead



(e) Average End-to-End Delay



(f) Average Verification Time



(g) Network Throughput

Figure 5.5: Simulation Results by Varying Frequency of Policy Change

5.3.2 Simulation Results by Varying Packet Transmission Rate

To see the results by varying the packet transmission rate, the static parameters include; the frequency of policy change is set to 5 and default timeout values of flow rule on data plane are set. The dynamic parameter is chosen as packet transmission rate that is set to 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7 and 0.8 ms randomly. Based on the above parameters, simulation was run, and results are shown in Figure 5.6.

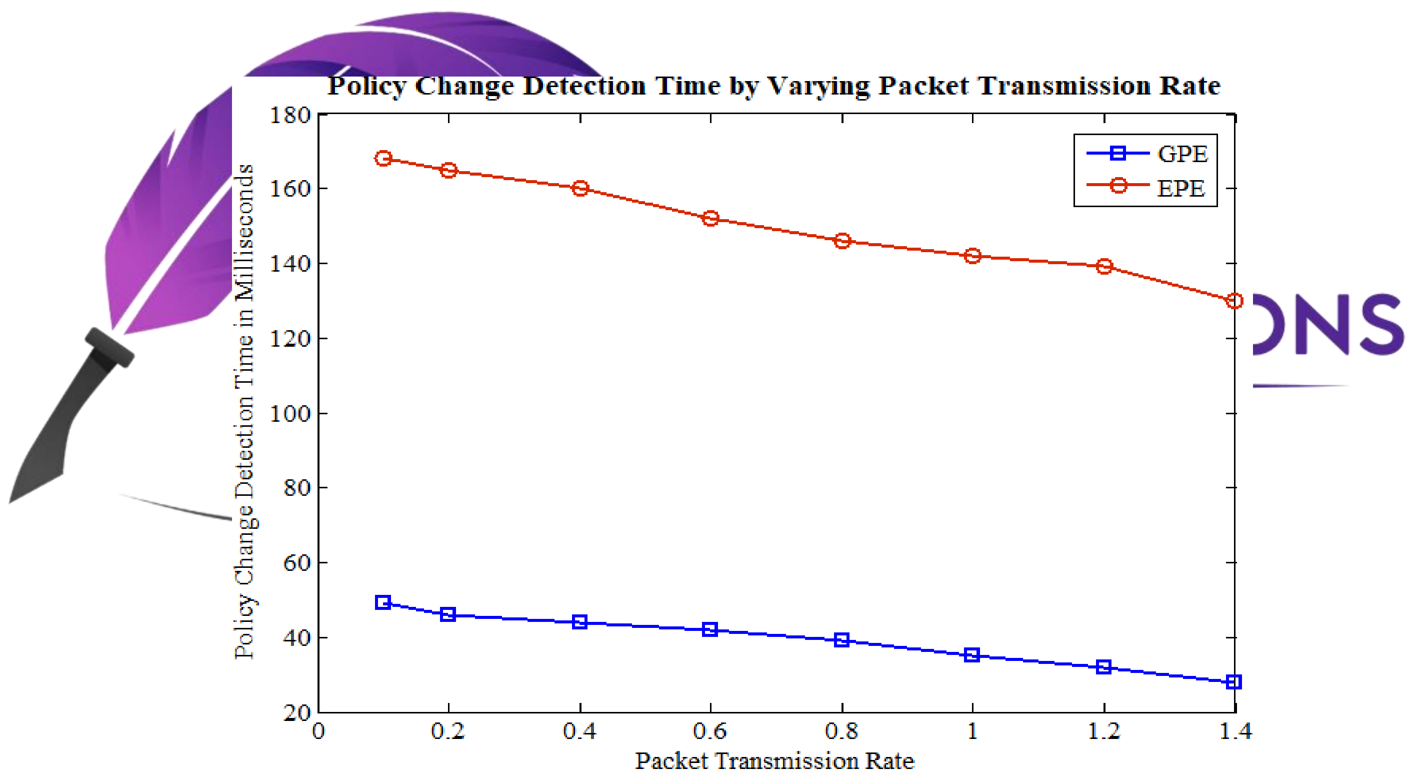
Figure 5.6 (a) shows simulation results with respect to PCDT by varying packet transmission rate. The results indicate that by decreasing packet transmission rate of source hosts, PCDT is much lower in case of GPE as compared to EPE. This is due to the fact that GPE detects policy change more efficiently as compared to EPE. Moreover, by decreasing the packet transmission rate, PCDT decreases in both approaches. This is because by decreasing the packet transmission rate, a smaller number of flow requests arrives at the controller, thus, the controller has more free time to detect policy change. Figure 5.6 (b) shows PVP increases with the decrease in packet transmission rate in both approaches. The reason is that the chances of flow rules expiry increase, hence, more new flow rules arrive at the controller, and subsequently the controller computes the flow rules as per new network policies. However, it is quite clear that with this decrease in packet transmission rate, the GPE has a smaller number of packets that violate the network policies as compared to EPE. The logic for this is that GPE takes shorter time to detect the policy change as compared to EPE.

ACADEMIC SOLUTIONS

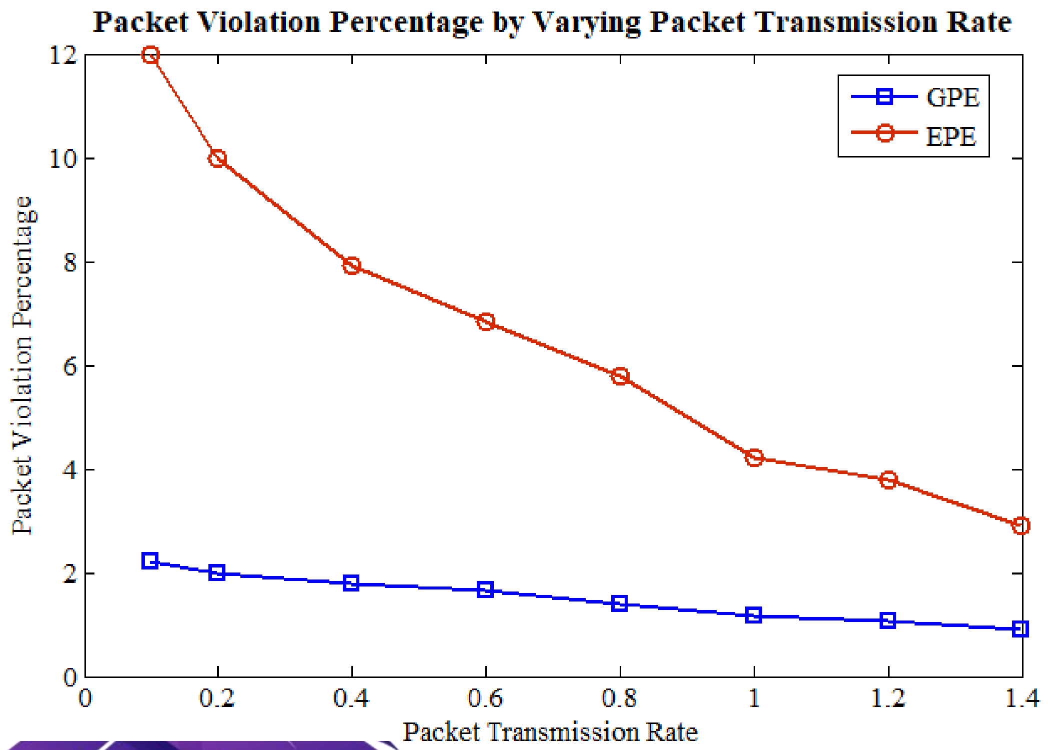
Figure 5.6 (c) shows SPD percentage is consistently better in case of GPE as compared to EPE by varying packet transmission rate. This is due to the reason that GPE detects and implements the network policy change mechanism early. This in turn results in larger value of SPD. It is also noted that GPE is quite consistent in case of high data rate, however, EPE is not consistent and more packet violations occur in case of high data rate. Figure 5.6 (d) shows that NOH is decreasing with the increase in packet transmission rate. However, GPE provides lower NOH due to the less percentage of packet violation. Figure 5.6 (e) shows lower AED in case of GPE, due to its lesser policy change detection time as compared to EPE by varying packet transmission rate. The reason is that flow rules computed as per changed policy along with their

installation at data plane and deletion of old rules as per old policy takes less time in case of GPE as compared EPE.

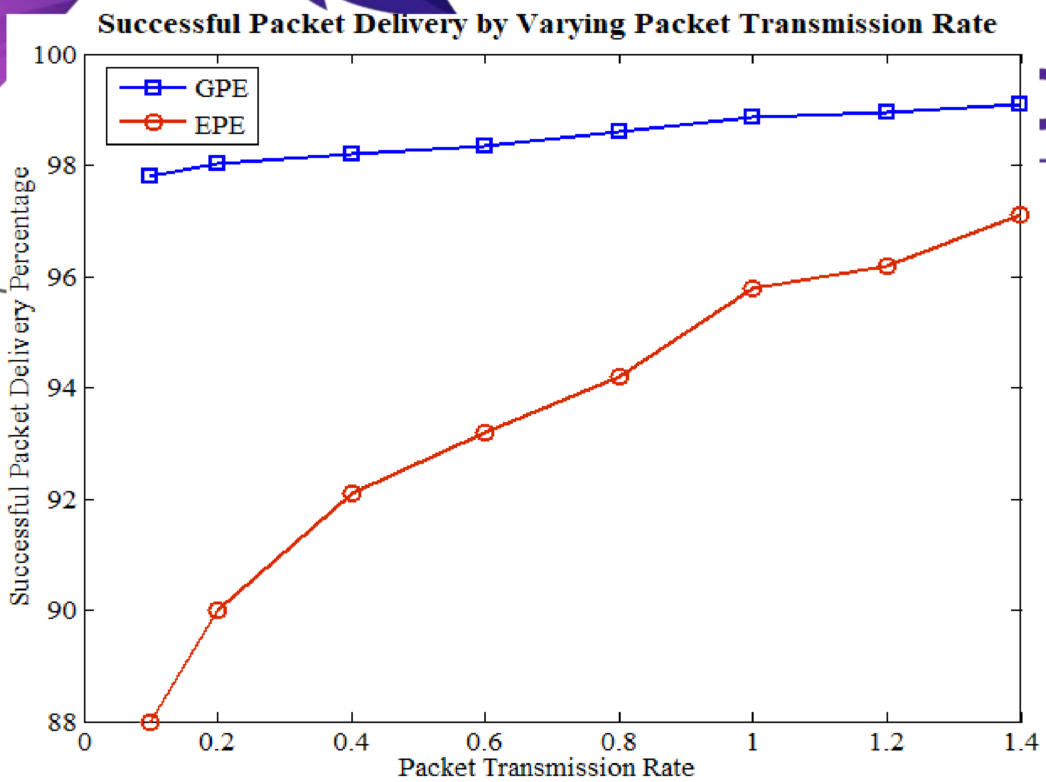
Figure 5.6 (f) represents that AVT in case of GPE is relatively lower due to its graph-based policy verification and detection as compared to EPE. The results indicate that, varying packet transmission rate between source and destination also affect AVT. However, it is quite clear from the results that GPE provides less AVT as compared to EPE. The simulation results as shown in Figure 5.6 (g) indicate that NTP remains better by varying packet transmission rate in case of GPE as compared to EPE which is due to the fact that smaller number of packets delivered to invalid interfaces during policy change process. It shows that GPE is quite effective in communication networks where frequent change in network policies occur.



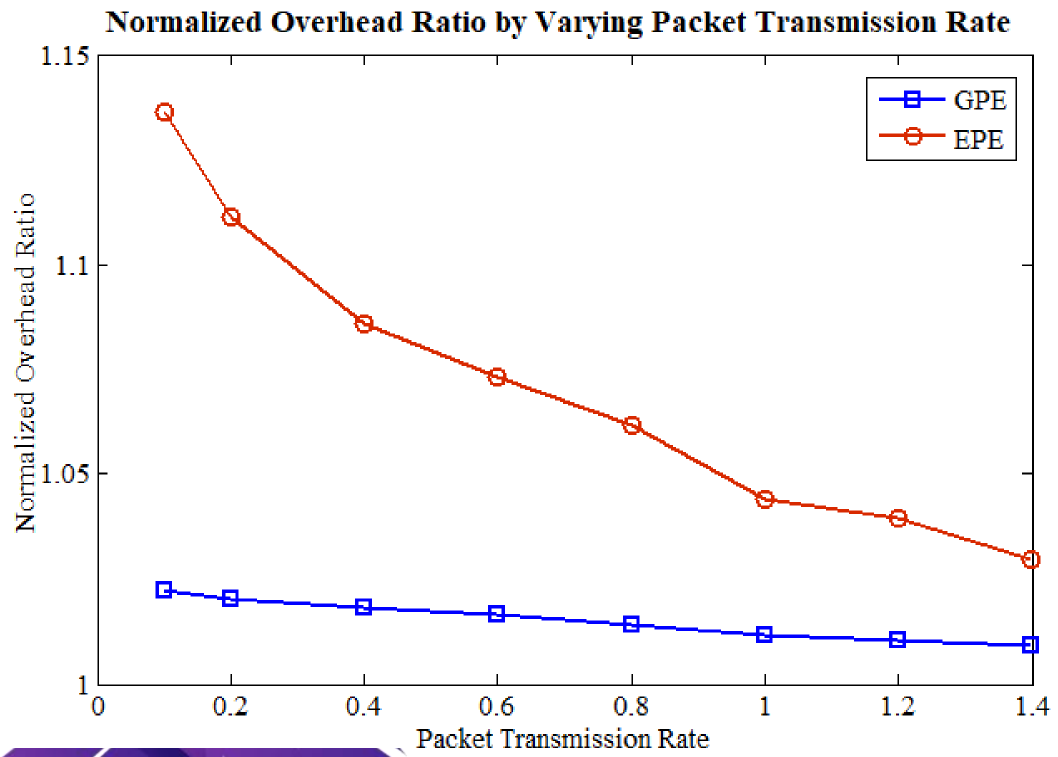
(a) Policy Change Detection Time



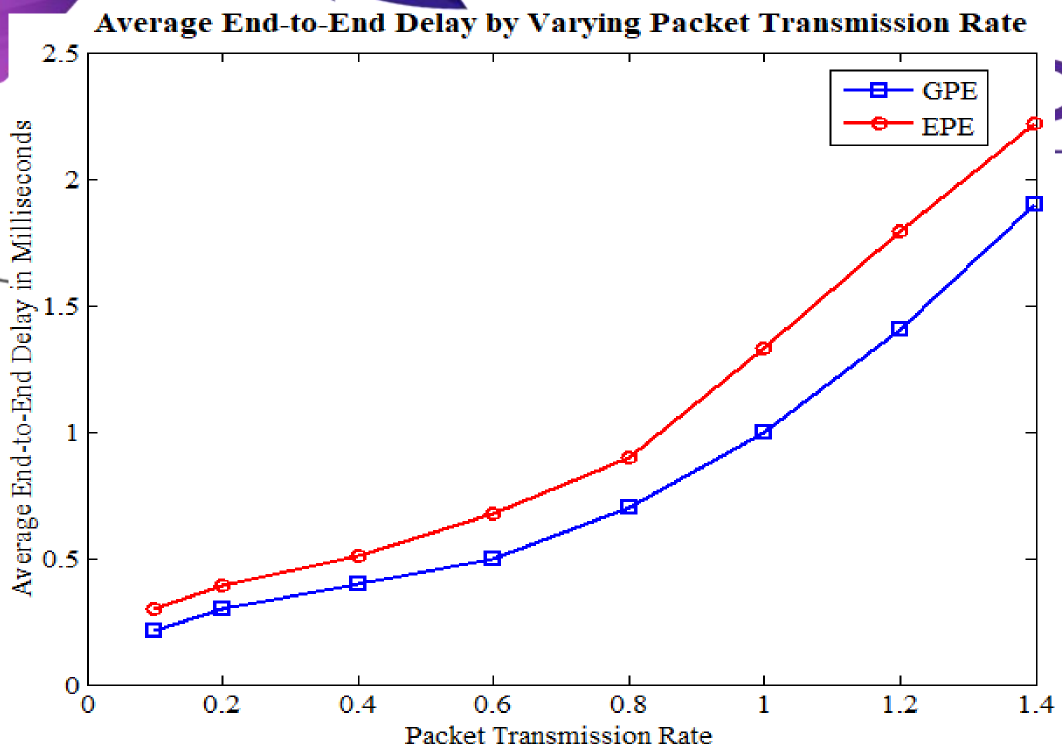
(b) Packet Violation Percentage



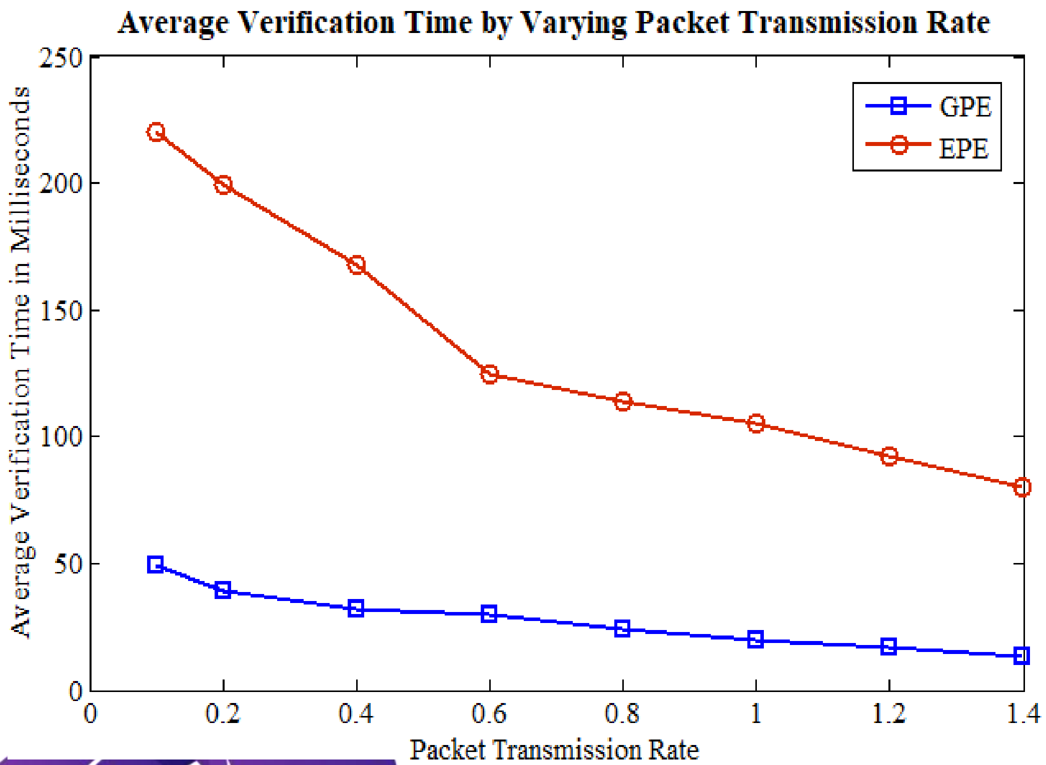
(c) Successful Packet Delivery



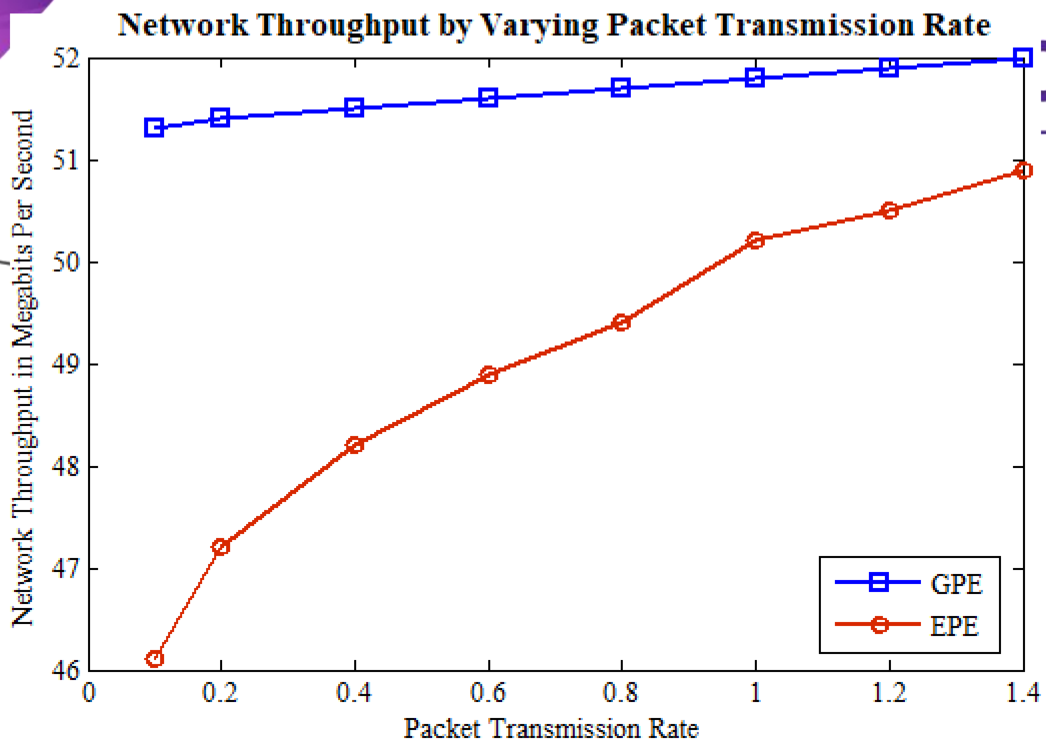
(d) Normalized Overhead



(e) Average End-to-End Delay



(f) Average Verification Time



(g) Network Throughput

Figure 5.6: Simulation Results by Varying Packet Transmission Rate

5.3.3 Simulation Results by Varying Timeout Value

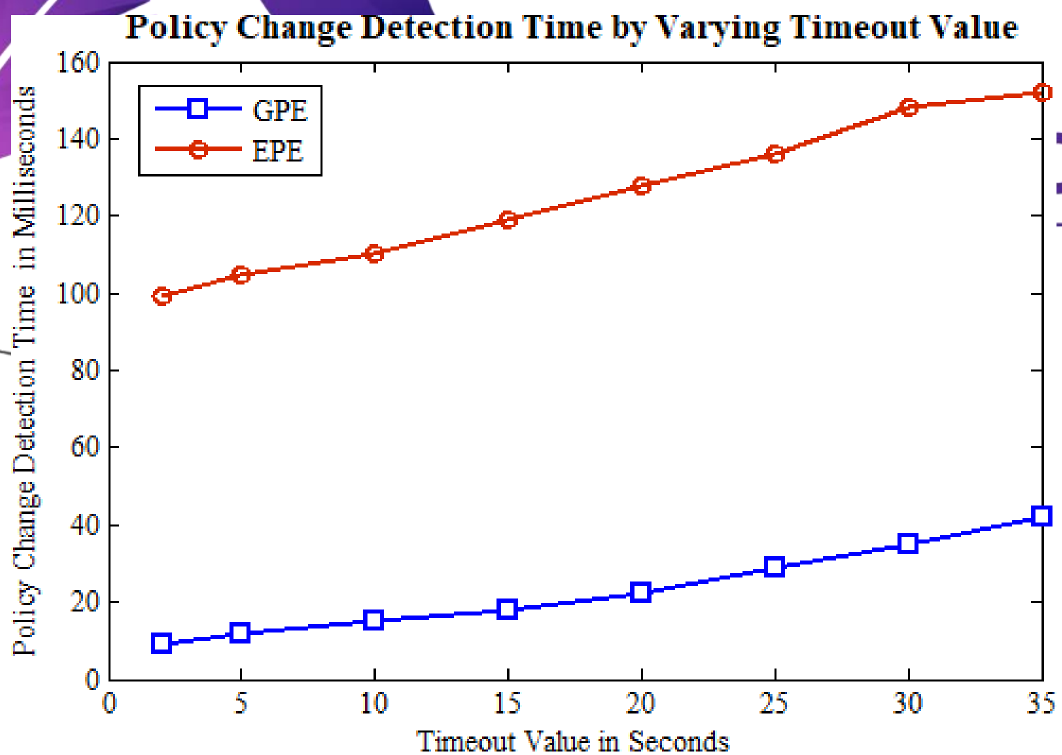
To see the results by varying timeout value of flow rule, the static parameters include; the frequency of policy change is set to 5, packet transmission rate is set to 0.6 ms and dynamic parameter is chosen as flow rule timeout value which is set to 2, 5, 10, 15, 20, 25, 30, and 35 seconds randomly. Based on the above parameters, simulation was run, and results are shown in Figure 5.7.

