Study on Network Property Verification in Software Defined Networking

September　２０２０

ZHANG　YICONG

Study on Network Property Verification in Software Defined Networking

Graduate School of Systems and Information Engineering

University of Tsukuba

September ２０２０

ZHANG　YICONG

Abstract

Due to the network complexity, networks are unlikely to be bug-free. Automatic tools will make network management easier for network administrators. Software-Defined Networking (SDN) is an innovational network architecture which gives network administrators the ability to directly control the whole network by programming on a centralized controller. In this study, we propose serval methods of fast verification for SDN networks. We implement our method and conduct experiments. The experimental results show that the proposed method can efficiently and accurately detect serval network properties, such as loops, black holes.

Contents

Abstract i

List of Figures v

List of Tables vii

1 Introduction 1
- 1.1 Introduction 1
- 1.2 Our Contribution 3
- 1.3 Structure of Thesis 3

2 Background 4
- 2.1 Softwared Defined Networking 4
- 2.2 OpenFlow 6
  - 2.2.1 OpenFlow Switch 6
  - 2.2.2 OpenFlow Controller 8
- 2.3 Apache Spark 8

3 Related Works 11
  - 3.0.1 Detecting Network Errors 11
  - 3.0.2 Attacks from SDN application 13

4 Problem Statement 14
- 4.1 Loop-freedom and Nonexistence of Black Holes 14
- 4.2 Firewall and Bypass Threat in OpenFlow Networks 14
  - 4.2.1 Firewall in OpenFlow Networks 14
  - 4.2.2 Packet Modification in OpenFlow 15
  - 4.2.3 The Firewall Bypass Threat in OpenFlow Networks 17
  - 4.2.4 System Model and Assumptions 19

5 A Novel Method against the Firewall Bypass Threat in OpenFlow Networks 20
- 5.1 Violation Detection 20
  - 5.1.1 Flow path in OpenFlow networks 20
  - 5.1.2 Forwarding Graph 21
  - 5.1.3 Firewall Deny Space and Firewall Policy Violation 23
- 5.2 Violation Resolution 23
  - 5.2.1 Initialization 23

5.2.2 Update the Forwarding Graph . . . 24
5.2.3 Update the Firewall Deny Space . . . 25
5.3 Implementation . . . 25
5.3.1 Implementation Environment . . . 25
5.3.1.1 Mininet . . . 25
5.3.1.2 Floodlight . . . 26
5.3.2 Implementation . . . 27
5.4 Performance Evaluation . . . 28
5.5 Conclusion . . . 31

6 Atomic Predicates Based Data Plane Properties Verification in Software Defined Networking Using Spark 32
6.1 Proposed Network Model . . . 33
6.1.1 Network Packet . . . 33
6.1.2 Forwarding Graph . . . 36
6.1.3 Atomic Predicates . . . 38
6.2 Proposed Data Plane Verification Scheme . . . 40
6.2.1 Preprocessing . . . 41
6.2.2 Computing Atomic Predicates with Spark . . . 43
6.2.3 Building Forwarding Graph . . . 44
6.2.4 Computing the Reachability . . . 45
6.2.5 Application Cases . . . 46
6.2.5.1 Firewall Model . . . 46
6.2.5.2 Problem Definition . . . 47
6.2.5.3 Verification with Reachability Tree . . . 48
6.2.6 Dynamic Update . . . 49
6.3 Implementation and Performance Evaluation . . . 50
6.4 Conclusion . . . 54

7 Atomic Predicate Based Network Property Verification for SDN Networks 56
7.1 Network Model and Problem Statement . . . 56
7.1.1 Network Model . . . 56
7.1.2 Predicate . . . 57
7.2 Problem Statement . . . 58
7.2.1 Atomic Predicate . . . 59
7.2.2 AP tree . . . 60
7.2.3 Loop-freedom and Nonexistence of Black Holes . . . 61
7.3 Proposed Method . . . 61
7.3.1 Preprocessing . . . 61
7.3.2 Computing Atomic Predicates . . . 62
7.3.3 AP Tree Construction . . . 63
7.3.4 Computing Packet Reachability . . . 64
7.3.5 Updating . . . 65
7.4 Implementation and Performance Evaluation . . . 66
7.5 Conclusion . . . 69