Figure 5.7 (a) shows that PCDT is quite less at all timeout values in GPE as compared to EPE. It is due to the graph based implementation mechanism of GPE which takes less amount of time during the policy change detection process as compared to EPE which takes larger time during comparing of different matrices. It is also noted that PCDT is increasing with the decrease in timeout values due to the large number of flow rules requests at controller. It is due to that the flow rules expire early, and packet-in messages increase which increase overhead at controller to install flow rules. Figure 5.7(b) shows that PVP is increasing with the increase in flow rule timeout value. This is because by increasing timeout value, the flow rules remain in flow table of switches by more time which results in more packet violations in case of policy change. In addition, the controller has to work more for the handling of packet-in messages which are substantial due to less timeout value. Moreover, the results also reveal that PVP in case of GPE always remains low as compared to EPE throughout the simulation process.

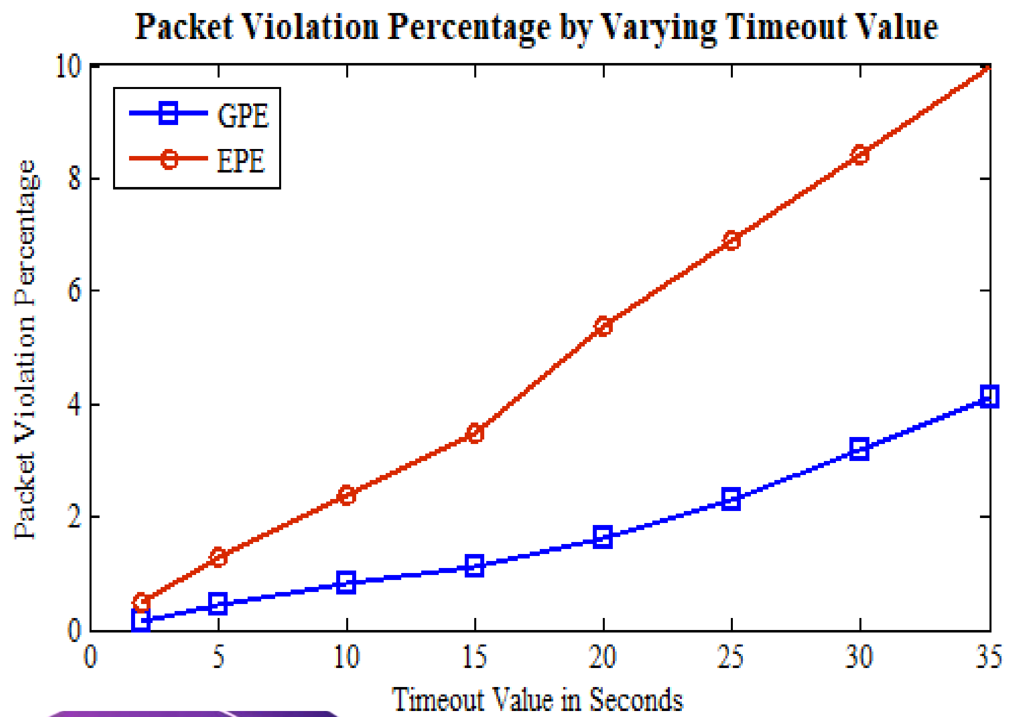
ACADEMIC SOLUTIONS

Figure 5.7 (c) shows that SPD percentage is consistently better in case of GPE as compared to the EPE by varying timeout value. The reason for better SPD percentage in case of GPE is that it reduces packet violations which are substantial in case of EPE due to the late detection of policy change. In addition, there is effect of varying timeout value on SPD percentage which increases with the decrease in timeout value due to the lesser flow rule time in flow table of switches and vice versa. Figure 5.7 (d) shows that NOH ratio decreases with the decrease in timeout value and vice versa. In addition, it is noted that GPE provides lower NOH ratio due to the less percentage of packet violations at all timeout values as compared EPE.

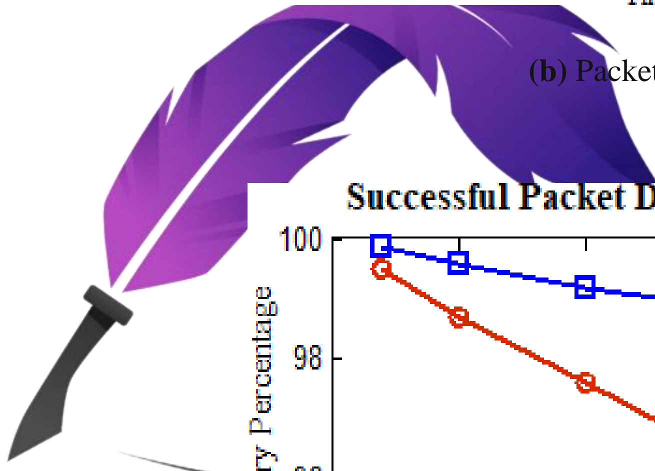
Figure 5.7 (e) shows lower AED in case of GPE as compared to EPE at all timeout values. Although AED increases with the decrease in timeout value, however, it is noted that it remains low in case of GPE as compared to its competitor EPE. This is due to the lower overhead at controller in case of high timeout values, because the controller has to install, delete and verify fewer number of flow rules as per network policies. Figure 5.7 (f) represents that AVT in case of GPE is much lower than EPE due to its efficient policy change detection and verification mechanism. The simulation results show that flow rule timeout value affects AVT as the controller has to implement and process more Packet-In messages from end hosts in case of less timeout values of flow rules due to the expiry of flow rules in flow tables of switches. However, from simulation results, it is quite clear that GPE provides less AVT as compared to EPE at all flow timeout values. Simulation results in Figure 5.7 (g) show that GPE provides better NTP as compared to EPE by varying flow timeout value due to the better SPD percentage and less packet violation percentage.

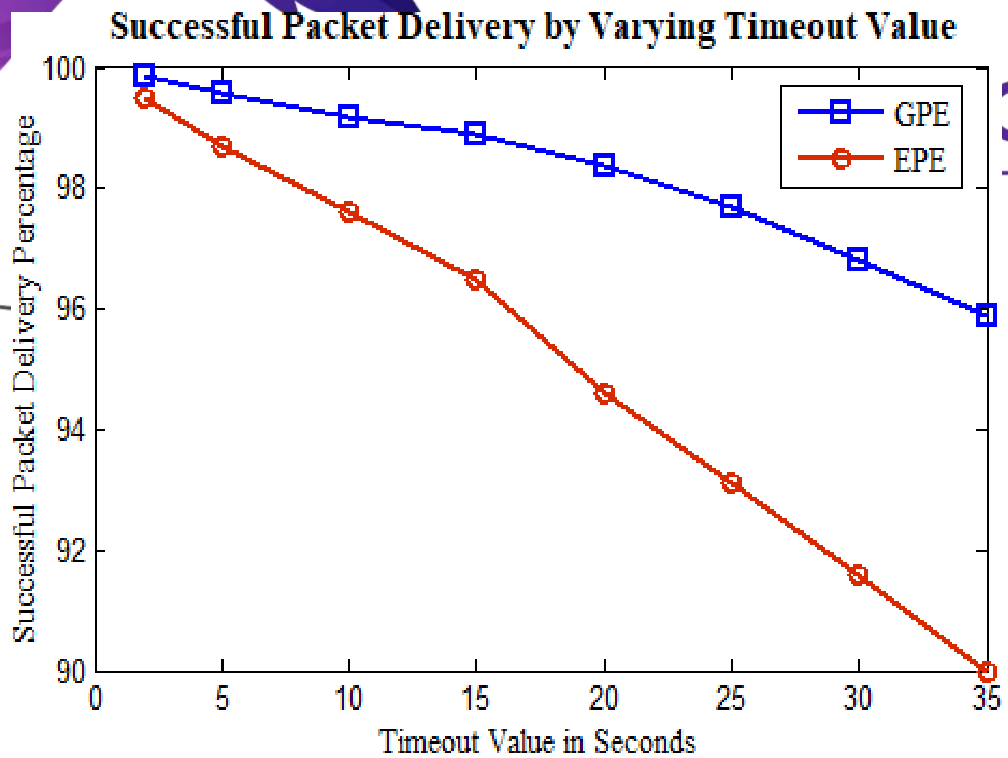


(a) Policy Change Detection Time

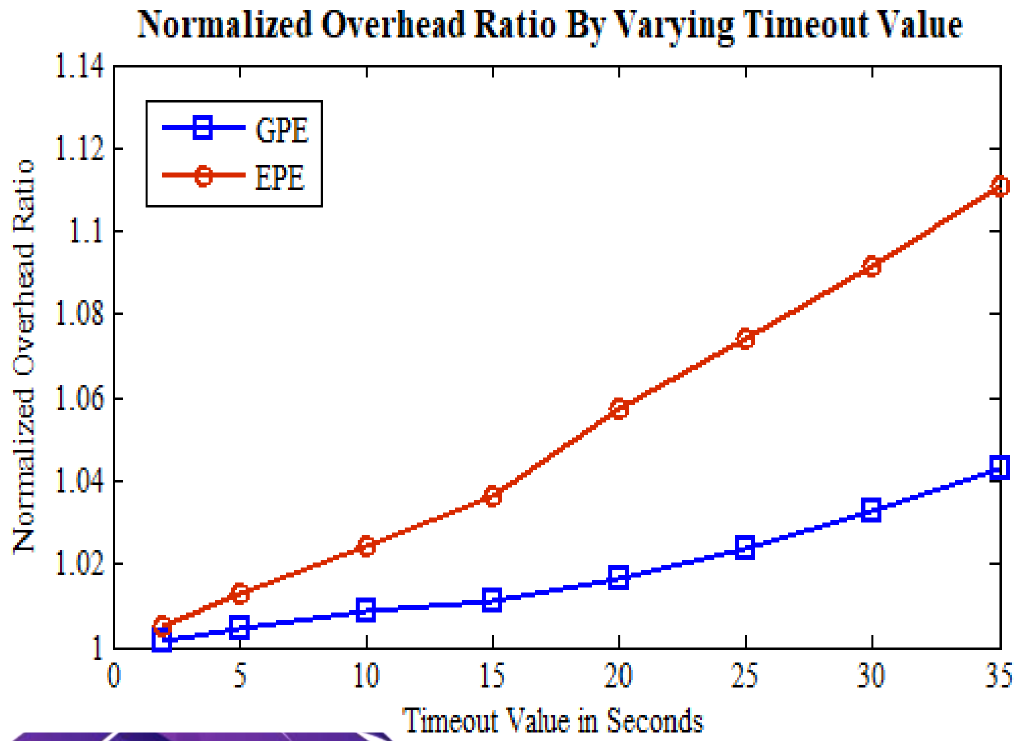


(b) Packet Violation Percentage

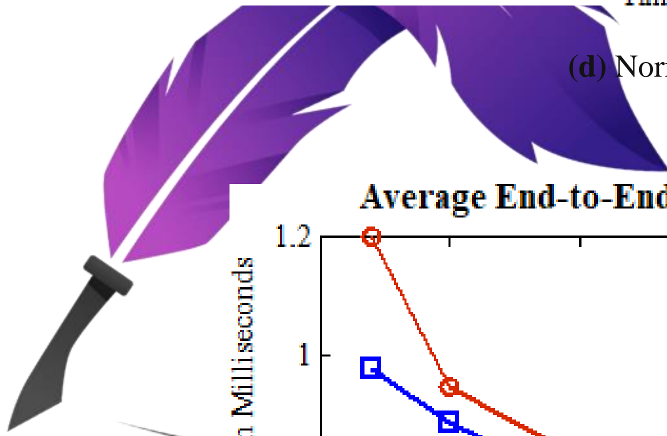


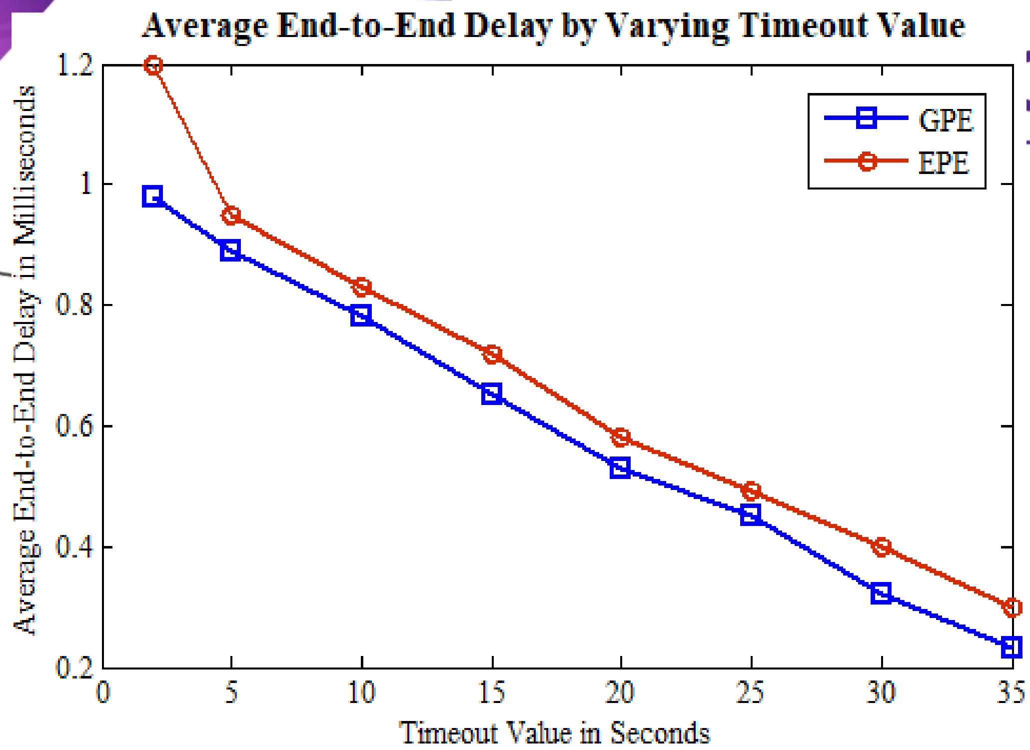


(c) Successful Packet Delivery

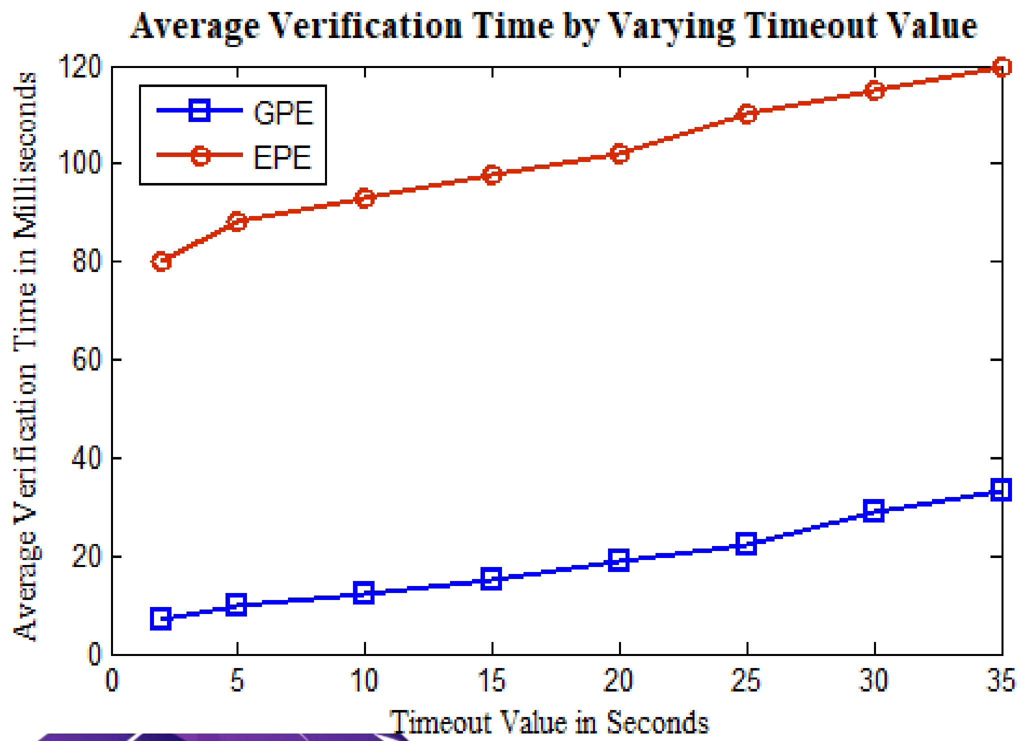


(d) Normalized Overhead

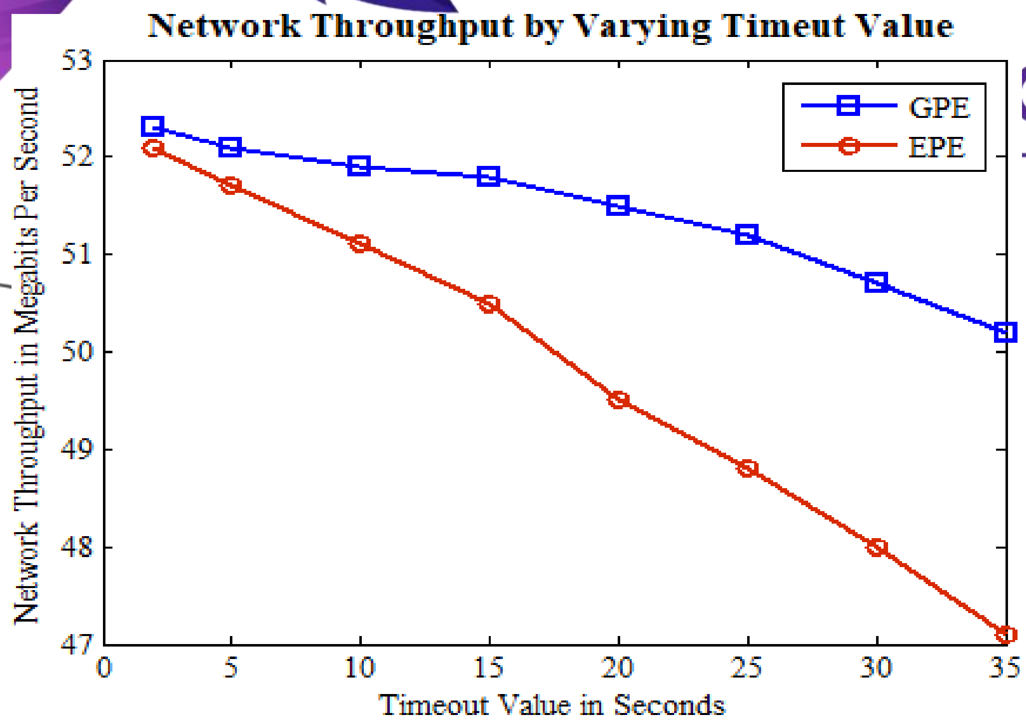




(e) Average End-to-End Delay



(f) Average Verification Time



(g) Network Throughput

Figure 5.7: Simulation Results by Varying Timeout Value

5.3.4 Simulation Results by Varying Number of Switches

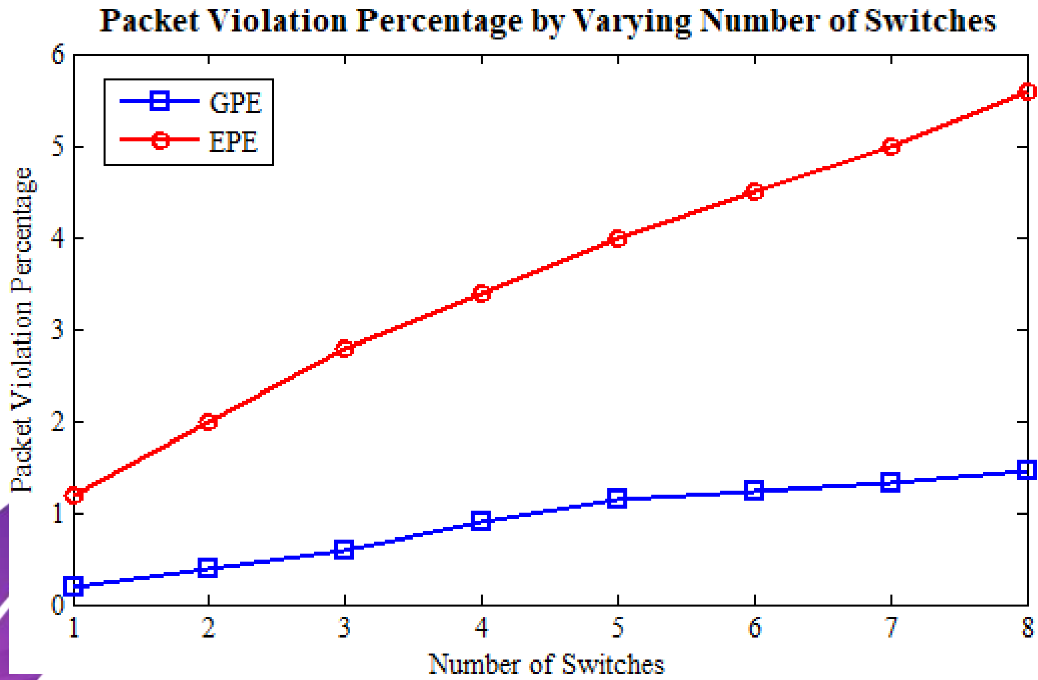
To analyze the results by varying number of switches in our network topology, the static parameters include; the frequency of policy change is set to 5, packet transmission rate is set to 0.6 ms, flow rule timeout value is set to default and dynamic parameter is chosen as number of switches which is set to 1, 2, 3, 4, 5, 6, 7, and 8 randomly. Based on the above parameters, simulation was run, and results are shown in Figure 5.8.