8 Conclusion **70**
8.1 Conclusion 70

Acknowledgements **71**

Bibliography **72**

List of Publications **76**

List of Figures

2.1 A high-level overview of the SDN architecture. 5
2.2 OpenFlow switch specification. 6

4.1 Inserting one flow entry into the switch's flow table to bypass an SDN firewall. 18
4.2 Inserting three flow entries into the three switches's flow tables to bypass an SDN firewall. 18

5.1 All flow paths form a directed graph. 21
5.2 Forwarding Graph of the network in Fig. 5.1. 22
5.3 Implementation frame of our method. 27
5.4 Flowchart of our method. 28
5.5 Example scenario for performance comparison. 29
5.6 Delay of evaluating a variety number of flow entries against firewall policies. 30
5.7 As more and more rule insert requests are evaluated, the delay are increasing but still remaining in a acceptable range. 31

6.1 An example of BDD subgraphs representing a prefix match field $100x$. . . 35
6.2 An example of the box model 36
6.3 An example of forwarding graph. 37
6.4 An example of forwarding graph with atomic predicates. 40
6.5 Flowchart of preprocessing with Spark. 43
6.6 Flowchart of computing atomic predicates set with Spark. 44
6.7 An example of a reachability tree. 47
6.8 A topology example of the datasets. 51
6.9 Computation time (ms) of initialization. 52
6.10 Computation time (ms) of reachablity computation. 52
6.11 Computation time (ms) of computing atomic predicates. 52
6.12 Computation time (ms) of detecting firewall violation computing time. . . 54

7.1 (a) Three predicates. (b) The packet header space and five atomic predicates. (c) A sample network including the three predicates. 58
7.2 A small network consists two switches. 62
7.3 Flow chart of preprocessing with Spark. 62
7.4 Flow chart of computing atomic predicates set with Spark. 63
7.5 AP Tree of predicates in Fig. 7.1 (b). (a) Original AP Tree. (b) Pruned AP Tree. (c) Optimized AP Tree. 63
7.6 Computing reachability for packets specified by a_4.(a) Normal path. (b) Loop. (c) Black hole. 65

7.7 A topology example of the datasets. 67
7.8 Computation time (ms) of initialization. 67
7.9 Computation time (ms) of reachablity computation. 68
7.10 Computation time (ms) of computing atomic predicates. 68

List of Tables

2.1 Components of a flow entry in a flow table. 7

4.1 Set-Field actions [1]. 17

5.1 Specifications of the environment we used. 25
5.2 Features comparison. 29

6.1 Experimental Datasets Information. 50
6.2 Storage overheads of reachability trees of all ports (bytes). 53

7.1 Experimental Datasets Information. 66

Chapter 1

Introduction

In this Chapter, we first briefly introduce our research work. After the introduction, we present our goal in our research and the structure of this thesis.

1.1 Introduction

As an innovational network architecture, Software-Defined Networking (SDN) provides us many valuable advantages over the traditional network architecture, especially the ability to easily manage the network. The basic concept of SDN is to build a network operation system which enables network administrators to manage and control the whole network traffic by programming on a controller software, without manually configuring each device. As a result, SDN is suitable for nowadays large scale networks applications and Internet of Things (IoT) development [2]. Implementing SDN data centers becomes a new trend for network developing and researching [3].

Unlike the traditional network, implementing routing policies in switches and routers, S-DN decouples data plane and control plane from switches and routers into the networks. In an SDN network, the control plane consist of one or more logically centralized controller, these controllers managing the network data plane by applying routing policies to the switches. Routing policies can be defined by network administrators or automatically created by the controller.

To communicate with and install rules into the switches, a communication interface protocol between the controller and the switches is necessary. As the first standard communications interface defined for SDN, OpenFlow [4] are still commonly used to implement SDN. OpenFlow introduces the OpenFlow controller, the OpenFlow switch and the OpenFlow protocol. The OpenFlow controller enables users to directly control the flow forwarding by installing flow entries into the flow tables of the OpenFlow switches, the OpenFlow switch is also able to ask the controller how to deal with a undefined data flow, all these kinds of interactions are achieved by OpenFlow protocol defined messages.

Nowadays, data centers withstand failures using the principles of scale-out and redundant design. The underlying assumption is that the system reacts correctly to failures [5]. Common routing failures include routing loops and black-holes. Some errors only become visible when a change is made. Meanwhile, network operators make mistakes more than they thought. In a study of Internet services failures [6], operator error was found to be the largest single cause of configuration errors.

To achieve a more reliable network, formal analysis methods and automated reachability analysis tools are needed. This kind of network property verification is supposed to verify network configuration and essential network properties (e.g., loop-free).

Recent work on network property verification focuses on graph based verification method. With a graph-based network model, reachability analysis is transformed into graph problem. Building the model and computing reachability of the model are the core parts of graph based verification method. By *reachability*, we mean that can a packet with specific header values sent from port x reach port y. Network properties is verified by checking reachability. VeriFlow [7] divides rules into equivalence classes (EC) and models the EC network as a graph to detect misconfigurations such as loops. NetPlumber[8] maintains a dependency graph between flow rules based on Header Space Analysis (H-SA) [9] which models packet as a point in a geometric space and networking box as a switch transfer function.

To reduce the verification time, atomic predicate (AP) [10] is introduced to compress Access Control Lists (ACLs) and forwarding rules to a small set of equivalence classes that can be specified efficiently. AP Classifier [11] provides a novel data structure to improve efficiency classifying packets into atomic predicate to analyze packet flow behavior.

In this study, we make our own contribution to model the SDN networks and formalize the problem. Then we propose serval methods of fast verification for SDN networks. We start with the firewall bypass threat. We implement our method and conduct experiments. The result of experiments shows that our methods can compute reachability effectively and efficiently in SDN networks.

1.2 Our Contribution

In this study, our goal is *to design a network property verification method for SDN networks in data center with better performance, e.g., fast analysis time, smaller storage cost.* To achieve the goal, we propose three methods of fast verification for SDN networks. We also addressed the challenge of building a robust firewall for SDN networks by analyzing the specification of packet modification behavior in OpenFlow, and its impact to the traditional firewall deployed in SDN networks. We make our own contribution to formalize and model the SDN network system and the firewall bypass threat, then propose a method to accurately detect and prevent firewall bypass threat in SDN. Finally, we implement our method and conduct experiments. The result of experiments shows that our methods can compute reachability effectively and efficiently in SDN networks.

1.3 Structure of Thesis

This thesis is organized as follows.

- Chapter 1 provides an introduction to our research. In the end, we illustrate the purpose of this thesis.
- Chapter 2 gives a brief, comprehensive introduction to preliminaries to our system.
- Chapter 3 discusses related works.
- Chapter 4 presents a detailed description and analysis of our problem.
- Chapter 5 presents our first proposed method.
- Chapter 6 presents our second proposed method.
- Chapter 7 presents our third proposed method.
- Chapter 8 concludes our work.

Chapter 2

Background

This chapter aims to put readers in the right context. First, we give a brief description of SDN architecture and OpenFlow protocol, then we explain how switches handle data flow by OpenFlow protocol. After that, we give an introduction to big data and Apache Spark.

2.1 Softwared Defined Networking

The principal concept of SDN is to separate the control plane and data plane. Therefore, the switches and routers are no longer to decide how to process the packets by their routing protocols, but only acting as forwarding devices, the decisions are made by a logically centralized controller. As a direct advantage, network administrators are free from manually configuring each individual network device, now they can dynamically manage and change the network traffic by a controller software located in their computers or the server. Furthermore, SDN gives more flexibility to the administrators in applying routing policies than traditional network by letting the controller access to the information of the whole network, like the network topology, the state of the traffic load, and the information of the switches and links. For example, the administrator can easily adjust the traffic patterns to different types of flow via the controller.

A high-level overview of a basic SDN architecture is shown in Fig. 2.1, which comprise three different layers. The SDN applications exist in the application layer, they are programs that communicate their network requirements to the SDN controller and use a abstracted view of the network provide by the controller to make their internal decisions and other opeartions. Typical examples of SDN applications are load balancers and network monitors. Other applications include security application, which is the