Figure 5.8 (a) shows that PVP is quite less in GPE as compared to EPE by varying number of switches in our network topology to analyze the effectiveness of our proposed approaches in denser and less denser networks. The reason of less packet violations in GPE is due to the effective policy implementation mechanism as compared to EPE which results in lesser packet violations by varying number of switches. Figure 5.8 (b) shows that SPD percentage is consistently better in case of GPE as compared to the EPE by varying number of switches. Therefore, our proposed approach can be implemented effectively in denser as well as in less denser networks. Figure 5.8 (c) shows that NOH ratio decreases with the decrease in number of switches and vice versa due to the computation of shortest path and accordingly installation/deletion of flow rules along the path in case of policy change. This parameter indicates that with lesser number of switches, the network is less dense and accordingly controller have less computations to perform that results in faster installation and deletion of flow rules. So, overall normalized overhead ratio decreases in less dense networks. However, this impact will be less in controller with high computation power.

ACADEMIC SOLUTIONS

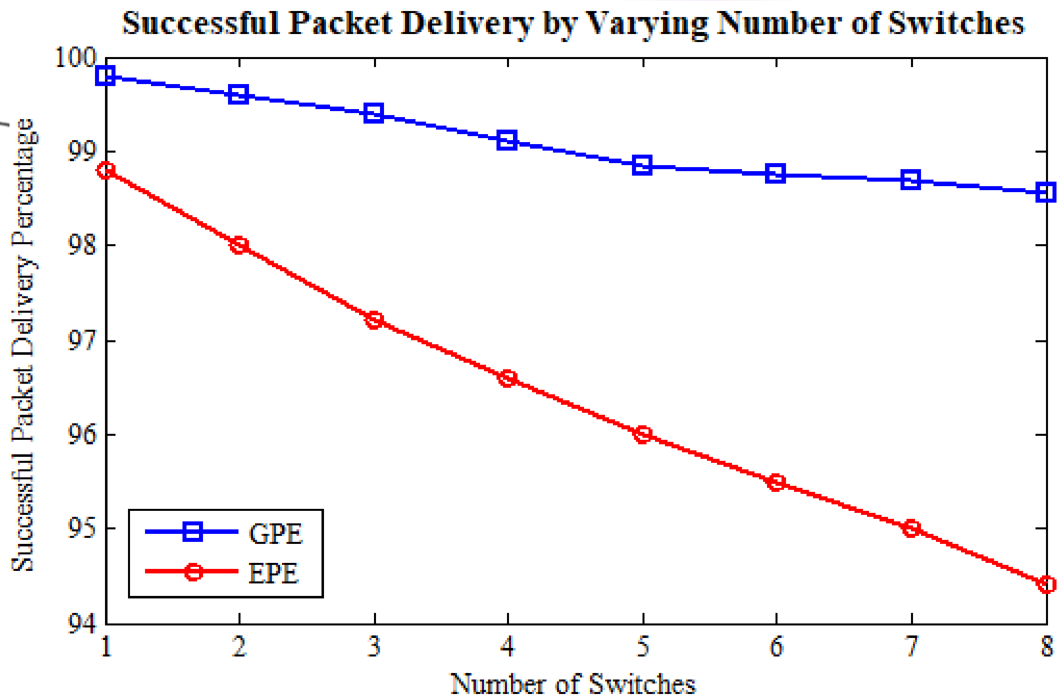
Figure 5.8 (d) indicates lower AED in case of GPE as compared to EPE in less as well as in more dense networks. The AED is increasing with the increase in number of switches and vice versa. It is due to the longer communication path in dense networks as compared to less dense networks. Figure 5.8 (e) represents that AVT is lower in less dense networks due to the reduced computations at controller. It is because there are a smaller number of switches, links and hosts which results in shorter communication path between source and destination. As a result, there is less computation time at controller to detect and implement network policy change. It is however noted that GPE

remains lower as compared to EPE in less and more dense networks. The simulation results in Figure 5.8 (f) show that GPE provides better NTP as compared to EPE by varying number of switches due to the better SPD percentage and less packet violation percentage.

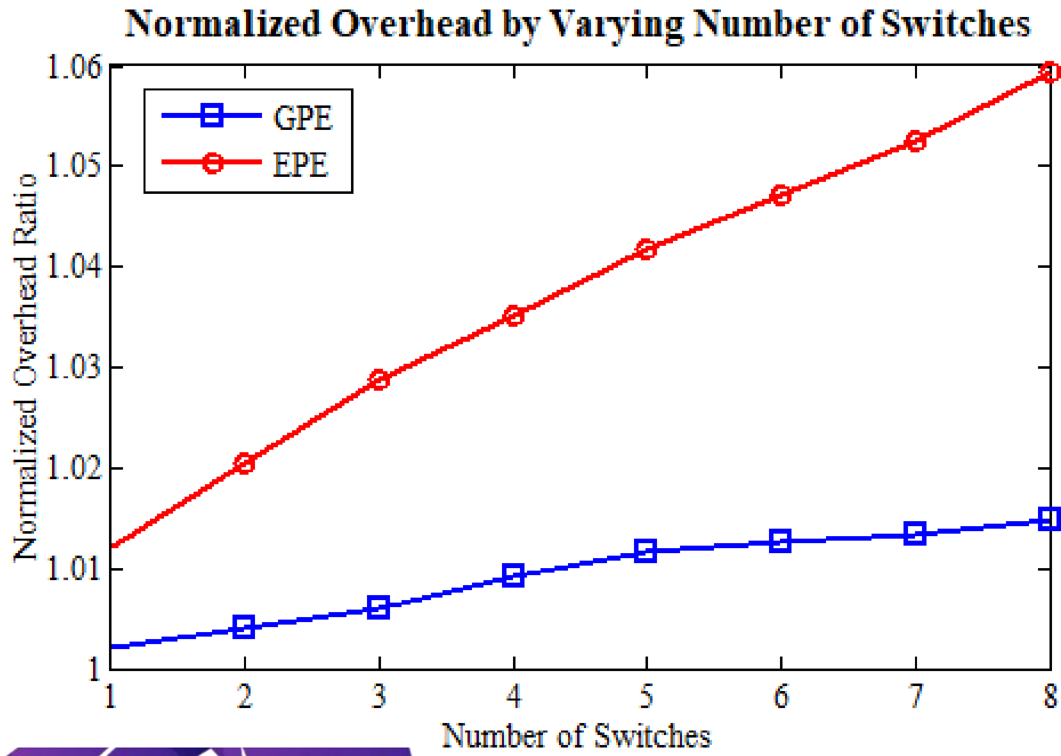


(a) Packet Violation Percentage

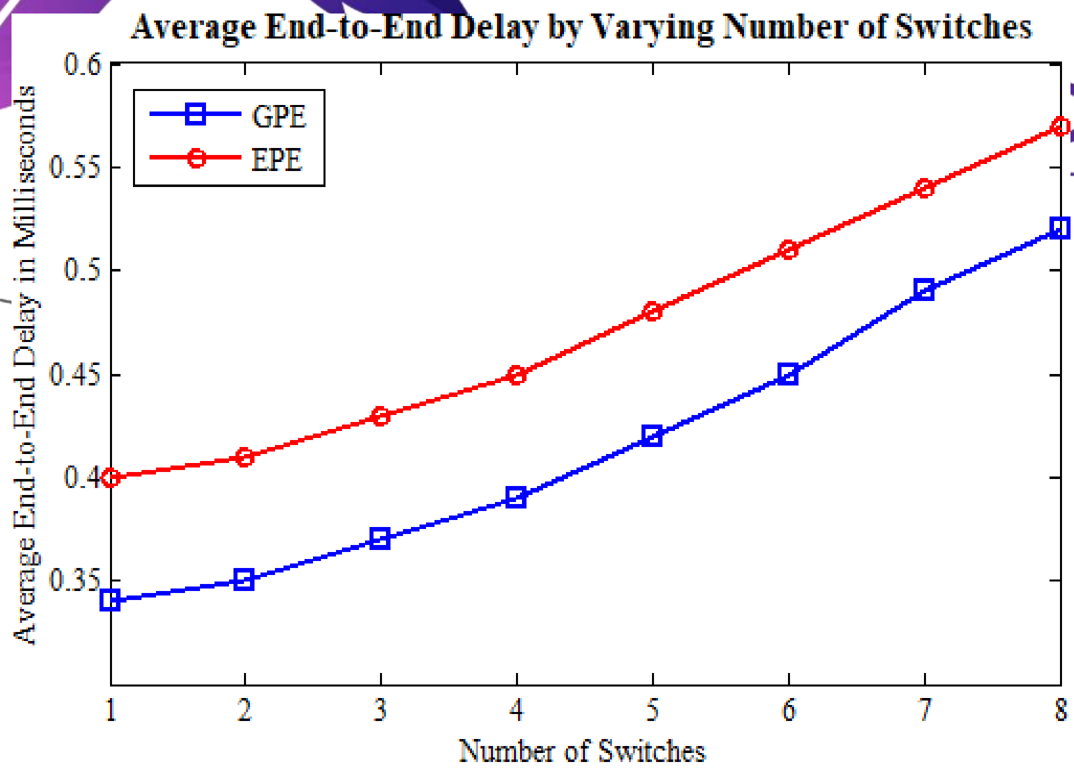
ACADEMIC SOLUTIONS



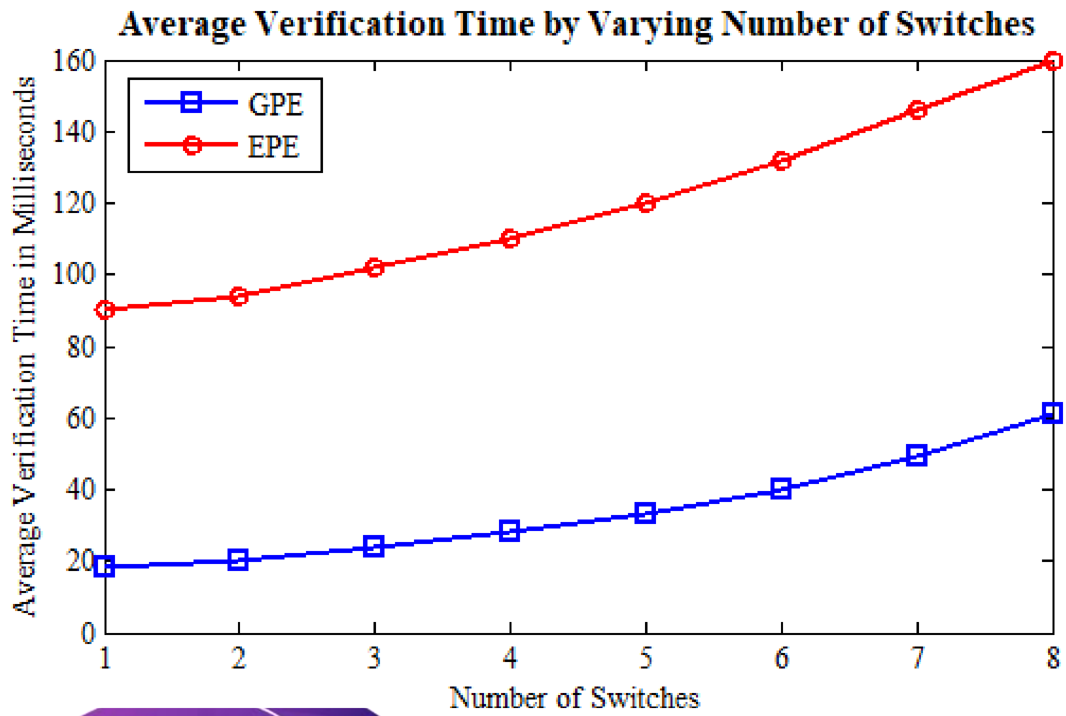
(b) Successful Packet Delivery



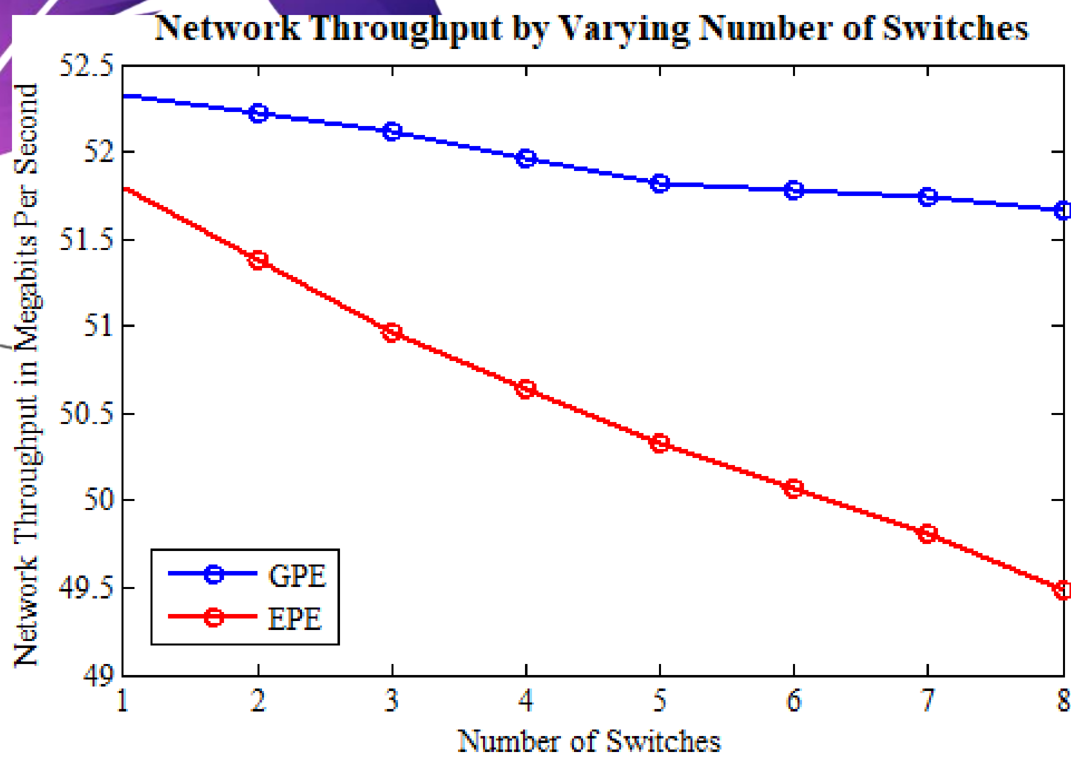
(c) Normalized Overhead



(d) Average End-to-End Delay



(e) Average Verification Time



(f) Network Throughput

Figure 5.8: Simulation Results by Varying Number of Switches

5.4 Analysis of Proposed Approaches by Utilizing Reactive and Proactive Flow Rule Installation Mechanisms

To analyze the performance of our proposed approaches (GPE and EPE) by utilizing reactive and proactive flow rule installation mechanisms, the Mininet SDN Emulator [30] and POX SDN controller [31] are used. In addition, HP Probook 450 G5 with Intel Core i5-8250U CPU@ 1.60 Ghz (8 CPUs), 8GB RAM and 1TB SATA HDD, running Linux operating system, Ubuntu 16.04 is utilized. The proposed application is written in python to be executed on POX controller. For performance analysis, the network topology is developed in Mininet 2.2 EEL by adding and configuring hosts, links, open-flow soft switches (OVS 2.5.2) which supports open flow 1.3 and higher versions, and POX SDN controller. In this research, the network topology and policies of an educational institute are used. The network comprises 9 OpenFlow soft switches and 30 hosts are added along with one SDN POX controller. Moreover, 1,000 to 80,000 packets are transmitted randomly from different sources to various destinations based on the network topology and network policies. The total simulation time of experimental evaluation is set to 100 seconds which is selected randomly for the fair analysis of results. Finally, the proposed approaches (GPE and EPE) are analyzed with the help of packet violation percentage, successful packet delivery percentage, normalized overhead ratio, average end-to-end delay, average verification time and network throughput parameters, by varying frequency of policy change, packet transmission rate and flow rule timeout values.

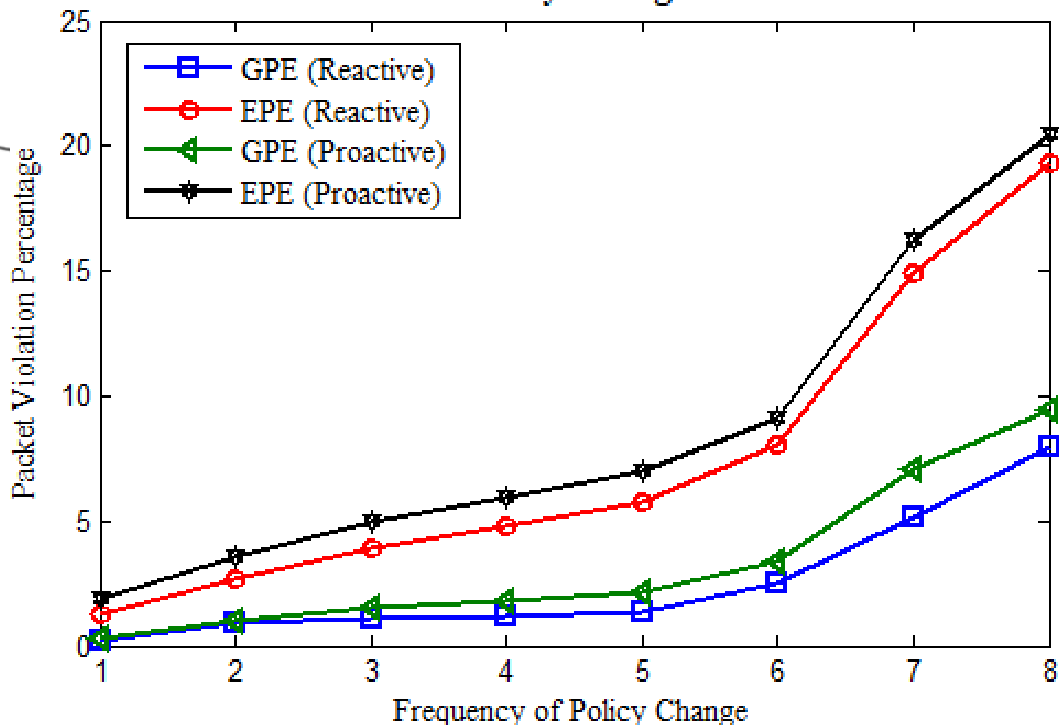
ACADEMIC SOLUTIONS

5.4.1 Simulation Results by Varying Frequency of Policy Change

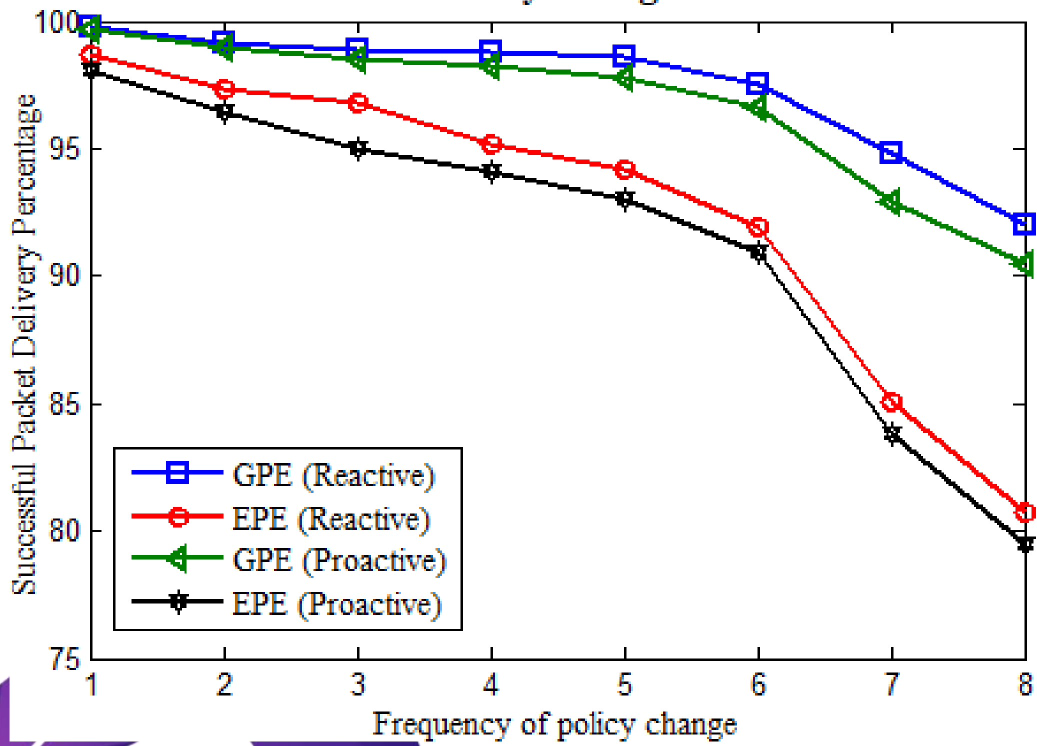
The simulation results are based on static and dynamic parameters. The static parameters include packet transmission rate that is set to 0.6 milliseconds (ms) and default time-out value of flow rules are set. The dynamic parameter is chosen as frequency of policy change, that is chosen as 1, 2, 3, 4, 5, 6, 7 and 8 policies, which changes randomly. Based on the above parameters, simulation was run, and results are shown in Figure 5.9.