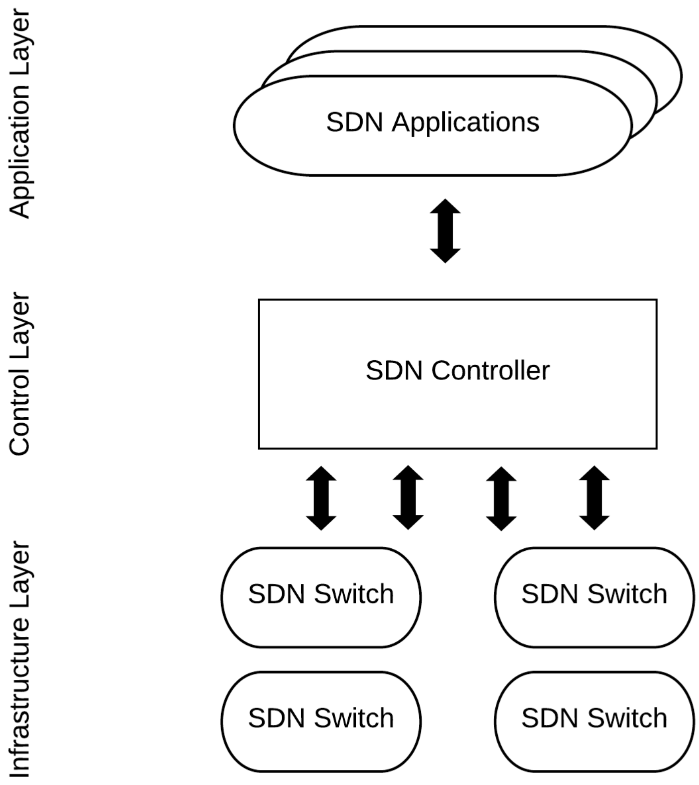

FIGURE 2.1: A high-level overview of the SDN architecture.

focus of this paper. The SDN controller exists in the controller layer, it is a logically centralized entity and usually a software in real life. Besides collecting and analyzing the information from the physical network devices to provide the abstract view for the SDN application, the controller is responsible for receiving the requirements form the applications about what they want to apply to the network . The controller translates the requirements to configuration commands for the network devices and apply them. POX[12], Floodlight[13], OpenDayLight[14] and Ryu[15] are some popular open source controllers. In the infrastructure layer, we use SDN switches to reference all SDN network devices. The SDN switches communicate with the controller to perform different control commands, by using communication protocols such like SoftRouter[16], Forwarding and Control Element Separation (ForCES)[17], and OpenFlow.

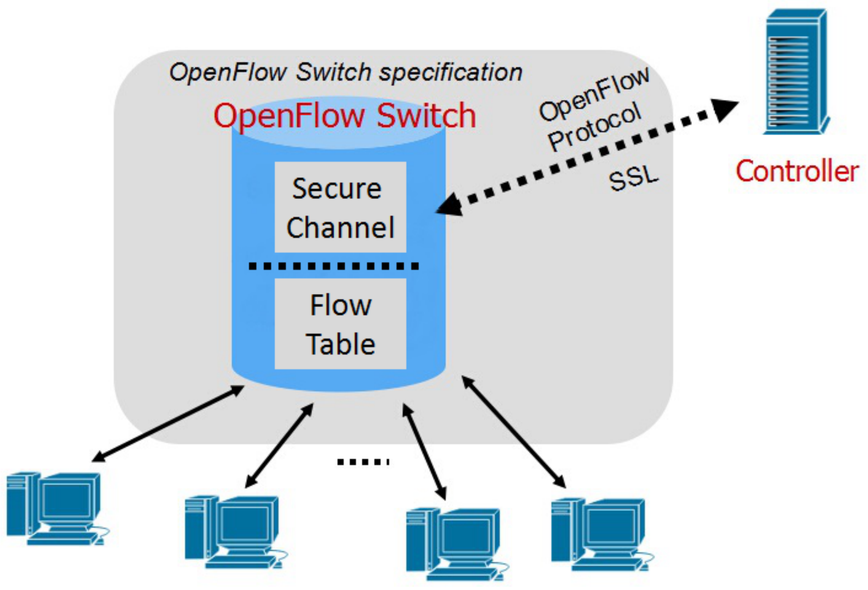

FIGURE 2.2: OpenFlow switch specification.

2.2 OpenFlow

OpenFlow is the first standard communications protocol defined for SDN, and it still is the most commonly used protocol at present. OpenFlow is introduce by the Open Networking Foundation (ONF)[18], by regularly extending OpenFlow's specification, ONF still leads its development. Fig. 2.2 presents the three main components comprised by the architecture of OpenFlow[19]: the data plane consists of OpenFlow swithes and at least one OpenFlow controller exists in the control plane. The control plane is connected with the switches by a secure channel using OpenFlow protocol. In the control plane, a controller are able to manage the data plane by installing, deleting or modifying flow-based rules in the switches. According to these rules, the switches decide how to process incoming packets.

In the rest of this section we give an overview of the OpenFlow controller and the OpenFlow switch based on the OpenFlow protocol specification version 1.0[1]. Note that although we adopted this version of OpenFlow in our research, the result is not specific to this version, the same conclusion can be reached with other versions of OpenFlow or other SDN protocols.

2.2.1 OpenFlow Switch

As shown in Fig. 2.2, an OpenFlow switch consists of a flow table and a secure channel. The flow table is in charge of storing rules and the secure channel connects the switch to an external controller. An OpenFlow switch processes packets according to its flow tables like a basic forwarding device. The content of flow table can be managed by the controller through the secure channel.

In the flow table, the rules are implemented as entries. For each entry, there are three components composed (see Table 2.1):

Header fields	Counters	Actions

Table 2.1: Components of a flow entry in a flow table.

- *Header fields*: information like source and destination ethernet addresses, TCP ports, IP addresses, etc, to match against incoming packet headers.
- *Counters*: statistics about flows like number of matched packets, number of bytes, number of errors, etc. Counters updates its data when packet headers matched header fields. There are many counters supported in the documentation, but most of them are optional.
- *Actions*:an instruction to tell the switch how to process an incoming packet with matched headers with the header fields. Actions supports multiple network behaviours like dropping the packet, forwarding the packet to next port, or applying some modifications to the packet. A set of non-conflict actions can be assigned together.

When the controller install a entry to the switch, a timeout value is set to tell the switch when the entry should be removed. There are two types of timeout assigned to each entry, a hard timeout, which is the maximal duration of a rule, and a soft timeout, which is the maximal interval between two packets matching the entry. The rule is removed when timeout is exceeded, no matter which one is. Then entry can be set a infinite timeout so it can be removed by the controller only .

The secure channel is a interface supported by the OpenFlow protocol to transmit message between the switch and the controller, the formation of message is defined in the OpenFlow protocol. In the OpenFlow protocol, there are three classes of message type defined: *controller-to-switch*, *asynchronous* and *symmetric.* The controller-to-switch message is initiated by the controller and sent to the switch, the controller use the controller-to-switch message to detect features, configure, program the switch and retrieve information. Without any solicitation from the controller, OpenFlow switch initiated asynchronous message to inform the controller about errors and events like packet arrivals, state changes. Symmetric messages can be sent by the controller or the switch without solicitation from other side, example for symmetric messages is tje echo message exchanged between the switch and the controller that can be used to indicate the statics of the connection like latency or liveness.

2.2.2 OpenFlow Controller

An OpenFlow controller is a software programmed to populate and manipulate the flow tables in the switches. The controller also keeps translating information about the network statics from the data plane to the services and application in the application layer.

When a data packet arrives at a OpenFlow switch, the packet header will be parsed to match against each entry in the flow table. If the packet header matched the header fields of a entry, the switch first updates the counters of that entry, then processes the packet according to the actions defined in the entry. Otherwise, if the packet header does not match any entry in the flow table, a default action will be applied to this kind of table-miss packet. If the switch is able to buffer the packet, it will encapsulate the packet header in a Packet_IN message and send it to the connected controller, otherwise the Packet_IN message will contain the full packet. The controller receives the Packet_IN message will make the decisions and so can install a specific entry in the requesting switch, or directly command the switch by sending a Packet_OUT message without installing any entry. With this table-miss strategy, a controller can manage all network data flows either in a proactive way and a reactive way.

This section is limited to the necessary background to ease the reading of this paper, many other features provided by OpenFlow are not introduced here, but in following Sections , additional information will be provided when needed.

2.3 Apache Spark

In recent years, due to the exponential growth and availability of huge amount of data with all possible variety, big data becomes a very popular term even outside of automated world [20]. Big data exhibits four main features: volume, velocity, variety, and value [21]. Volume is the one of the most important features, which indicates that the scale of collected data is extremely big. While the data volume is a big challenge, another serious challenge is collecting and processing those data in a timely way. That's what velocity means. Variety indicates the variety of data formats and data types and a variety of applications associated with, e.g., videos, images, multimedia, generated by different application and stored in different file systems. Lastly, the goal is to explore the potential value of the data to achieve better decision making.

To uncover the big data treasure, some kind of analysis tool is needed. A popular tool is parallel computing. Parallel computing is a type of computation that divides large problems into smaller ones, which can then be solved at the same time. Apache Spark is one of the latest parallel computing programming framework.