In proactive flow rule installation mechanism, the flow rules are installed prior to the communication as per network policies. When the network policy change triggers, the proactive flow rule installation mechanism has high impact on the reconfiguration of

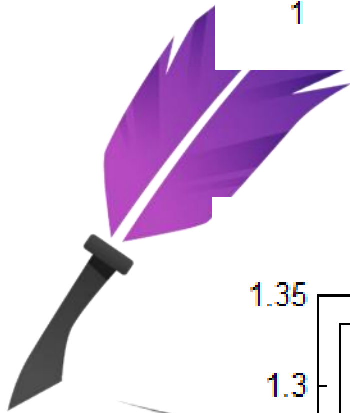
forwarding devices. To comply with the change in network policies, switches require more control transmission for the installation, deletion or modification of already installed flow rules. Specifically, the nth order flow rules placement (computing and deploying the flow rule entries for all sources and destinations of a certain policy) make switches more resource hunger in proactive flow rule installation mechanism. Dynamically policy change in proactive and reactive flow rules installation mechanism depends on the semantical reconfiguration of the reachability graph and number of policies. The proactive approach may also require disconnection of network services at a discrete-time interval to handle the exponential flux of network policies. The results also show substantial packet violation percentage in case of proactive as compared to reactive as shown in Figure 5.9 (a). Similarly, the successful packet delivery percentage and network throughput decreases with the increase in frequency of network policy change and it affects more in proactive as compared to reactive flow rule installation as shown in Figure 5.9 (b) and (d), respectively. Finally, the normalized overhead ratio is also increases in case of proactive as compared to reactive flow rule installation as shown in Figure 5.9(c) due to the more computation overhead in case of policy change.

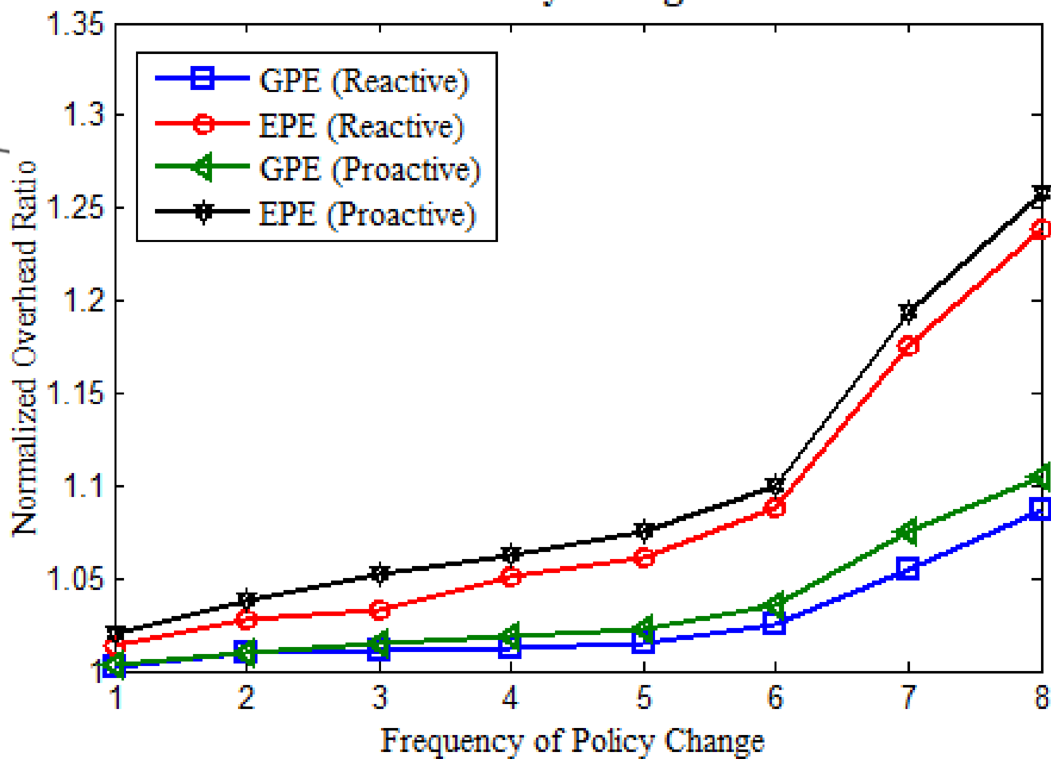


(a) Packet Violation Percentage

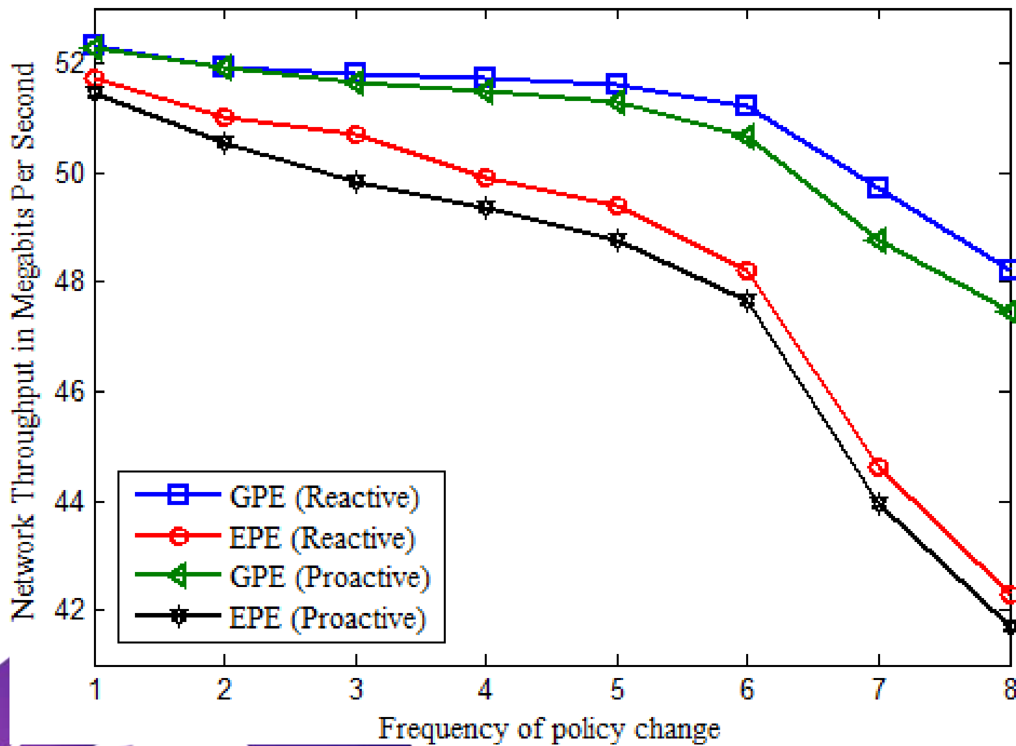


(b) Successful Packet Delivery





(c) Normalized Overhead



(d) Network Throughput

ACADEMIC SOLUTIONS

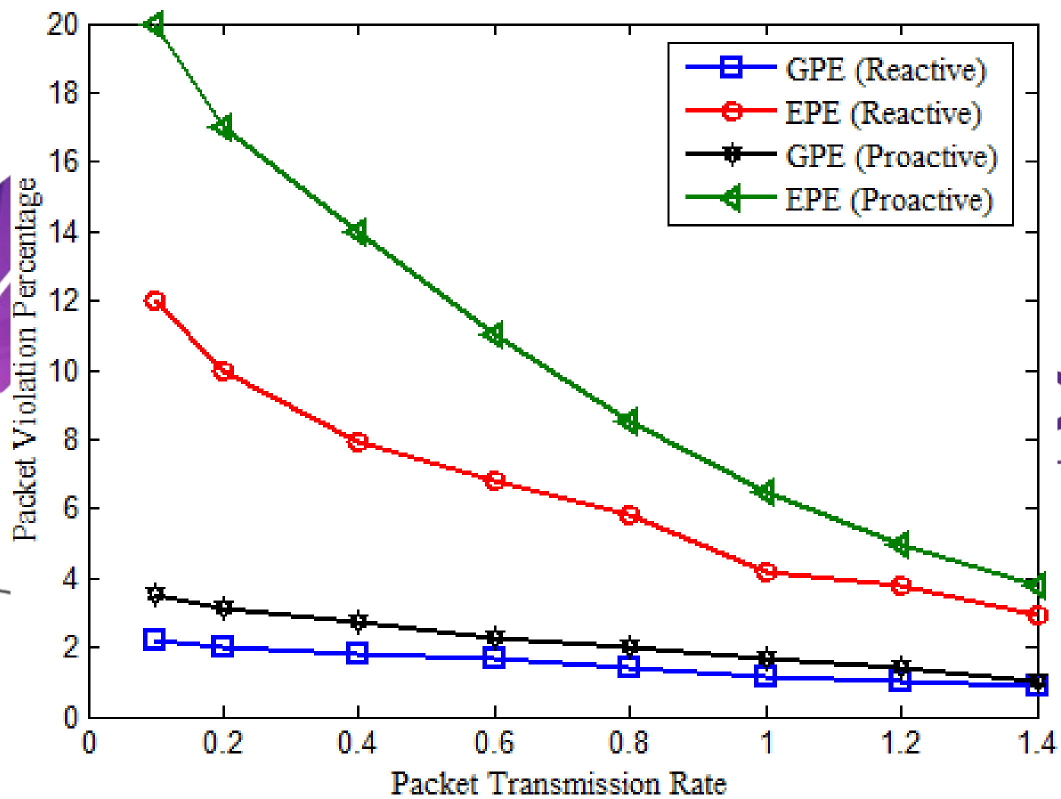
Figure 5.9: Simulation Results by Varying Frequency of Policy Change

5.4.2 Simulation Results by Varying Packet Transmission Rate

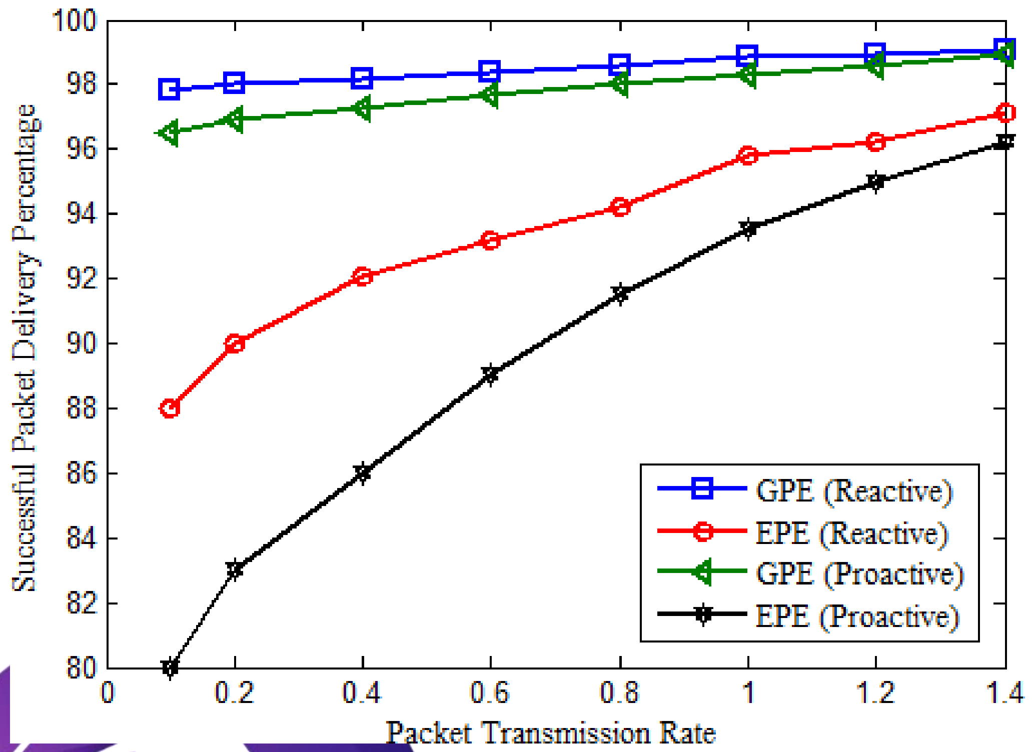
To analyze the results by varying packet transmission rate, the static parameters include; frequency of policy change is set to 5 and default timeout values of flow rule are set. The dynamic parameter is chosen as packet transmission rate that is set to 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7 and 0.8 ms randomly. Based on these parameters, simulation was run and results are shown in Figure 5.10.

Figure 5.10 shows simulation results by varying packet transmission rate. The results indicate that by decreasing packet transmission rate, a smaller number of packet violations occur in case of reactive as compared to proactive flow installation as shown in Figure 5.10 (a). Similarly, the successful packet delivery percentage and network throughput increases with the decrease in packet transmission rate in both reactive and proactive flow rule installation. However, reactive flow rule installation by varying

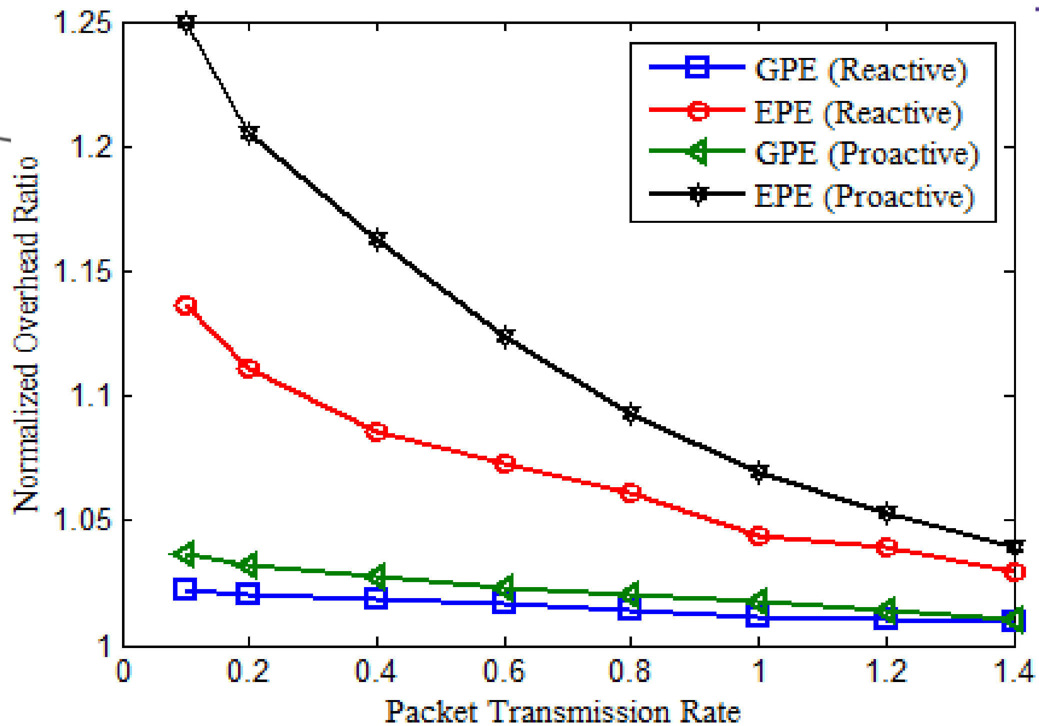
frequency of policy change performs well due to its efficient policy change detection and implementation mechanism. This because a smaller number of packets violating the network policy during implementation of policy change mechanism as shown in Figure 5.10 (b) and 5.10 (d). The normalized overhead ratio is also better in reactive due to the smaller number of flow rules installation, deletion and modification in case of reactive as compared to proactive flow rule installation as shown in Figure 5.10 (c). This is due to the fact that reactive has less overhead ratio of flow rule installation due to the selective flow rules of only those policies for which communication is desired.



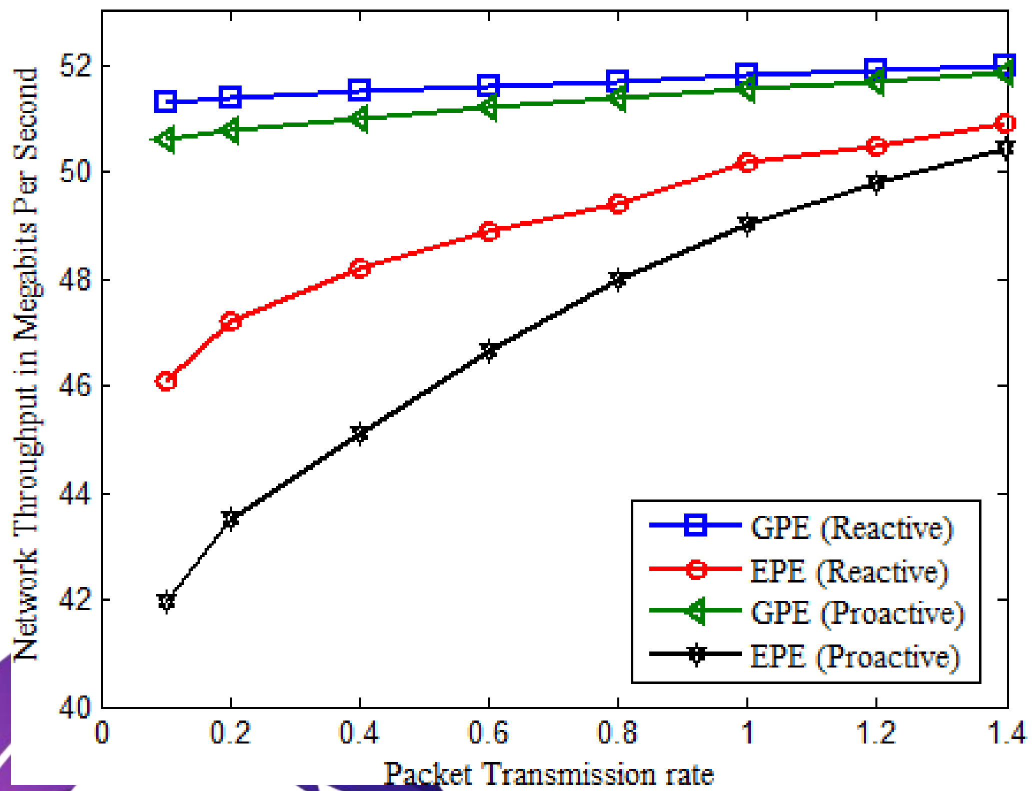
(a) Packet Violation Percentage



(b) Successful Packet Delivery Percentage



(c) Normalized Overhead



(d) Network Throughput

ACADEMIC SOLUTIONS

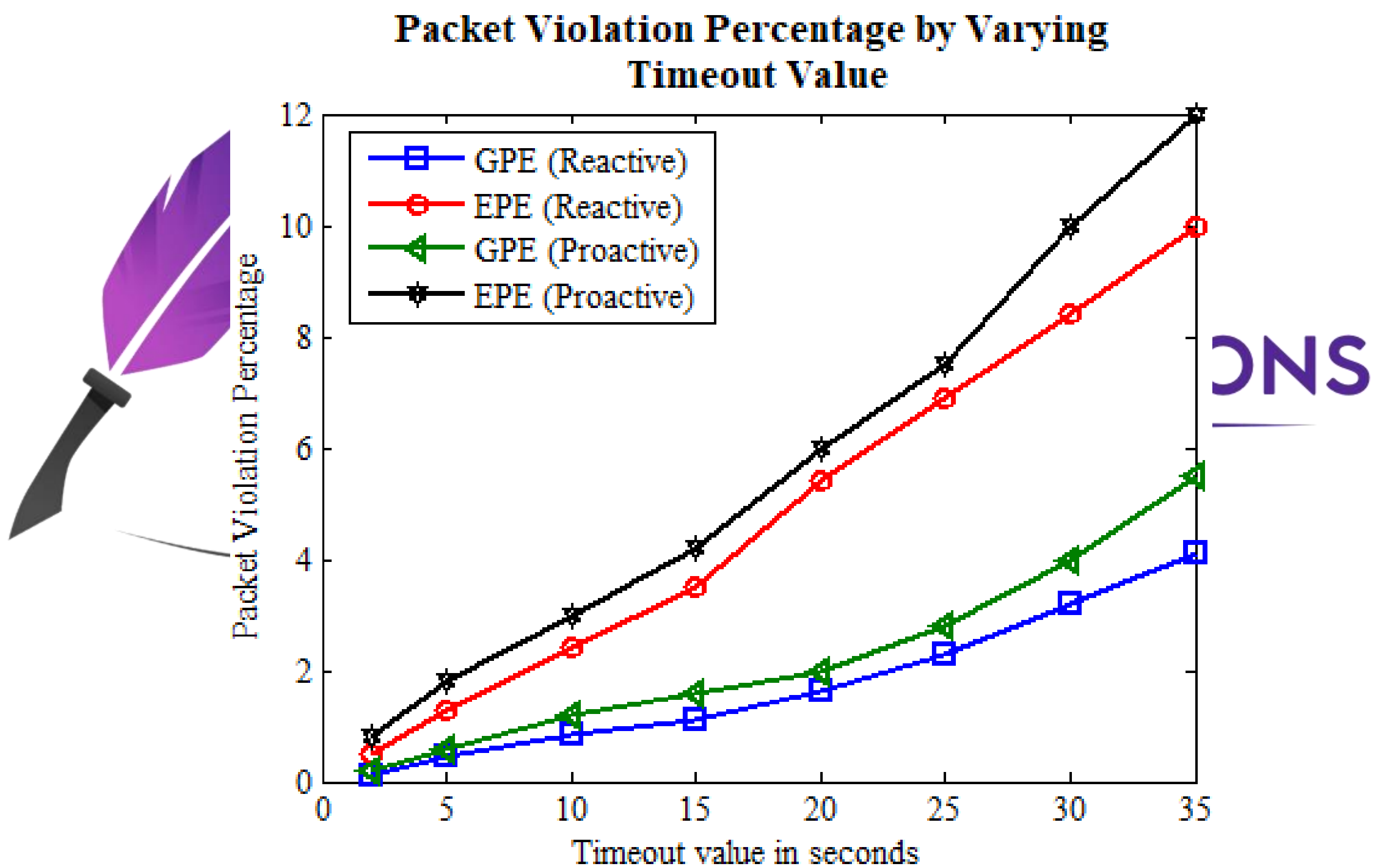
Figure 5.10: Simulation Results by Varying Packet Transmission Rate