Spark was developed at the University of California, Berkeley's AMPLab, and later donated to the Apache Software Foundation, which has maintained it since. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. Comparing Spark to Apache's older product Hadoop MapReduce [22], Spark has improved over Hadoop by including the strengths of Hadoop but has made up for its weaknesses. It therefore provides highly efficient batch processing of Hadoop and furthermore the latency involved is also less. Also, iterative processing capabilities for things like machine learning algorithms and graphical analysis are provided by Spark, yet architecturally impossible for MapReduce. Spark has therefore fulfilled the need of parallel execution requirements of analytics professionals and is an irreplaceable tool in big data community [23].

Resilient distributed dataset (RDD) is the most important element to achieve Spark's function. RDD is a distributed data structure physically partitioned inside a cluster across multiple machines, but logically considered as a centralized entity. Within a cluster, inter machine data shuffling can be lowered by controlling how various RDDs are co-partitioned. There is a partition-by operator which by redistributing the data in the original RDD creates a new RDD across machines in the cluster. Fast access is the obvious benefit when RDD is optimally cached in RAM. Currently in the analytics world caching granularity is done at the RDD level. It is like all or none. Either the entire RDD is cached or none of the RDD is cached. If sufficient memory is available in the cluster Spark will try to cache the RDD. This is done based on the least recent use (LRU) eviction algorithm. Expression of application logic as a sequence of transformations is possible through an abstract data structure which RDD provides. This process can happen regardless of the underlying distributed nature of data.

There are two types of operations for expressing application logic in Spark: *transformations*, which perform operations on an input RDD to produce one or more new RDDs, and *actions*, which launch the series of transformations to return the final result RDD.

To compute the atomic predicate set, there are two operations we valued, *map* and *reduce*. These two operations perfectly meet our needs.

Transformation *map*: A *map* is a transformation operation. Map operation applies a function to each element of the original RDD and then returns the result as a new RDD. In the Map, programmer can define his own application logic. The same logic will be applied to all the elements of RDD. Transformation map takes one elements as input, then process it according to a custom function and returns one result element at a time. Thus, the original and transformed RDDs will typically have the same number of elements.

Action *reduce*: A *reduce* is a action operation. Reduce aggregates the elements of a RDD according to a custom function. It is a wide operation as it is shuffling data from multiple partitions and reduces to a single value. The custom function should have two arguments of the same data type and the return type of the function also must be same as argument types.

Chapter 3

Related Works

SDN provide us with many valuable advantages over the traditional network architecture, especially the ability to easily program the network. But like other new technologies, advantages that may also lead to new problems, including new configuration issues and security vulnerabilities. Different users share one controller, or multiple controllers operate in the same domain without effective coordination may create conflicting rules which leading to black hole, loop or other configuration errors in SDN. Malicious attackers or applications also may use this vulnerability to bypass security policies by installing strategic entries to redirect traffic. This chapter describes potential configuration problems as well as security problems, and discusses the work that has been done to find and resolve the problems.

3.0.1 Detecting Network Errors

For a large networks consisting of hundreds of switches, with multiple applications requiring different traffics, the SDN controllers needs to install approximately 50,000 new flows per second to meet those requirements[24], hence, a quick, efficient mechanism to automatically detect and correct the errors as well as ensure security policies and fault tolerance is very urgent. Recent work on network property verification for OpenFlow network focuses on the following three aspects [25].

(1) Finite state machine based verification. Packet is treated as a finite state machine. Verifying network property is equivalent to determine if there is an undesirable state in the set of final states. FlowChecker [26] is the earliest work in this area, it uses binary decision diagrams to detect intra-switch configuration errors within a single flow table. To identify errors generated by SDN applications. Flover[27] uses modulo theories and

assertion sets to achieve a automatic flow policies verification method. When the controller try to add a flow rule, Flover verifies the new rule against the non-bypass security properties enforced within the network. Header Space Analysis (HSA) [9] models packet as a point in a geometric space and networking box as a switch transfer function. Instead of using an available verifier, it designs multiple algorithms to verify different network properties.

(2) Boolean satisfiability based verification. Network property verification is transformed into boolean satisfiability problem (SAT) on logic network. Anteater[28] analyzes network configuration problems by using a static analysis approach for checking invariants. It converts rules to Boolean expressions, translates network properties to instances of SAT problems and analyzes network configuration problems by using a static analysis approach for checking invariants.