5.4.3 Simulation Results by Varying Timeout Value

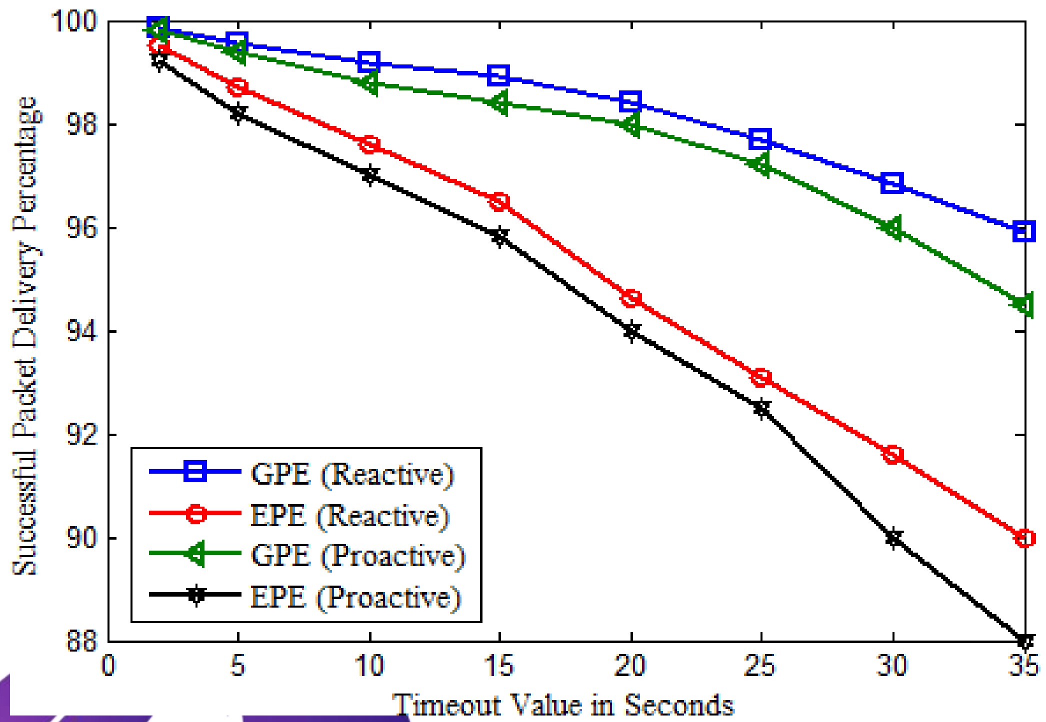
To see the results by varying timeout value of flow rules, the static parameters include; frequency of policy change which is set to 5, packet transmission rate is set to 0.6 ms and dynamic parameter is chosen as flow rule timeout value which is set to 2, 5,10, 15, 20, 25, 30, and 35 seconds randomly. Based on the above parameters, simulation was run, and results are shown in Figure 5.11.

By varying timeout values of flow rules, it impacts the packet violations. As shown in Figure 5.11 (a), the packet violation percentage increases with the increase in timeout value. It is due to the fact that flow rules remain in flow tables of switches with greater time and due to the proactive mechanism, these flow rules are installed in advance. So, in case of policy change more packet violations occur as compared to reactive mechanism. Moreover, the lookup time of flow rules in case of policy change also

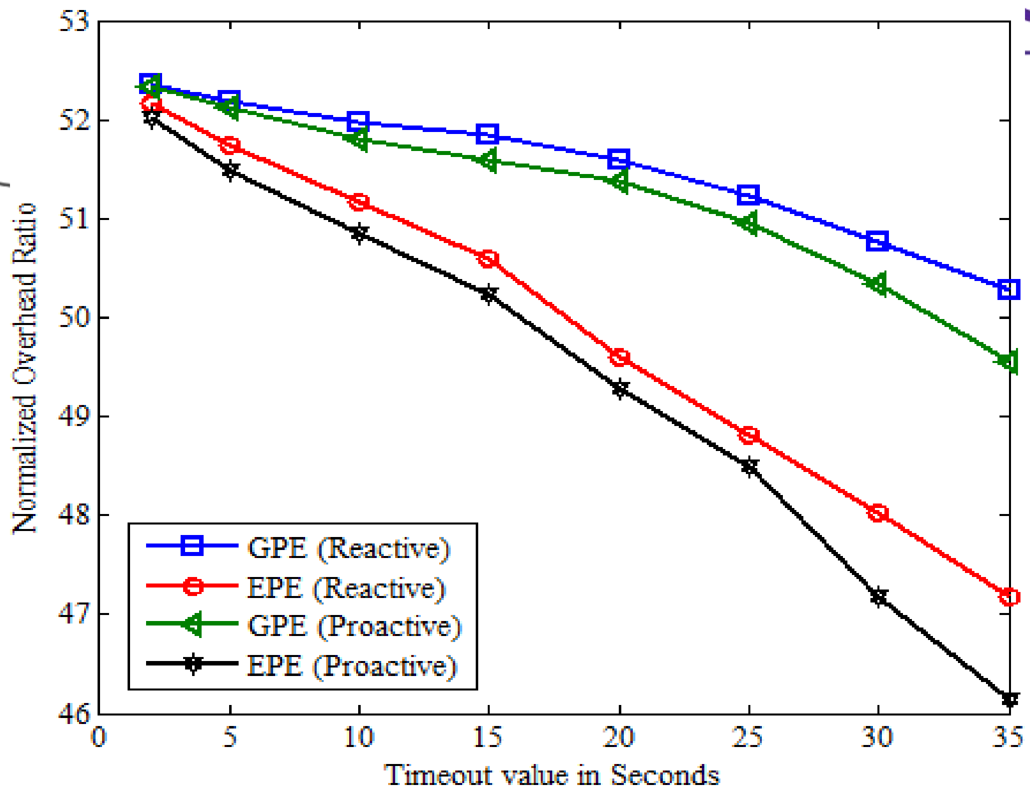
increases along with the increase in deletion of old and installation of new flow rules in proactive flow rule installation. All these factors result in increased number of packet violations in case of proactive as compared to reactive. In addition, the successful packet delivery percentage and network throughput decreases in case of proactive as compared to reactive as shown in Figure 5.11 (b) and (d). Finally, the normalized overhead ratio of proactive flow rules installation is greater than reactive due to the addition and deletion of more flow rules as compared to reactive as shown in Figure 5.11 (c). It is because in proactive flow rule installation there are flow rules which installed but never used during communication which resultantly incur additional overhead.



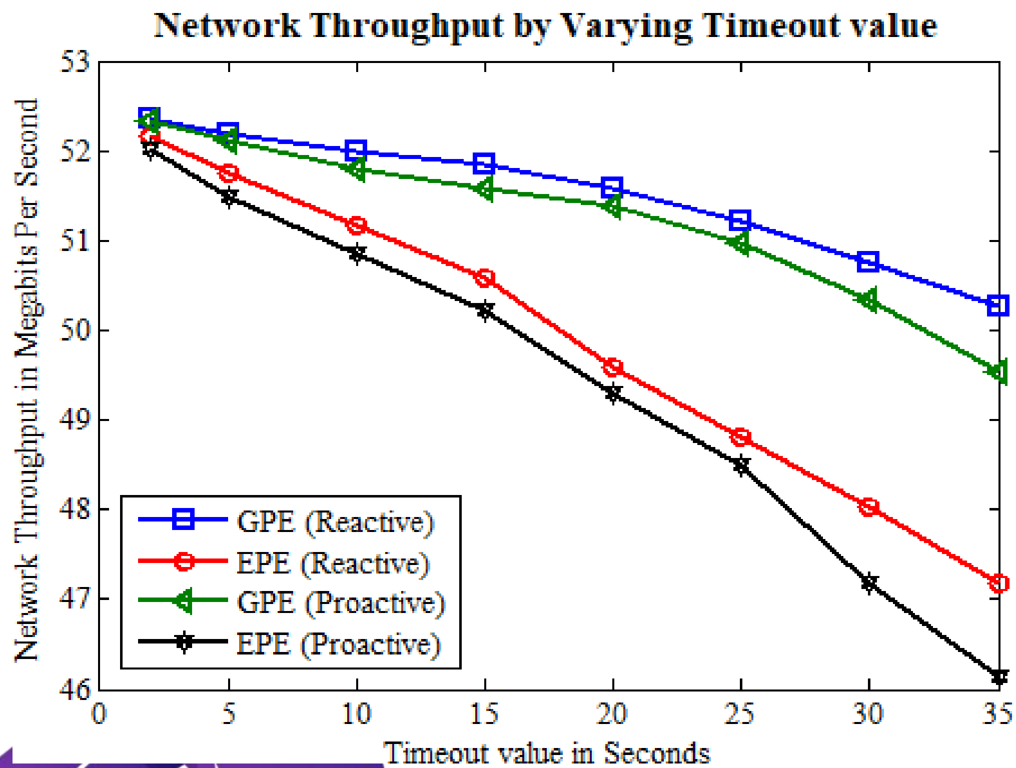
(a) Packet Violation Percentage



(b) Successful Packet Delivery



(c) Normalized Overhead



(d) Network Throughput

Figure 5.11: Simulation Results by Varying Timeout Value

5.5 Summary

ACADEMIC SOLUTIONS

In this chapter, we have presented simulation results based on our proposed approaches to show practically the improvement of Packet Violation Percentage, Successful Packet Delivery Percentage, Normalized Overhead Ratio, Average End-to-End Delay and Average Verification Time. In addition, we discussed these results by comparing with the existing approaches [52,58]. The simulation results based on GPE proposed approach indicate that Policy Change Detection Time is decreased up to 80%, Packet Violation Percentage minimized to 82%, Successful Packet Delivery Percentage increased to 12%, Normalized Overhead Ratio decreased to 13%, Average End-to-End Delay reduced to 23%, Average Verification Time reduced to 77% and Network Throughput is increased up to 14.3% as compared to EPE by varying frequency of policy change, data rate, flow timeout value and number of switches. The simulation results show that the proposed approaches help to implement network policy change mechanism in an effective manner which result in increasing network performance and efficiency.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This research work proposes two novel approaches (EPE and GPE) to handle network policy change mechanism in SDN. The EPE approach detects network policy change via matrices based on destination IP addresses and GPE with the help of multi-attributed graphs based on high level names, port number and transport layer protocols. Both proposed approaches detect policy change automatically and compute flow rules as per new policies along with the shortest path between source and destination. In addition, these approaches delete old installed flow rules from the switches that conflict with the changed policies and install new flow rules along the shortest path. Moreover, the computed flow rules are cached at controller in hash table data structure which help to track flow rules in an efficient way whenever policy change event occurs. In proposed approaches of this research work, the network policies are represented in 5 tuple based on low-level commands in case of EPE and 6 tuple via high level abstractions in case of GPE.

For experimentation and analysis of Proposed Approaches, different performance metrics are utilized. In addition, the network topology and network policies of an educational institute are opted for fair analysis. The results based on first proposed approach EPE show that the Packet Violation Percentage is reduced to 99.8%, Successful Packet Delivery Percentage is increased up to 98%, Normalized Overhead Ratio is decreased up to 88.8% and Network Throughput is increased up to 99.8% as compared to existing approaches [52,58]. Moreover, the simulation results based on second proposed approach GPE indicate that Policy Change Detection Time is decreased up to 80%, Packet Violation Percentage minimized to 82%, Successful Packet Delivery Percentage increased to 12%, Normalized Overhead Ratio decreased to 13%, Average End-to-End Delay reduced to 23%, Average Verification Time reduced to 77% and Network Throughput is increased up to 14.3% as compared to EPE by varying policy change frequency, data rates and flow timeout values. The simulation results show that proposed approaches help to implement network policy change mechanism in an effective way which results in increasing the network efficiency.

ACADEMIC SOLUTIONS

6.2 Future Work

Our research work can be extended in multiple ways. The network policy change mechanism can be analyzed by utilizing link/node failures to calculate the optimum path for flow rule installation. The link/node failures occur frequently in communication networks which result in modifying network topology and policies. Due to the delay in implementing network policies as per changed network topology cause packet violations. PrePass-Flow [226] predicts link failures in advance by recomputing the locations of network policies. Moreover, it installs flow rules based on recomputed location as per network policies proactively. In this way, it helps to minimize packet violations and network reachability problems in case of link failures. It utilizes two supervised ML-based models, Logistic Regression (LR) model and Support Vector Machine (SVM) model to predict link failures. By implementing such kind of mechanisms with our proposed approaches will help to improve network efficiency and availability. In addition, in SDN Policy Languages, Controller Platforms and Simulation/Emulation Tools, the network policy change mechanism can also be employed to facilitate network administrators while implementing network policies.

ACADEMIC SOLUTIONS

This work may be integrated with network firewalls [215-217] to effectively implement network policy change mechanism. By implementing this work, the network communication will be inspected in a better way and packet violations will be avoided that will results in increasing overall network performance. This work may also be expanded to integrate it with network monitoring and debugging tools/schemes [218-221] to analyze the effectiveness of the policy change mechanism. Furthermore, it may also be investigated that how security applications running on SDN controller and traditional network security protocols behave in case of network policy change. In addition, it can be extended in implementing environments where distributed controller platforms [222-224] need to be implemented and analyzed to investigate the network performance. This work can be investigated with diverse implementation areas of SDN, like, energy efficient SDN networks, wireless networks, network virtualization, cloud computing platforms and SDN migration mechanisms. The proposed mechanisms can be extended in hybrid SDN deployments to analyze the effectiveness of this research work with respect to network efficiency.

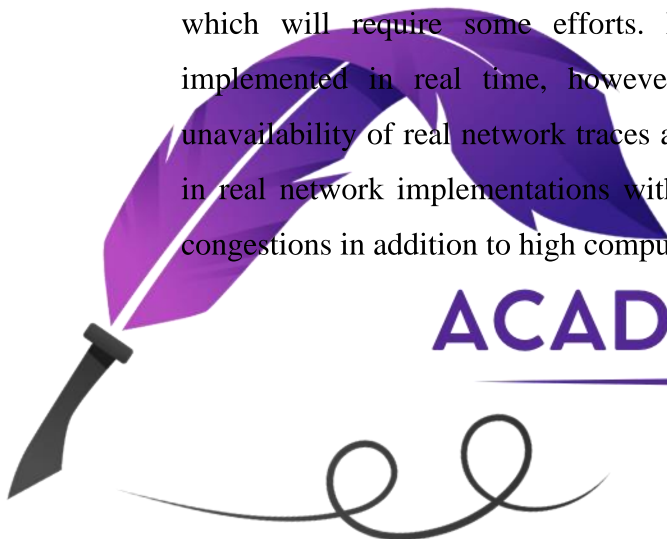
The network programmability feature of SDN is used to enrich response functionality. The data plane provides a possibility of adding new functions that are more competent to secure the entire network. In prevention systems, security policies are defined to stop attackers to contact targets which require investigation once policies change. The dynamic flow control features of SDN enhance the detection of attacks without adding middleboxes and virtually turn switches to network security devices that can prevent attacks dynamically. Moreover, ML-based SDN may include network optimization, improving network security and high-quality training datasets. Some other broader perspectives of SDN like software-defined mobile networks and software-defined vehicular networks are also important areas to explore. Regarding QoS, the researchers are crying out experiments with real matrix through different network topologies that each flow may have different QoS requirements. If DROM [225] is extended with QoS routing, more efficient and enhanced results can be generated. The QoS measures the traffic conditions and traffic classification while DROM dynamically measures the reliability, effectiveness and awareness of QoS. The queuing delay of switches and the processing delay of the server improves the QoS. This can be achieved by using a more efficient and effective way of network policy implementation.

ACADEMIC SOLUTIONS

For traffic engineering, some machine learning techniques provide fundamental improvements as compared to the traditional traffic engineering paradigm. Many researchers devoted their skills to develop efficient systems for traffic classification, routing, and traffic optimizations. If network policies are formed intelligently using some machine learning algorithms i.e., random forest, Support Vector Machine (SVM) [226], K-Nearest Neighbors (KNN), etc. Apart from machine learning, deep learning models such as Artificial Neural Network (ANN), Convolutional Neural Network (CNN) may also be adopted. For example, if we can predict the change in network policies before its occurrence and take the needful measures, then we can achieve much better throughput. These approaches will make controller more intelligent which results in more efficient handling of network policy change phenomena. Moreover, Quality of Service (QoS) parameters also need to be considered to manage the network traffic to improve network throughput. However, machine learning algorithms have their own limitations, like false negative [227]. For traffic classification, most of the research conducted so far is on labeled data set using a supervised learning approach. Few works are done using semi-supervised learning where some of the data are labeled and some

are not given labels. The same is the scenario with unsupervised learning. Another important learning approach is reinforcement learning which is a black box approach when we consider traffic classification. In this regard, algorithms can be designed to classify the traffic data in such a way that the new classified data learns from the experience.

We have implemented and tested our proposed approaches using Mininet [30] with POX controller [31] and Pyretic language[11] with limited number of policies, hosts and switches. Therefore, our proposed approaches can be used directly on those SDN controllers which support Pyretic language. However, our proposed approaches can be implemented in other SDN controllers which use other than Pyretic language by mapping the network policies of such language into our format of network policies which will require some efforts. Moreover, our proposed approaches can be implemented in real time, however, we were unable to implement it due to unavailability of real network traces and network environment. The results may vary in real network implementations with large number of network policies and traffic congestions in addition to high computation power of SDN controller.


ACADEMIC SOLUTIONS

Chapter 7

References

[1] T. Benson, A. Akella, and D. Maltz, “Unraveling the complexity of network management,” in Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, ser. NSDI’09, Berkeley, CA, USA, pp. 335–348, 2009.

[2] N. Feamster, J. Rexford, and E. Zegura, “The road to SDN: An intellectual history of programmable networks,” ACM SIGCOMM Computer Communication Review, vol. 44, no. 2, pp. 87–98, Apr. 2014.

[3] N. Mckeown, “How SDN will Shape Networking,” October 2011. [Online]. Available: http://www.youtube.com/watch?v=c9-K5OqYgA [Accessed: September 5, 2016]

[4] R. Alvigzu, G. Maier, N. Kukreja, A. Pattavina, R. Morro, A. Capello, and C. Cavazzoni, “Comprehensive survey on SDN: Software defined networking for transport networks,” IEEE Communications Surveys & Tutorials, vol. 19, no. 4, pp. 2232–2283, 2017.

[5] M. Mousa, A. M. Bahaa-Eldin, and M. Sobh, “Software defined networking concepts and challenges,” in Proc. Int. Conf. on IEEE Computer Engineering & Systems (ICCES), pp. 79–90, 2016.

[6] S. Schenker, “The Future of Networking, and the Past of Protocols”, October 2011. [Online]. Available: http://www.youtube.com/watch?v=YHeyuD89n1Y [Accessed: September 20, 2016]

[7] Kerravala, Zeus. "Configuration management delivers business resiliency." The Yankee Group , 2002.

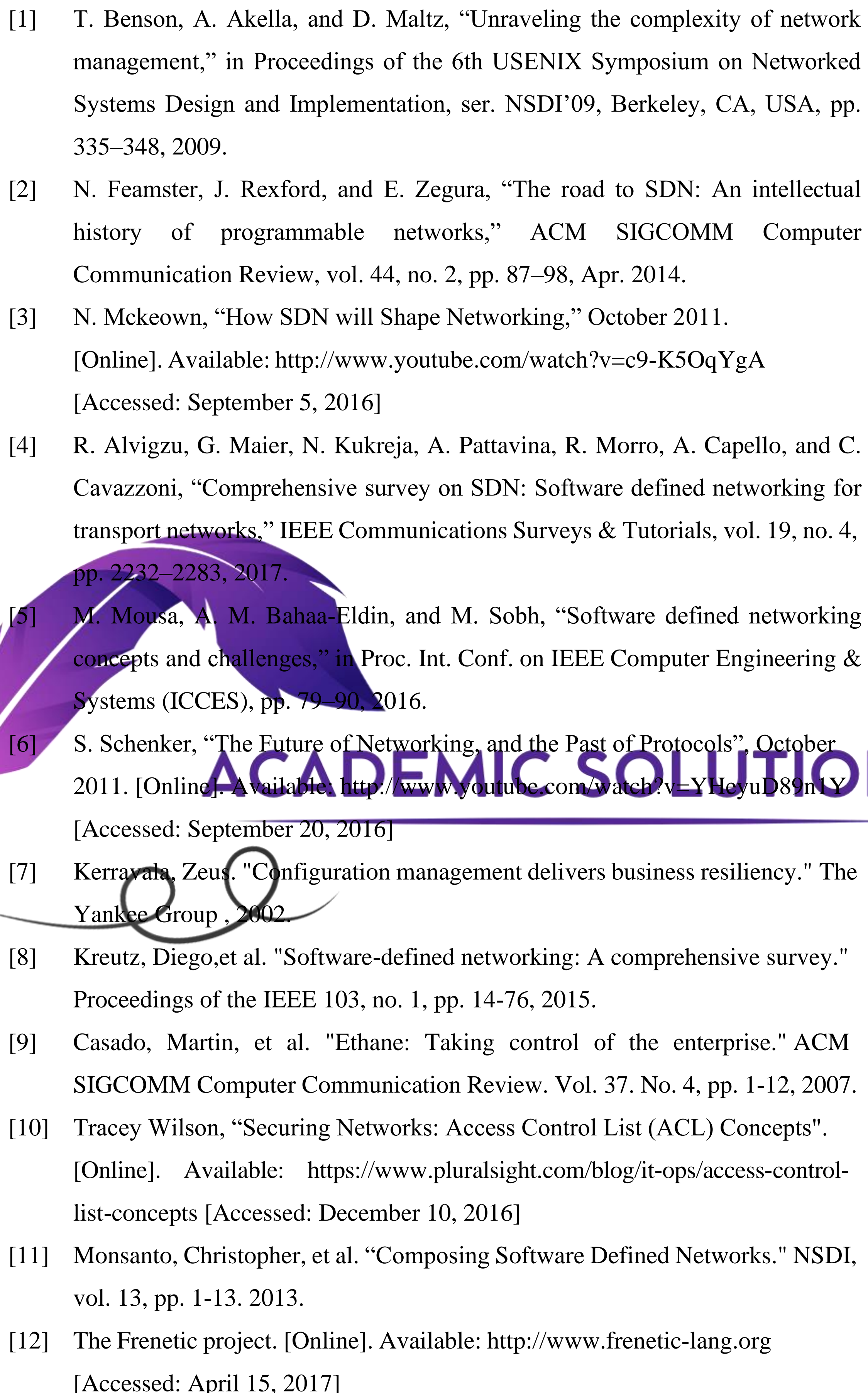
[8] Kreutz, Diego,et al. "Software-defined networking: A comprehensive survey." Proceedings of the IEEE 103, no. 1, pp. 14-76, 2015.