(3) Graph based verification. The algorithm for computing reachability of the graph is the core algorithm to verify network properties. VeriFlow [7] divides rules into equivalence classes (EC) and models the EC network as a graph to detect misconfigurations such as loops, unavailable paths. With the ability to intercept flow rules before they are applied to the switches, VeriFlow is able to verify invariants in real-time. But the number of ECs can be quite large when rules have multiple matching fields. It will slow down verification speed. Another Graph based verification solution is NetPlumber[8]. NetPlumber maintains a dependency graph between flow rules to incrementally check for compliance of state changes. It is developed based on Header Space Analysis, the author's previous work. NetPlumber improves upon HSA by working incrementally with more flexibility in checking complex policy queries so that it is fast enough to verify every flow rule updating request in real time. But it takes long time when link up/down event occurs.

The analysis shows that the first two techniques are generally slow compared with graph-based ones. Furthermore, graph search techniques make graph based verification able to verify network-wide invariants, and handles dynamic changes in real time. To reduce the verification time, Libra [5] is the first to introduce MapReduce to realize parallel verification, but it assumes that packets are forwarded based on destination IP prefix and that headers are not modified. It is not suitable for OpenFlow network. Recently, a new method has been introduced by [10], [29], called atomic predicates. Atomic predicates have the following property: Each given predicate is equal to the disjunction of a subset of atomic predicates and can be stored and represented as a set of integers that identify the atomic predicates. The conjunction (disjunction) of two predicates can be computed as the intersection (union) of two sets of integers. Thus, intersection and union of packet sets can be computed very quickly. Based upon this idea, they developed a formal analysis method that enables very fast computation of reachability. But they didn't mention about firewall bypass threat.

3.0.2 Attacks from SDN application

As described above, the SDN applications are able to program the network by asking the SDN controller to do so. However, malicious SDN applications could produce some unwanted traffics in the network due to the lack of trust between the controller and applications, for example, traffics violated with the firewall policy, which makes it a serious security concern.

In order to prevent such problem, FortNOX[30] introduced a security-enforcement kernel based on the open-source NOX controller[31]. FortNOX implemented a digital signature mechanism to constrain different users or applications, and offers a method to verify conflicting flow rules before they reach the network. In this method, FortNOX first converts all rules to *alias reduced rules* by adding source addresses, destination addresses and every rewritten address caused by the rule's action instruction into an alias set. Then, these new rules can be easily checked against the firewall policy. However, this rule conflict analysis algorithm is unable to accurately track network traffic flows and only consider bypass threat in single flow policy.

In [32], the authors developed an algorithm to detect and resolve the conflict between firewall policies and flow rules for the SDN firewall application. For this conflict detection, the algorithm defined a firewall authorization space according to all firewall policies, and uses HSA for modeling the network to create and maintains a shifted flow graph. Once the conflict was detected, the algorithm blocks the violated flow path by inserting deny rules for conflict caused by firewall policies update, or refusing the request for conflict caused by new flow rule update request. Most recently, FlowGuard [33] proposed a solution based on HSA to detect firewall policy violations in SDN with a more refined conflict resolution. FlowGuard also use HSA to maintain its flow path spaces, but unlike [32] that simply blocking the flow or rejecting the rule update, FlowGuard offers five different conflict resolutions: flow rejecting, dependency breaking, update rejecting, flow removing and packet blocking. FlowGuard can select appropriate strategy for different extent of the violation. However, since these two solutions both are working in a reactive way, they have to assume there are no violation before they were deployed to the system, they can not actively detect this potentially violation.

Chapter 4

Problem Statement

Since SDN allows network applications to operate with switches in the networks directly, it faces a variety of security challenges.

4.1 Loop-freedom and Nonexistence of Black Holes

In our research, we focus on specific two essential network properties, loop-freedom and nonexistence of black holes. In SDN network, a loop could happen when mistakenly installed forwarding rules. If a packet is sent into a looped path, it can loop until its time to live (TTL) value expired. On the other hand, a black hole could happen when mistakenly deleted a forwarding rule. If a packet is forwarded to a switch with no match forwarding rules, it will be dropped without informing the source.

4.2 Firewall and Bypass Threat in OpenFlow Networks

As a new network architecture, new configuration issues lead to new security problems. One of these security problems is the firewall policy can be bypassed in SDN. In this section, we discuss the problem of deploying a traditional firewall application in SDN, then describe and analyze the firewall bypass threat.