[9] Casado, Martin, et al. "Ethane: Taking control of the enterprise." ACM SIGCOMM Computer Communication Review. Vol. 37. No. 4, pp. 1-12, 2007.

[10] Tracey Wilson, “Securing Networks: Access Control List (ACL) Concepts". [Online]. Available: https://www.pluralsight.com/blog/it-ops/access-control-list-concepts [Accessed: December 10, 2016]

[11] Monsanto, Christopher, et al. “Composing Software Defined Networks." NSDI, vol. 13, pp. 1-13. 2013.

[12] The Frenetic project. [Online]. Available: http://www.frenetic-lang.org [Accessed: April 15, 2017]


ACADEMIC SOLUTIONS

[13] Voellmy, Andreas, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. "Maple: Simplifying SDN programming using algorithmic policies." In ACM SIGCOMM Computer Communication Review, vol. 43, no. 4, pp. 87-98. ACM, 2013.

[14] McKeown, Nick, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: enabling innovation in campus networks." ACM SIGCOMM Computer Communication Review, 38:2, pp. 69-74, 2008.

[15] Understanding OpenFlow Flow Entry Timers on Devices Running Junos OS. [Online]. Available: www.juniper.net/documentation/enuS/junos13.3/topics jconcept/junos-sdn-openflow-flow-entry-timers-overview.html [Accessed: December 15, 2016]

[16] Zhu, Huikang, Hongbo Fan, Xuan Luo, and Yaohui Jin. "Intelligent timeout master: Dynamic timeout for SDN-based data centers." In Integrated Network Management (IM), IFIP /IEEE International Symposium on, pp. 734-737, 2015.

[17] Berde, Pankaj, et al. “ONOS: towards an open, distributed SDN OS." Proceedings of the third workshop on Hot topics in software defined networking. ACM, pp. 1-6, 2014.

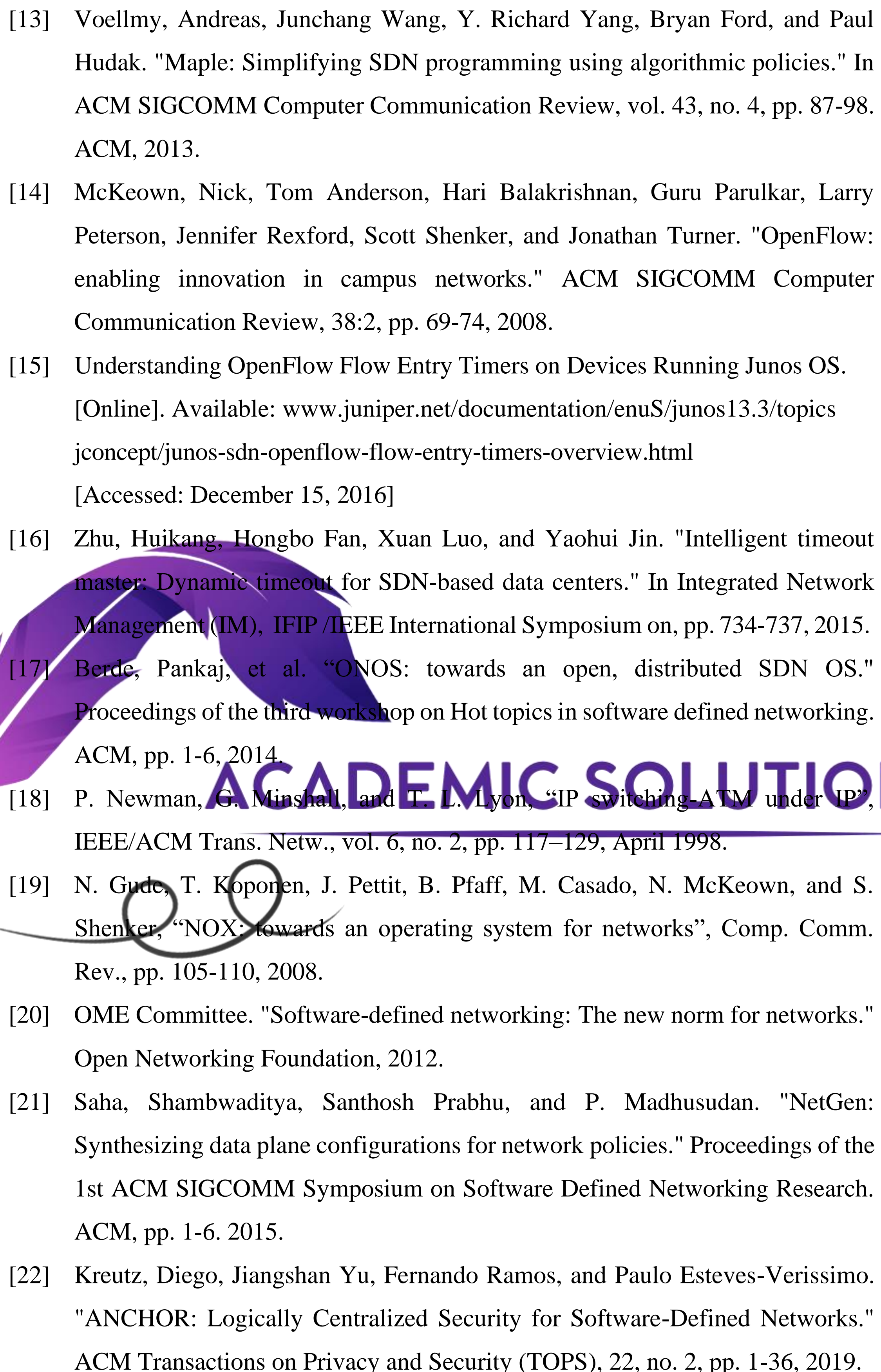
[18] P. Newman, G. Minshall, and T. L. Lyon, “IP switching-ATM under IP”, IEEE/ACM Trans. Netw., vol. 6, no. 2, pp. 117–129, April 1998.

[19] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, “NOX: towards an operating system for networks”, Comp. Comm. Rev., pp. 105-110, 2008.

[20] OME Committee. "Software-defined networking: The new norm for networks." Open Networking Foundation, 2012.

[21] Saha, Shambwaditya, Santhosh Prabhu, and P. Madhusudan. "NetGen: Synthesizing data plane configurations for network policies." Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research. ACM, pp. 1-6. 2015.

[22] Kreutz, Diego, Jiangshan Yu, Fernando Ramos, and Paulo Esteves-Verissimo. "ANCHOR: Logically Centralized Security for Software-Defined Networks." ACM Transactions on Privacy and Security (TOPS), 22, no. 2, pp. 1-36, 2019.



[23] Varadharajan, Vijay, Kallol Karmakar, Uday Tupakula, and Michael Hitchens. "A Policy-Based Security Architecture for Software-Defined Networks." IEEE Transactions on Information Forensics and Security, 14:4, pp. 897-912, 2019.

[24] Braga, Rodrigo, et al. "Lightweight DDoS flooding attack detection using NOX/OpenFlow." In LCN, vol. 10, pp. 408-415. 2010.

[25] Giotis, Kostas, Georgios Androulidakis, and Vasilis Maglaris. "Leveraging SDN for efficient anomaly detection and mitigation on legacy networks." In IEEE third European Workshop on Software Defined Networks, pp. 85-90, 2014.

[26] Z. Kerravala, "Configuration management delivers business resiliency", The Yankee Group, November 2002.

[27] Data Center SDN. [Online]. Available: https://www.opennetworking.org/images/stories/news-and-events/sdn-solutions-showcase/data-centers-final.pdf [Accessed: August 30, 2019]

[28] Inter-Datacenter WAN with centralized TE using SDN and OpenFlow. [Online]. Available: https://www.opennetworking.org/wp-ontent/uploads/2013/02/cs-googlesdn.pdf [Accessed: August 30, 2019]

[29] Google opens up on its SDN. [Online]. Available: https://www.networkworld.com/article/2937656/google-opens-up-on-its-sdn.html [Accessed: August 30, 2019]

[30] Mininet Network Emulator. [Online]. Available. www.mininet.org [Accessed: August 10, 2019]

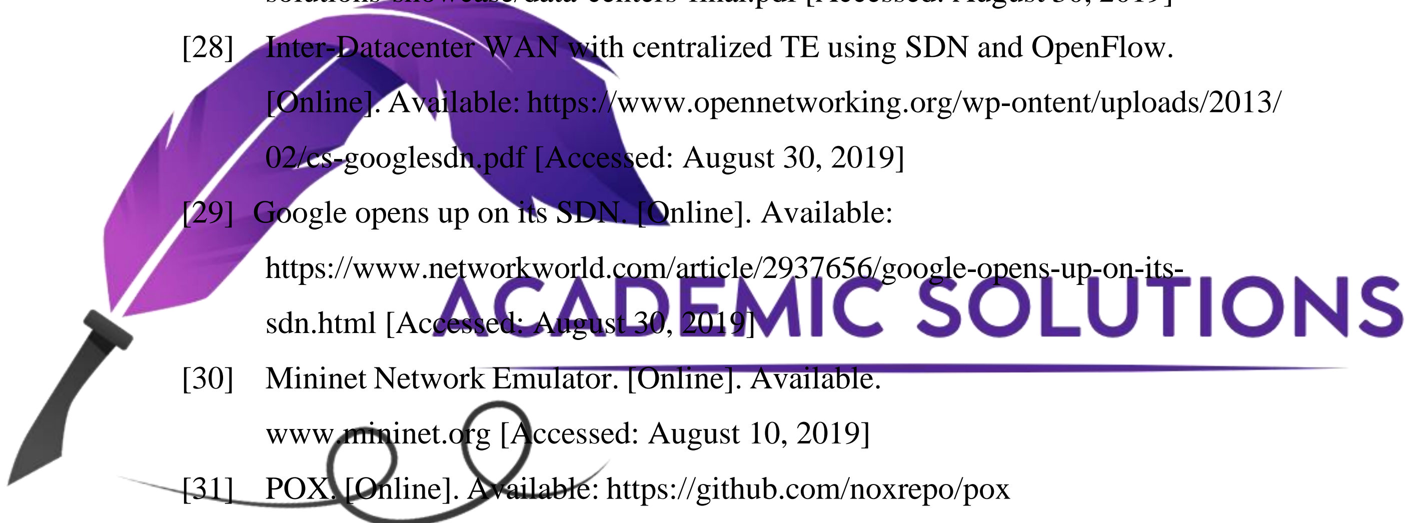
[31] POX. [Online]. Available: https://github.com/noxrepo/pox [Accessed: June 20, 2019]

[32] Berde, Pankaj, et al. "ONOS: towards an open, distributed SDN OS." Proceedings of the third workshop on hot topics in software defined networking, ACM, pp. 1-6, 2014.

[33] Dixit, Advait, et al. "Towards an elastic distributed SDN controller." ACM SIGCOMM Computer Communication Review, Vol. 43. No. 4, pp. 7-12, 2013.

[34] Tootoonchian, Amin, and Yashar Ganjali. "HyperFlow: A distributed control plane for OpenFlow." Proceedings of the Internet network management conference on research on enterprise networking, Volume 3, 2010.

[35] Katta, Naga, et al. "Ravana: Controller fault-tolerance in software-defined networking." Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, pp. 1-12, 2015.



[36] Blial, Othmane, Mouad Ben Mamoun, and Redouane Benaini. "An overview on SDN architectures with multiple controllers." Journal of Computer Networks and Communications, 10.1155, 2016.

[37] Braun, Wolfgang, and Michael Menth. “Software-defined networking using OpenFlow: Protocols, applications and architectural design choices." Future Internet, 6.2, pp. 302-336, 2014.

[38] Sarrar, N.; Uhlig, S., Feldmann, A., Sherwood, R., Huang, X., “Leveraging Zipf’s Law for Traffic Offloading.” Computer Communication Review, 42 (1), pp. 16-22, 2012

[39] D. Hock, et al., “Pocoplc: Enabling dynamic pareto-optimal resilient controller placement in SDN networks.” IEEE Conference in Computer Communications.” pp. 115–116, 2014

[40] M. Guo and P. Bhattacharya, “Controller placement for improving resilience of software-defined networks,” IEEE Fourth International Conference on Networking and Distributed Computing, pp. 23-27, 2013.

[41] Y. Hu, W. Wendong, X. Gong, X. Que, and C. Shiduan, “Reliability aware controller placement for software-defined networks.” IFIP/IEEE International Symposium on Integrated Network Management, pp. 672–675, 2013.

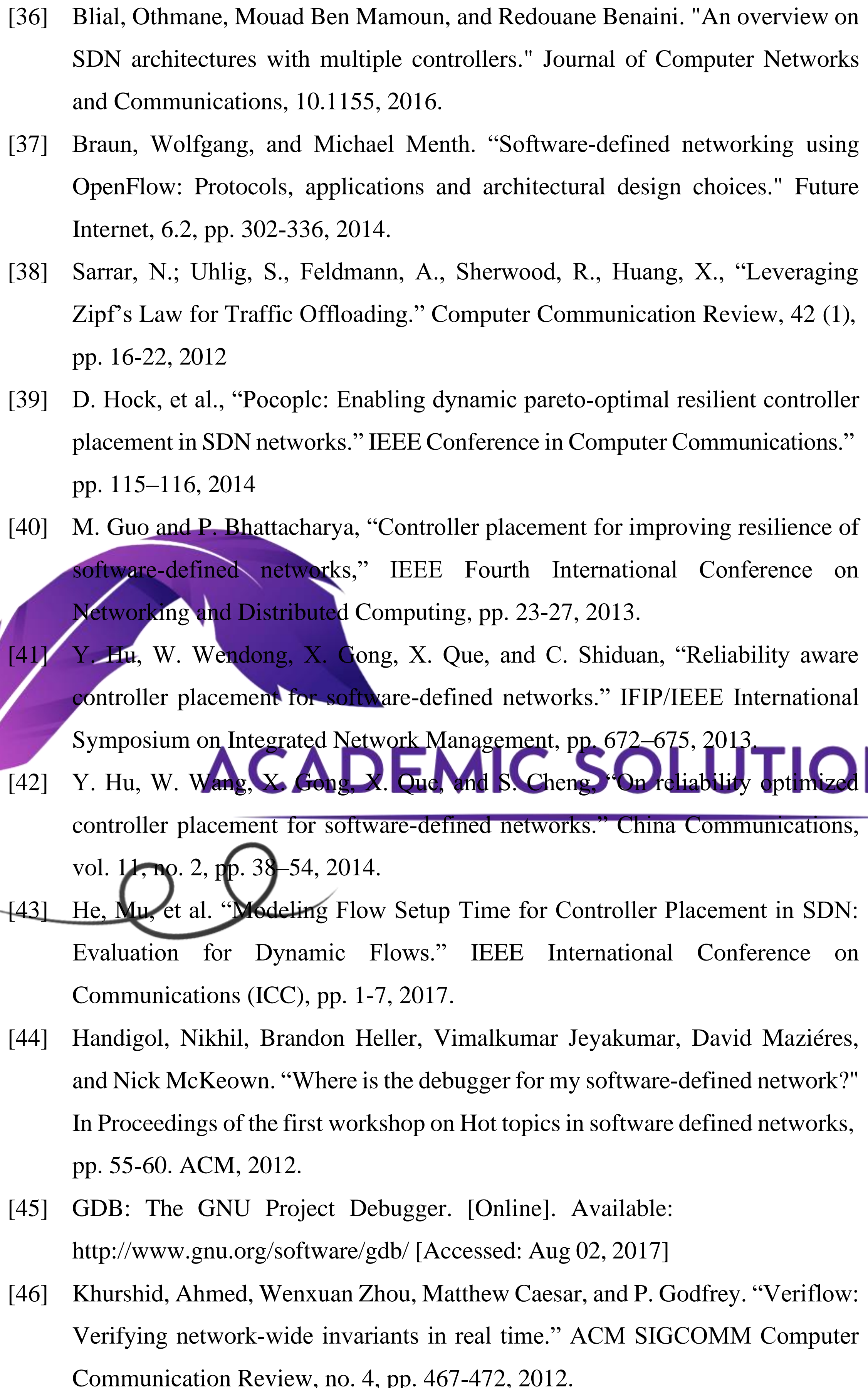
[42] Y. Hu, W. Wang, X. Gong, X. Que, and S. Cheng, “On reliability optimized controller placement for software-defined networks.” China Communications, vol. 11, no. 2, pp. 38–54, 2014.

[43] He, Mu, et al. “Modeling Flow Setup Time for Controller Placement in SDN: Evaluation for Dynamic Flows.” IEEE International Conference on Communications (ICC), pp. 1-7, 2017.

[44] Handigol, Nikhil, Brandon Heller, Vimalkumar Jeyakumar, David Maziéres, and Nick McKeown. “Where is the debugger for my software-defined network?" In Proceedings of the first workshop on Hot topics in software defined networks, pp. 55-60. ACM, 2012.

[45] GDB: The GNU Project Debugger. [Online]. Available: http://www.gnu.org/software/gdb/ [Accessed: Aug 02, 2017]

[46] Khurshid, Ahmed, Wenxuan Zhou, Matthew Caesar, and P. Godfrey. “Veriflow: Verifying network-wide invariants in real time.” ACM SIGCOMM Computer Communication Review, no. 4, pp. 467-472, 2012.



[47] Varghese, G., “Network Algorithmics: An interdisciplinary approach to designing fast networked devices.”, 2004.

[48] Al-Shaer, Ehab, et al. “Network configuration in a box: Towards end-to-end verification of network reachability and security.” 17th IEEE International Conference on Network Protocols, ICNP, pp. 123-132, 2009.

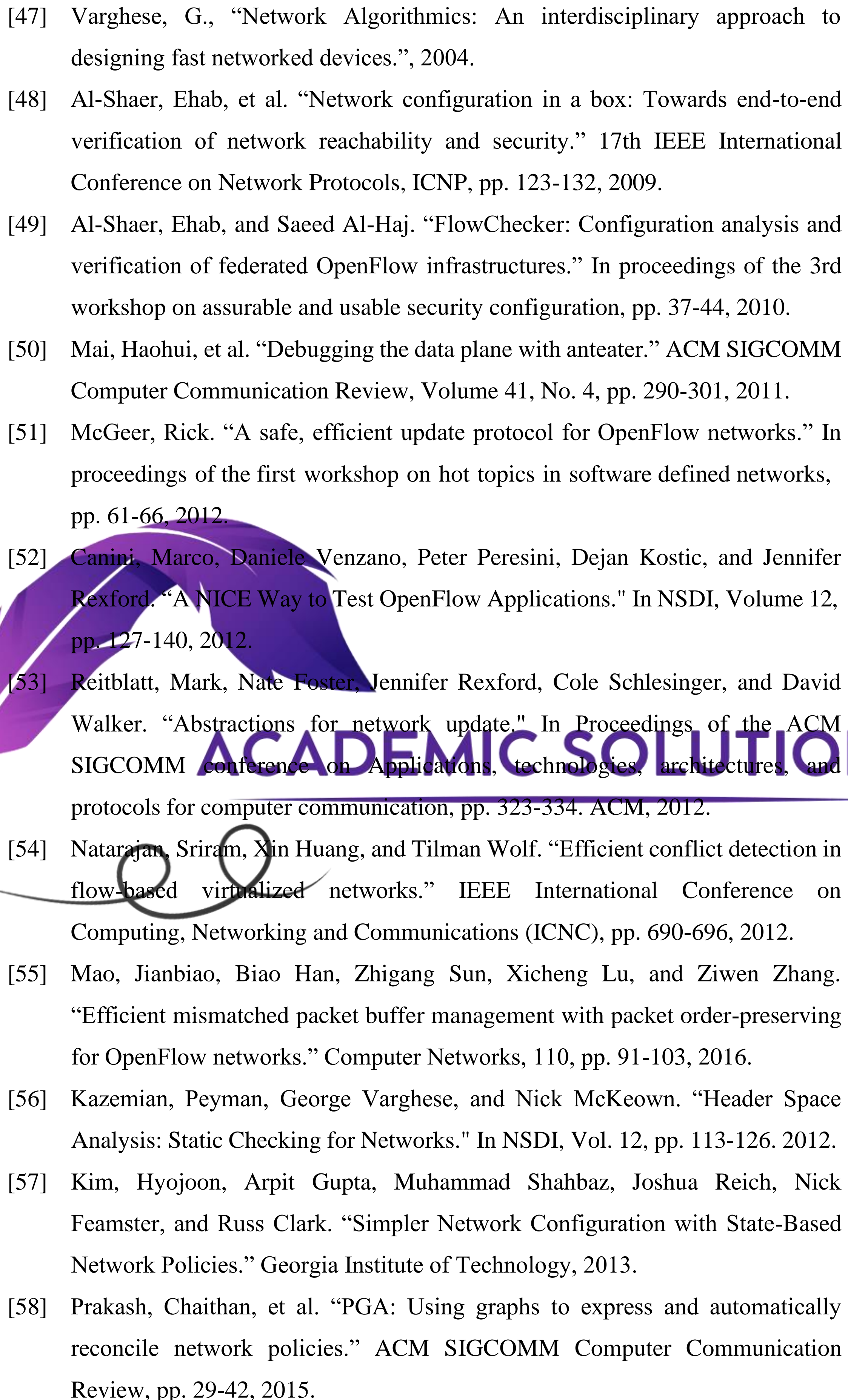
[49] Al-Shaer, Ehab, and Saeed Al-Haj. “FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures.” In proceedings of the 3rd workshop on assurable and usable security configuration, pp. 37-44, 2010.

[50] Mai, Haohui, et al. “Debugging the data plane with anteater.” ACM SIGCOMM Computer Communication Review, Volume 41, No. 4, pp. 290-301, 2011.

[51] McGeer, Rick. “A safe, efficient update protocol for OpenFlow networks.” In proceedings of the first workshop on hot topics in software defined networks, pp. 61-66, 2012.

[52] Canini, Marco, Daniele Venzano, Peter Peresini, Dejan Kostic, and Jennifer Rexford. “A NICE Way to Test OpenFlow Applications." In NSDI, Volume 12, pp. 127-140, 2012.

[53] Reitblatt, Mark, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. “Abstractions for network update." In Proceedings of the ACM SIGCOMM conference on Applications, technologies, architectures, and protocols for computer communication, pp. 323-334. ACM, 2012.



[54] Natarajan, Sriram, Xin Huang, and Tilman Wolf. “Efficient conflict detection in flow-based virtualized networks.” IEEE International Conference on Computing, Networking and Communications (ICNC), pp. 690-696, 2012.

[55] Mao, Jianbiao, Biao Han, Zhigang Sun, Xicheng Lu, and Ziwen Zhang. “Efficient mismatched packet buffer management with packet order-preserving for OpenFlow networks.” Computer Networks, 110, pp. 91-103, 2016.

[56] Kazemian, Peyman, George Varghese, and Nick McKeown. “Header Space Analysis: Static Checking for Networks." In NSDI, Vol. 12, pp. 113-126. 2012.

[57] Kim, Hyojoon, Arpit Gupta, Muhammad Shahbaz, Joshua Reich, Nick Feamster, and Russ Clark. “Simpler Network Configuration with State-Based Network Policies.” Georgia Institute of Technology, 2013.

[58] Prakash, Chaithan, et al. “PGA: Using graphs to express and automatically reconcile network policies.” ACM SIGCOMM Computer Communication Review, pp. 29-42, 2015.

[59] Handigol, Nikhil, et al. “Aster* x: Load-balancing web traffic over wide-area networks.” Open Networking Summit Demo, 2011.

[60] Valenza, Fulvio, Serena Spinoso, and Riccardo Sisto. "Formally specifying and checking policies and anomalies in service function chaining." Journal of Network and Computer Applications, Volume 46, 102419, 2019.

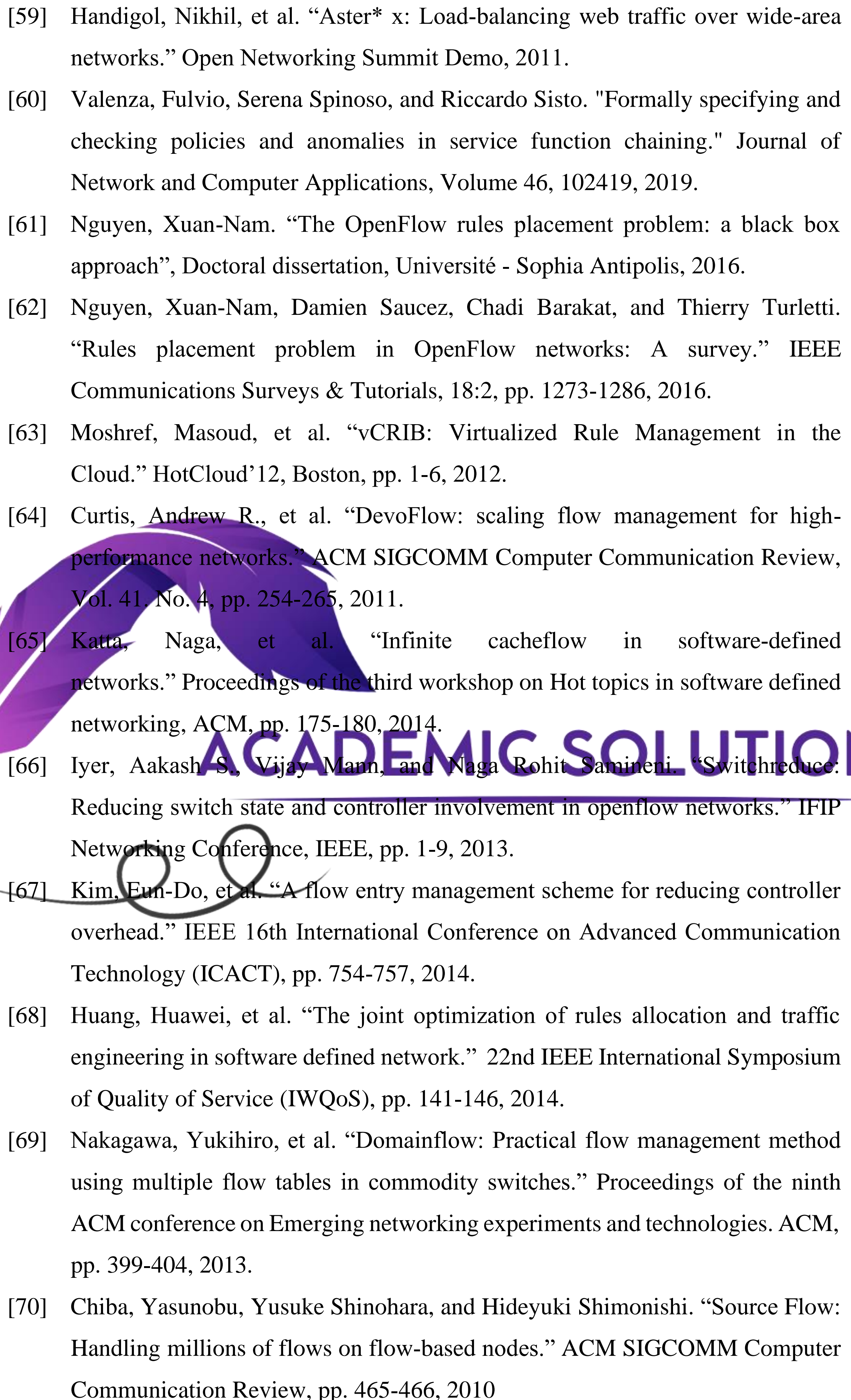
[61] Nguyen, Xuan-Nam. “The OpenFlow rules placement problem: a black box approach”, Doctoral dissertation, Université - Sophia Antipolis, 2016.

[62] Nguyen, Xuan-Nam, Damien Saucez, Chadi Barakat, and Thierry Turletti. “Rules placement problem in OpenFlow networks: A survey.” IEEE Communications Surveys & Tutorials, 18:2, pp. 1273-1286, 2016.

[63] Moshref, Masoud, et al. “vCRIB: Virtualized Rule Management in the Cloud.” HotCloud’12, Boston, pp. 1-6, 2012.

[64] Curtis, Andrew R., et al. “DevoFlow: scaling flow management for high-performance networks.” ACM SIGCOMM Computer Communication Review, Vol. 41, No. 4, pp. 254-265, 2011.

[65] Katta, Naga, et al. “Infinite cacheflow in software-defined networks.” Proceedings of the third workshop on Hot topics in software defined networking, ACM, pp. 175-180, 2014.



[66] Iyer, Aakash S., Vijay Mann, and Naga Rohit Samineni. “Switchreduce: Reducing switch state and controller involvement in openflow networks.” IFIP Networking Conference, IEEE, pp. 1-9, 2013.

[67] Kim, Eun-Do, et al. “A flow entry management scheme for reducing controller overhead.” IEEE 16th International Conference on Advanced Communication Technology (ICACT), pp. 754-757, 2014.

[68] Huang, Huawei, et al. “The joint optimization of rules allocation and traffic engineering in software defined network.” 22nd IEEE International Symposium of Quality of Service (IWQoS), pp. 141-146, 2014.

[69] Nakagawa, Yukihiro, et al. “Domainflow: Practical flow management method using multiple flow tables in commodity switches.” Proceedings of the ninth ACM conference on Emerging networking experiments and technologies. ACM, pp. 399-404, 2013.

[70] Chiba, Yasunobu, Yusuke Shinohara, and Hideyuki Shimonishi. “Source Flow: Handling millions of flows on flow-based nodes.” ACM SIGCOMM Computer Communication Review, pp. 465-466, 2010

[71] Sanabria-Russo, Luis, Jesus Alonso-Zarate, and Christos Verikoukis. "SDN-Based Pro-Active Flow Installation Mechanism for Delay Reduction in IoT." In IEEE Global Communications Conference (GLOBECOM), pp. 1-6, 2018.

[72] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker, "Rethinking enterprise network control. IEEE/ACM Transactions on Networking (TON).", 17:4, pp. 1270-1283, 2009.

[73] 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, pp. 69-74, 2008.

[74] M. Moshref, M. Yu, A. Sharma, and R. Govindan, "Scalable rule management for data center.", In Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation, pp. 157-170, 2013.

[75] Yan, Bo, Yang Xu, Hongya Xing, Kang Xi, and H. Jonathan Chao, "Cab: A reactive wildcard rule caching system for software-defined networks.", In Proceedings of the third workshop on Hot topics in software defined networking, pp. 163-168. ACM, 2014.

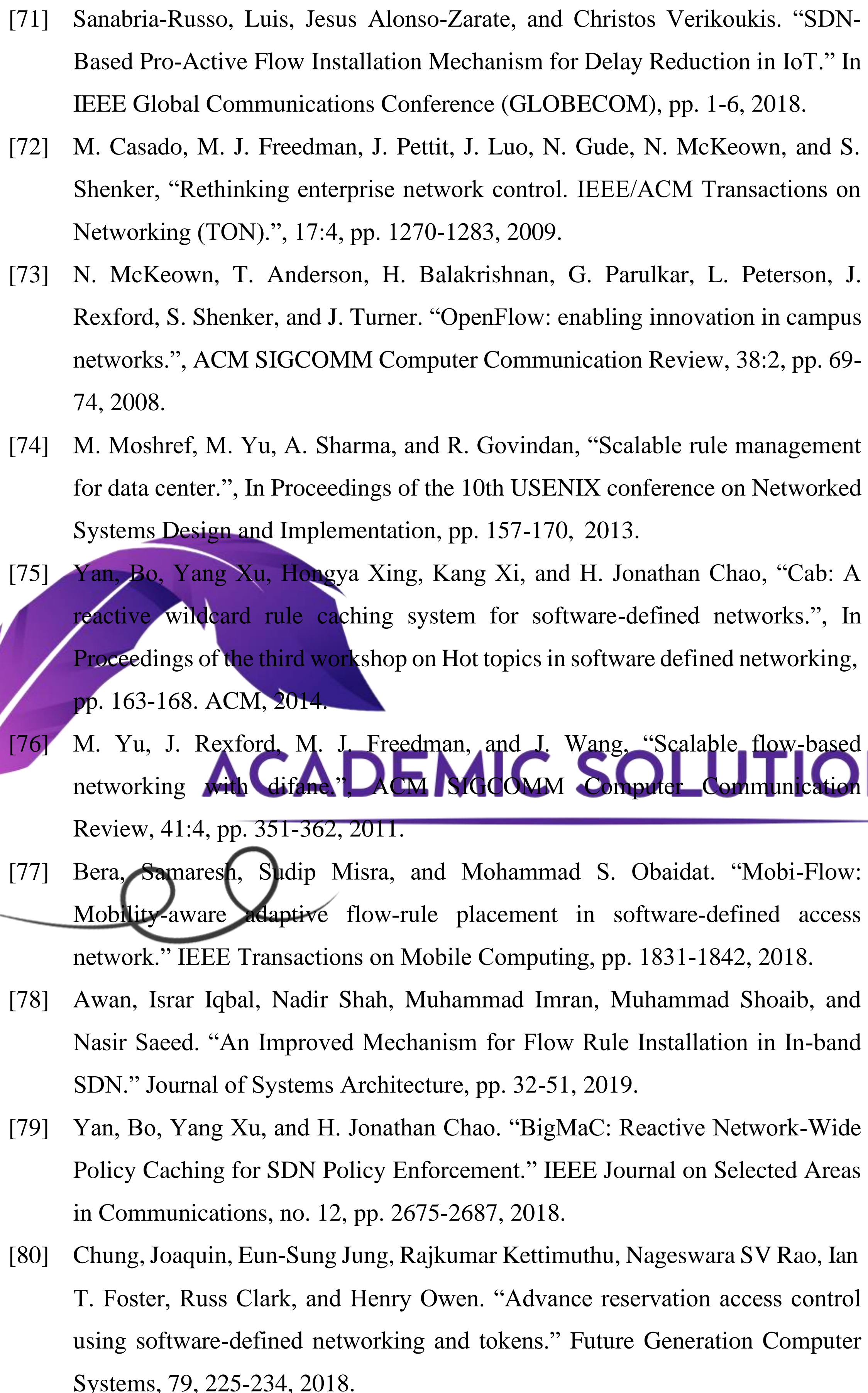
[76] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane.", ACM SIGCOMM Computer Communication Review, 41:4, pp. 351-362, 2011.

[77] Bera, Samaresh, Sudip Misra, and Mohammad S. Obaidat. "Mobi-Flow: Mobility-aware adaptive flow-rule placement in software-defined access network." IEEE Transactions on Mobile Computing, pp. 1831-1842, 2018.

[78] Awan, Israr Iqbal, Nadir Shah, Muhammad Imran, Muhammad Shoaib, and Nasir Saeed. "An Improved Mechanism for Flow Rule Installation in In-band SDN." Journal of Systems Architecture, pp. 32-51, 2019.

[79] Yan, Bo, Yang Xu, and H. Jonathan Chao. "BigMaC: Reactive Network-Wide Policy Caching for SDN Policy Enforcement." IEEE Journal on Selected Areas in Communications, no. 12, pp. 2675-2687, 2018.

[80] Chung, Joaquin, Eun-Sung Jung, Rajkumar Kettimuthu, Nageswara SV Rao, Ian T. Foster, Russ Clark, and Henry Owen. "Advance reservation access control using software-defined networking and tokens." Future Generation Computer Systems, 79, 225-234, 2018.



[81] Panda, Aurojit, et al. "Verifying Reachability in Networks with Mutable Data paths." NSDI, 2017.

[82] Heller, Brandon, et al. "Leveraging SDN layering to systematically troubleshoot networks." In Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking, pp. 37-42. ACM, 2013.

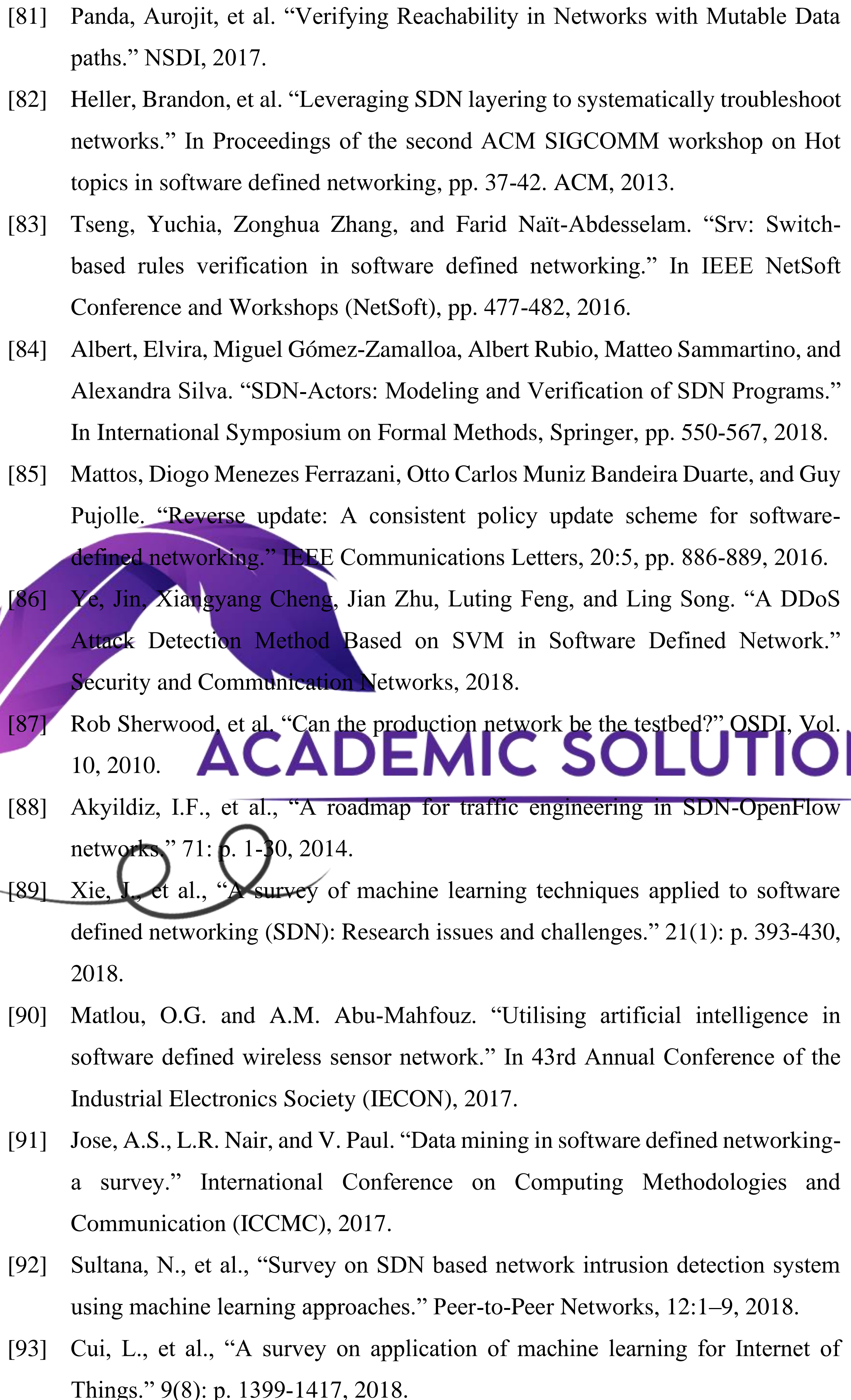
[83] Tseng, Yuchia, Zonghua Zhang, and Farid Naït-Abdesselam. "Srv: Switch-based rules verification in software defined networking." In IEEE NetSoft Conference and Workshops (NetSoft), pp. 477-482, 2016.

[84] Albert, Elvira, Miguel Gómez-Zamalloa, Albert Rubio, Matteo Sammartino, and Alexandra Silva. "SDN-Actors: Modeling and Verification of SDN Programs." In International Symposium on Formal Methods, Springer, pp. 550-567, 2018.

[85] Mattos, Diogo Menezes Ferrazani, Otto Carlos Muniz Bandeira Duarte, and Guy Pujolle. "Reverse update: A consistent policy update scheme for software-defined networking." IEEE Communications Letters, 20:5, pp. 886-889, 2016.

[86] Ye, Jin, Xiangyang Cheng, Jian Zhu, Luting Feng, and Ling Song. "A DDoS Attack Detection Method Based on SVM in Software Defined Network." Security and Communication Networks, 2018.

[87] Rob Sherwood, et al. "Can the production network be the testbed?" OSDI, Vol. 10, 2010.



[88] Akyildiz, I.F., et al., "A roadmap for traffic engineering in SDN-OpenFlow networks." 71: p. 1-30, 2014.

[89] Xie, J., et al., "A survey of machine learning techniques applied to software defined networking (SDN): Research issues and challenges." 21(1): p. 393-430, 2018.

[90] Matlou, O.G. and A.M. Abu-Mahfouz. "Utilising artificial intelligence in software defined wireless sensor network." In 43rd Annual Conference of the Industrial Electronics Society (IECON), 2017.

[91] Jose, A.S., L.R. Nair, and V. Paul. "Data mining in software defined networking-a survey." International Conference on Computing Methodologies and Communication (ICCMC), 2017.

[92] Sultana, N., et al., "Survey on SDN based network intrusion detection system using machine learning approaches." Peer-to-Peer Networks, 12:1–9, 2018.

[93] Cui, L., et al., "A survey on application of machine learning for Internet of Things." 9(8): p. 1399-1417, 2018.

[94] Le, L.-V., et al., Applying Big Data, Machine Learning, and SDN/NFV for 5G Early-Stage Traffic Classification and Network QoS Control, 6(2): p. 36, 2018.

[95] Kreutz, Diego., et al., "Software-defined networking: A comprehensive survey." Proceedings of the IEEE , no. 1: 14-76, 2014.

[96] Wang, X., X. Li, and V.C. Leung, "Artificial intelligence-based techniques for emerging heterogeneous network: State of the arts, opportunities, and challenges." IEEE Access, 3: p. 1379-1391, 2015.

[97] Jamshidi, S., "The Applications of Machine Learning Techniques in Networking." 2019.

[98] Mohammed, A., et al., "Machine Learning and Deep Learning Based Traffic Classification and Prediction in Software Defined Networking." IEEE International Symposium on Measurements & Networking (M&N), 1-6, 2019.

[99] Nguyen, T.N., "The challenges in SDN/ML based network security: A survey." arXiv preprint arXiv:1804.03539, 2018.

[100] Clark, D.D., et al., "A knowledge plane for the internet." In Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications, 2003.

[101] Mishra, Ajay Kumar., "SDN Based Network Management for Enhancing Network Security." Diss. Dublin, National College of Ireland, 2020.

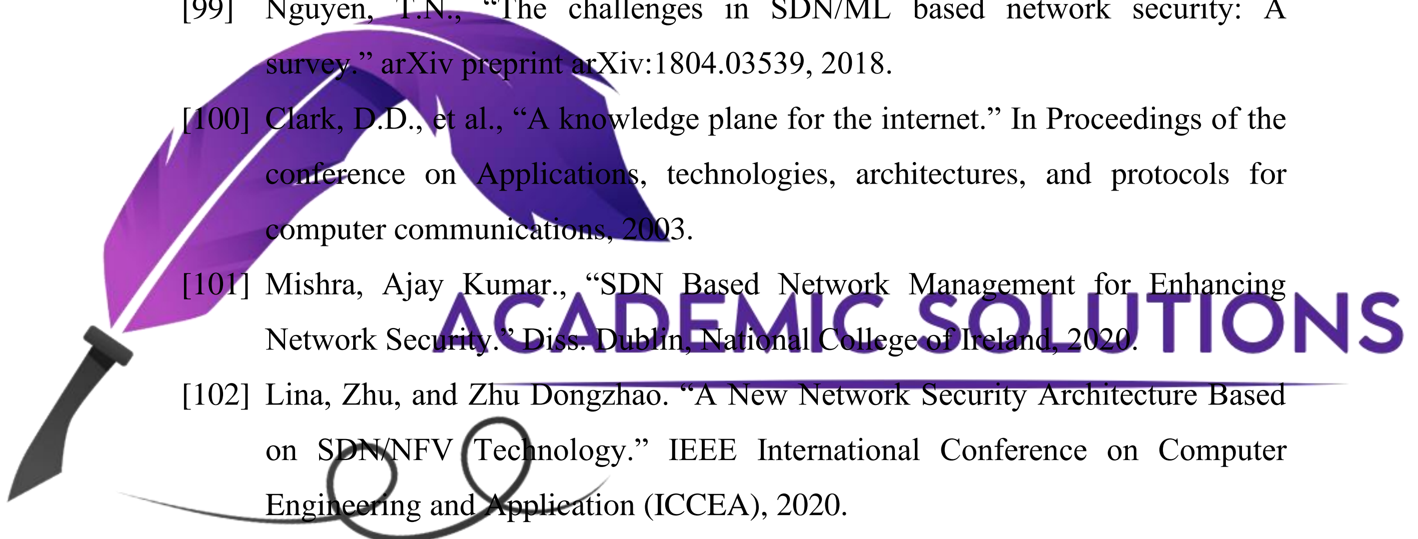
[102] Lina, Zhu, and Zhu Dongzhao. "A New Network Security Architecture Based on SDN/NFV Technology." IEEE International Conference on Computer Engineering and Application (ICCEA), 2020.

[103] Mudassar Hussain, and Nadir Shah. "Automatic rule installation in case of policy change in software defined networks." Telecommunication Systems, 68 (3), pp. 461-477, 2018.

[104] Mudassar Hussain, Nadir Shah, and Ali Tahir. "Graph-Based Policy Change Detection and Implementation in SDN.", Electronics, 8.10, pp. 1136, 2019.

[105] Oracle Virtual Box. [Online]. Available: https://www.virtualbox.org/ [Accessed: August 10, 2019]

[106] IBM RackSwitch G8264 Command Reference: Pages 103-107 [Online]. Available: http://www-01.ibm.com/support/docview.wss?uid=isg3T7000600&aid=1 [Accessed: Feb 10, 2019]



[107] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. "Serverswitch: A programmable and high-performance platform for data center networks." NSDI, pp. 2-2, 2011.

[108] Agrawal, Banit, and Timothy Sherwood. "Modeling TCAM power for next generation network devices." In ISPASS, pp. 120-129. 2006.

[109] Luo, Layong, Gaogang Xie, Steve Uhlig, Laurent Mathy, Kavé Salamatian, and Yingke Xie. "Towards TCAM-based scalable virtual routers." In Proceedings of the 8th international conference on Emerging networking experiments and technologies, pp. 73-84. ACM, 2012.

[110] Draves, Richard, Christopher King, Srinivasan Venkatachary, and Brian D. Zill. "Constructing optimal IP routing tables." In Infocom, vol. 99, pp. 88-97. 1999.

[111] Braun, Wolfgang, and Michael Menth. "Wildcard Compression of Inter-Domain Routing Tables for OpenFlow-Based Software-Defined Networking." EWSDN, pp. 25-30, 2014.

[112] Leng, Bing, Liusheng Huang, Chunming Qiao, Hongli Xu, and Xinglong Wang. "FTRS: A mechanism for reducing flow table entries in software defined networks." Computer Networks 122: 1-15, 2017.

[113] Vishnoi, Anilkumar, Rishabh Poddar, Vijay Mann, and Suparna Bhattacharya. "Effective switch memory management in OpenFlow networks." In Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, pp. 177-188. ACM, 2014.

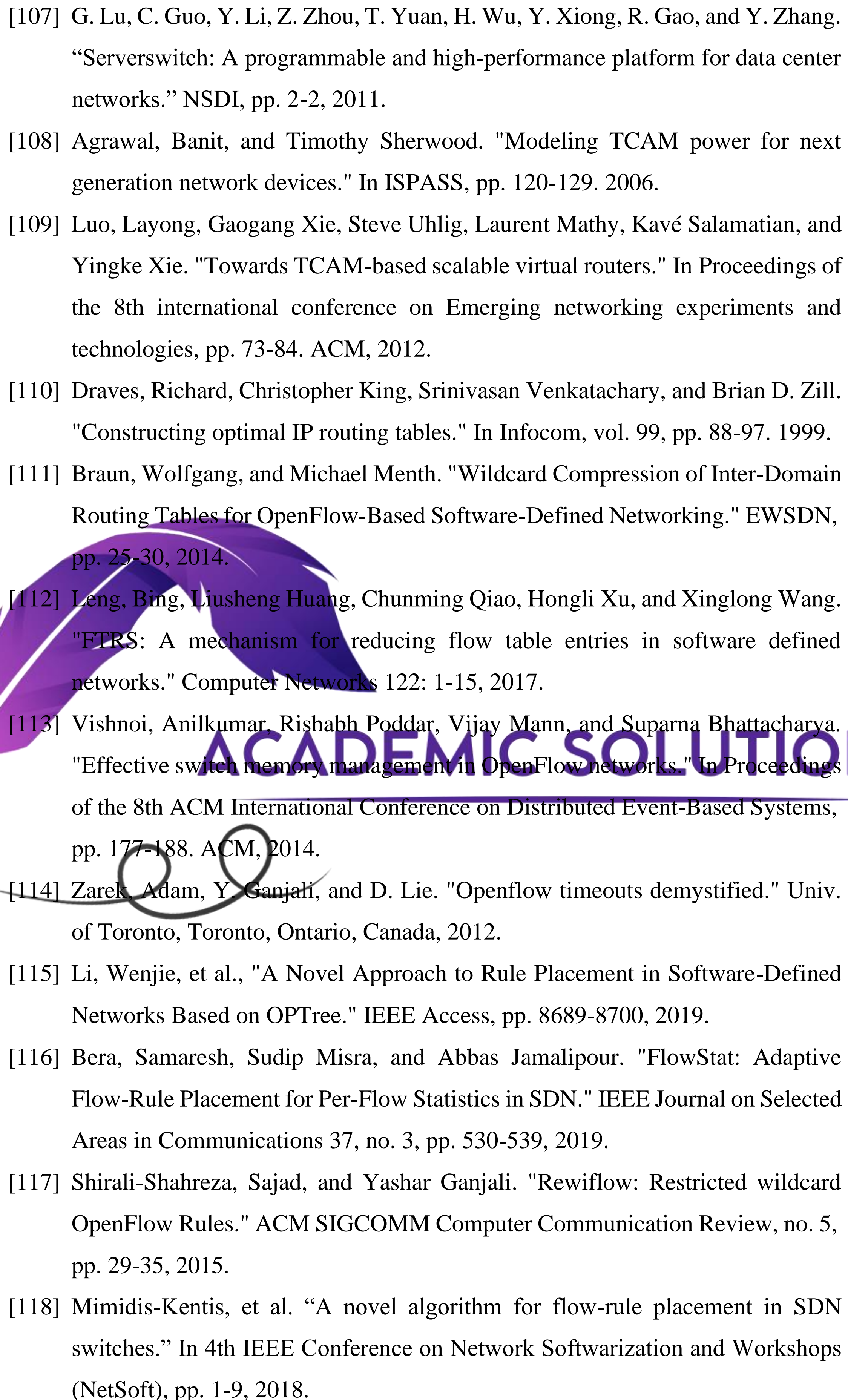
[114] Zarek, Adam, Y. Ganjali, and D. Lie. "Openflow timeouts demystified." Univ. of Toronto, Toronto, Ontario, Canada, 2012.

[115] Li, Wenjie, et al., "A Novel Approach to Rule Placement in Software-Defined Networks Based on OPTree." IEEE Access, pp. 8689-8700, 2019.

[116] Bera, Samaresh, Sudip Misra, and Abbas Jamalipour. "FlowStat: Adaptive Flow-Rule Placement for Per-Flow Statistics in SDN." IEEE Journal on Selected Areas in Communications 37, no. 3, pp. 530-539, 2019.

[117] Shirali-Shahreza, Sajad, and Yashar Ganjali. "Rewiflow: Restricted wildcard OpenFlow Rules." ACM SIGCOMM Computer Communication Review, no. 5, pp. 29-35, 2015.

[118] Mimidis-Kentis, et al. "A novel algorithm for flow-rule placement in SDN switches." In 4th IEEE Conference on Network Softwarization and Workshops (NetSoft), pp. 1-9, 2018.



[119] Rottenstreich, Ori. "Lossy compression of packet classifiers." In Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems, IEEE Computer Society, pp. 39-50, 2015.

[120] M. Monaco, O. Michel, and E. Keller, “Applying Operating System Principles to SDN Controller Design,” In Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks, p. 2, 2013.

[121] F. Durr, T. Kohler, J. Grunert, and A. Kutzleb, “Zerosdn: A message bus for flexible and light-weight network control distribution in SDN.” University of Stuttgart, Institute of Parallel and Distributed Systems (IPVS), 2016.

[122] Kim, Hyojoon, and Nick Feamster. "Improving network management with software defined networking." IEEE Communications Magazine, pp: 114-119, 2013.

[123] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms. MIT press, 2009.

[124] Karger, David, et al., “Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web.” In Proceedings of the 29th annual ACM symposium on Theory of computing, pp. 654-663, 1997.

[125] Open Networking Foundation, “OpenFlow switch specification.”, Version 1.5.0, 2015.

[126] Lee, Jeongkeun, et al. "Network policy whiteboarding and composition." ACM SIGCOMM Computer Communication Review, 45:4, pp. 373-374, 2015.

[127] Sammoud, Olfa, Christine Solnon, and Khaled Ghédira. "Ant algorithm for the graph matching problem." EvoCOP. 2005.

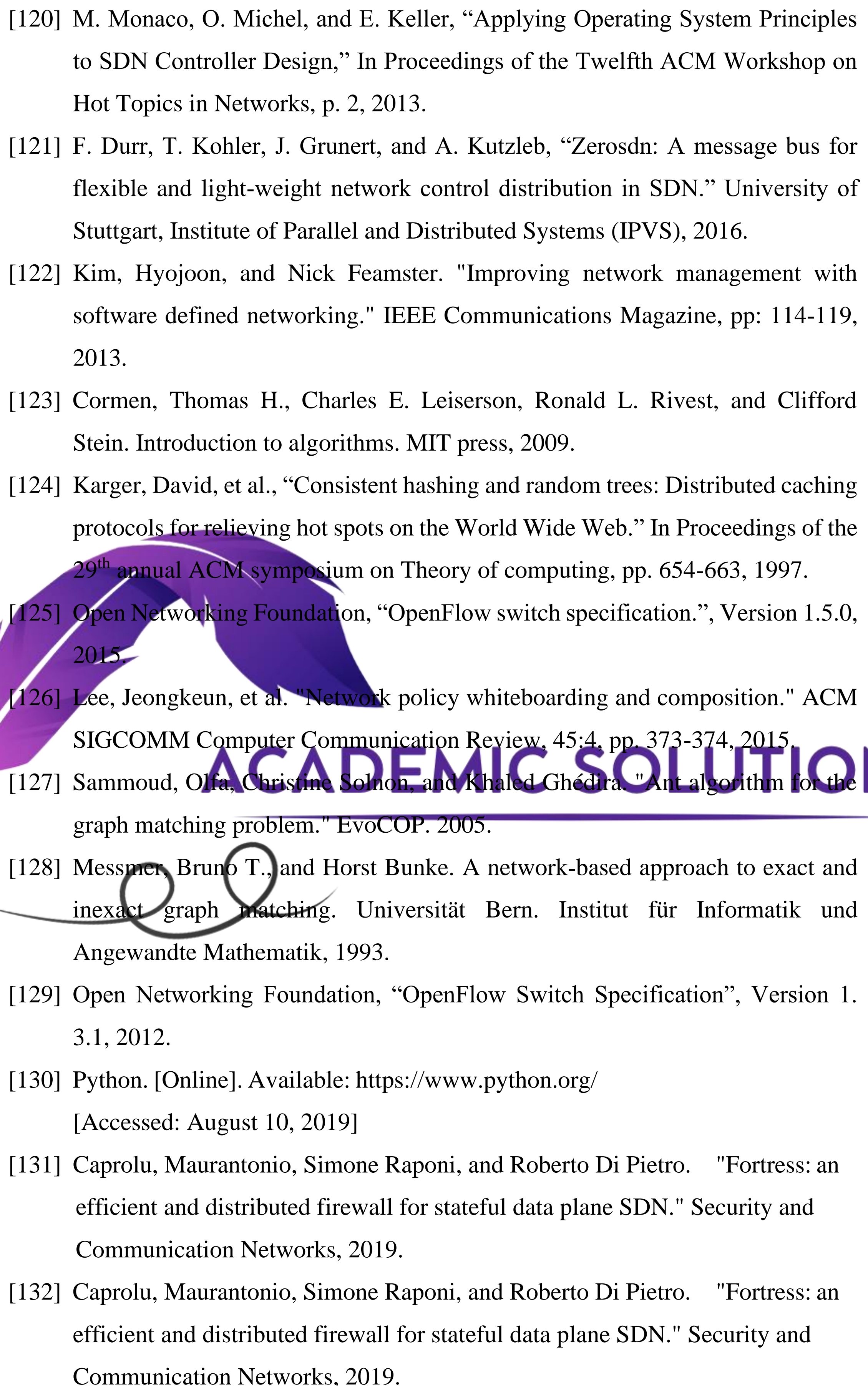
[128] Messmer, Bruno T., and Horst Bunke. A network-based approach to exact and inexact graph matching. Universität Bern. Institut für Informatik und Angewandte Mathematik, 1993.

[129] Open Networking Foundation, “OpenFlow Switch Specification”, Version 1. 3.1, 2012.

[130] Python. [Online]. Available: https://www.python.org/ [Accessed: August 10, 2019]

[131] Caprolu, Maurantonio, Simone Raponi, and Roberto Di Pietro. "Fortress: an efficient and distributed firewall for stateful data plane SDN." Security and Communication Networks, 2019.

[132] Caprolu, Maurantonio, Simone Raponi, and Roberto Di Pietro. "Fortress: an efficient and distributed firewall for stateful data plane SDN." Security and Communication Networks, 2019.



[133] Kim, Sunghwan, et al. "Secure Collecting, Optimizing, and Deploying of Firewall Rules in Software-Defined Networks." IEEE Access, 8:15166-15177, 2020.

[134] Hu, Hongxin, et al. "Towards a reliable firewall for software-defined networks." Computers & Security, 87: 101597, 2019.

[135] Flauzac, Olivier, et al. "SDN Architecture to prevent attacks with OpenFlow.", 8th IEEE International Conference on Wireless Networks and Mobile Communications (WINCOM), pp. 1-6, 2020.

[136] Shankar, Ganesh Handige, and V. V. Deepthi. "Method and system for debugging in a software-defined networking (SDN) system." U.S. Patent No. 10,243,778. 26 Mar. 2019.

[137] Yang, Ze, and Kwan L. Yeung. "Flow monitoring scheme design in SDN", Computer Networks, 167:107007, 2020.

[138] Shen, Shan-Hsiang. "An efficient network monitor for SDN networks." ACM SIGMETRICS Performance Evaluation Review 46.2, pp. 95-96, 2019.

[139] Amiri, Esmaeil, Emad Alizadeh, and Khalilollah Raeisi. “An efficient hierarchical distributed SDN controller model.” 5th IEEE Conference on Knowledge Based Engineering and Innovation (KBEI), pp. 553-557, 2019.

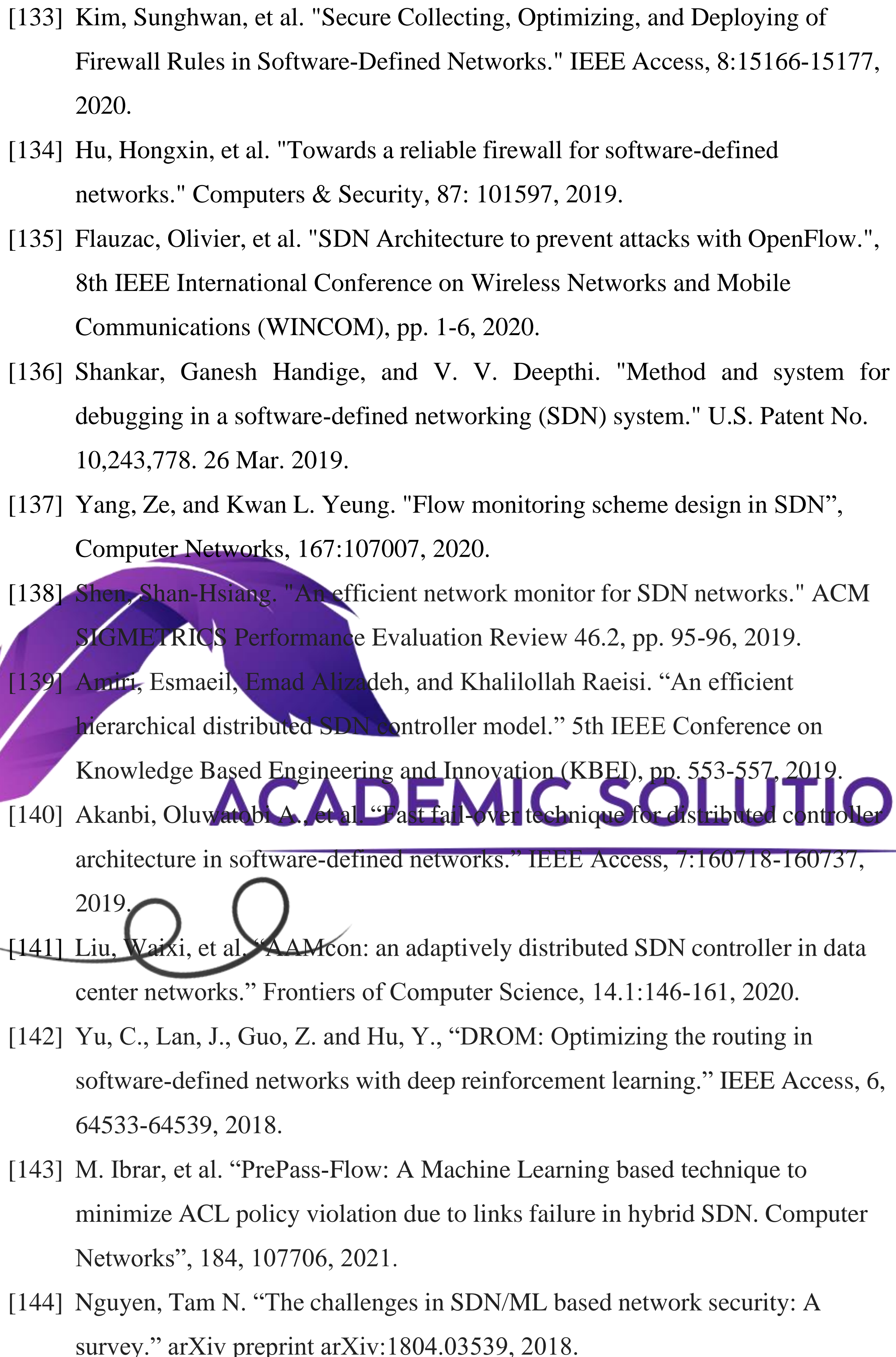
[140] Akanbi, Oluwatobi A., et al. “Fast fail-over technique for distributed controller architecture in software-defined networks.” IEEE Access, 7:160718-160737, 2019.

[141] Liu, Waixi, et al. “AAMcon: an adaptively distributed SDN controller in data center networks.” Frontiers of Computer Science, 14.1:146-161, 2020.

[142] Yu, C., Lan, J., Guo, Z. and Hu, Y., “DROM: Optimizing the routing in software-defined networks with deep reinforcement learning.” IEEE Access, 6, 64533-64539, 2018.

[143] M. Ibrar, et al. “PrePass-Flow: A Machine Learning based technique to minimize ACL policy violation due to links failure in hybrid SDN. Computer Networks”, 184, 107706, 2021.

[144] Nguyen, Tam N. “The challenges in SDN/ML based network security: A survey.” arXiv preprint arXiv:1804.03539, 2018.



Appendix A

ACADEMIC SOLUTIONS

Research Publications and Research Projects

A. Research Publications

The following research publications are part of thesis:

i. Mudassar Hussain, and Nadir Shah. "Automatic rule installation in case of policy change in software defined networks." Telecommunication Systems 68.3:461-477, 2018.

ii. Mudassar Hussain, Nadir Shah, and Ali Tahir. "Graph-Based Policy Change Detection and Implementation in SDN." Electronics 8.10: 1136, 2019.

iii. Mudassar Hussain, et al. "A Comprehensive Survey of Existing Approaches of Software Defined Networking" (Submitted).

B. Research Projects

i. This research work is partially funded by Higher Education Commission under NRPU research project number "5372/Federal/NRPU /R&D/HEC/2016", entitled "Fault Localization in Term of Security and Management Policy in Software Defined Networks" of 3.9 Million PKR.