4.2.1 Firewall in OpenFlow Networks

Firewall technology emerged in the late 1980s when the Internet was a fairly new technology in terms of its global use and connectivity [34]. A firewall is a network security application that monitors and controls the incoming and outgoing network traffic based on predetermined security rules [35]. Firewalls located on the gateway machine of a network are called network-based firewalls, and those positioned on network nodes are called host-based firewalls. The host-based firewall usually is a operating system service or a software application configured by the host user, and only benefits the host itself. On the contrary, the network-based firewall can secure the entire network by filtering traveling data according to a set of data management policies.

In a OpenFlow network, the OpenFlow controller manages the data traffics in the entire network by the routing policies it applied to switches. If an incoming packet is not defined, the switch will send it to the controller and ask the controller to determine how to deal with the packet. This behavior sounds like the OpenFlow controller is behaving like a traditional network-based firewall. To use an OpenFlow controller as a firewall, most modern OpenFlow controllers implement a built-in firewall application which behaves like a traditional firewall, e.g. Floodlight, a popular open-source OpenFlow controller developed in Java, has a Firewall module application. By inspecting Packet_IN massages sent from switches, the application can enforce access control list (ACL) rules. This kind of firewall works in a reactive way.

4.2.2 Packet Modification in OpenFlow

OpenFlow defines a *set_field* action type to provide more flexibly in programming the network. Set_field action is able to modify various values of the packet. The types of modify action supported by OpenFlow are shown in Table 4.1, the usefulness of an OpenFlow implementation is greatly increased by adopting these actions. Since The set_field actions are identified by the field types they modify, one entry can contain multiple set_field actions to set multiple field types, but for each field type, at most one set_field action can be set. This means the modification can be accomplished both in a single switch or in respectively multiple switches.

Action	**Associated Data**	**Description**
Set VLAN ID	12 bits	If no VLAN is present, a new header is added with the specied VLAN ID and priority of zero. If a VLAN header already exists, the VLAN ID is replaced with the specied value.

Set VLAN priority	3 bits	If no VLAN is present, a new header is added with the specied priority and a VLAN ID of zero. If a VLAN header already exists, the priority eld is replaced with the specified value.
Strip VLAN header	-	Strip VLAN header if present.
Modify Ethernet source MAC address	48 bits: Value with which to replace existing source MAC address	Replace the existing Ethernet source MAC address with the new value.
Modify Ethernet destination MAC address	48 bits: Value with which to replace existing destination MAC address	Replace the existing Ethernet destination MAC address with the new value.
Modify IPv4 source address	32 bits: Value with which to replace existing IPv4 source address	Replace the existing IP source address with new value and update the IP checksum (and TCP/UDP checksum if applicable). This action is only applicable to IPv4 packets.
Modify IPv4 destination address	32 bits: Value with which to replace existing IPv4 destination address	Replace the existing IP destination address with new value and update the IP checksum (and TCP/UDP checksum if applicable). This action is only applied to IPv4 packets.
Modify IPv4 ToS bits	6 bits: Value with which to replace existing IPv4 ToS eld	Replace the existing IP ToS eld. This action is only applied to IPv4 packets.

Modify transport source port	16 bits: Value with which to replace existing TCP or UDP source port	Replace the existing TCP/UDP source port with new value and update the TCP/UDP checksum. This action is only applicable to TCP and UDP packets.
Modify transport destination port	16 bits: Value with which to replace existing TCP or UDP destination port	Replace the existing TCP/UDP destination port with new value and update the TCP/UDP checksum This action is only applied to TCP and UDP packets.

TABLE 4.1: Set-Field actions [1].

4.2.3 The Firewall Bypass Threat in OpenFlow Networks

A simple bypass example is shown in Fig. 4.1, an SDN network uses OpenFlow and comprises one controller, one switch and four hosts. A firewall application locates in the controller blocks part of traffic between host A and host C by containing a firewall policy that every packet sent from host A to host C should be dropped. This firewall policy can be easily bypassed by installing a flow table entry with modify actions. Every packet sent from host A to host B will match the entry, then the switch will modify the source IPv4 address and the destination IPv4 address of the packet, the original packet will be manipulated as a packet sent form host D to host C and forwarded to host C. Note that although the result of this entry bypassed the firewall, each action does not directly against the firewall policy.

Meanwhile, the result of each entry may affect the packet matching in the next hop, this dependency of rules makes the detection more difficult. Now, considering a complex scenario in Fig. 4.2, which comprising three switches.Instead of installing one entry, we install three entries respectively into the flow table of three switches. The first entry modifies the source IP address of the packet sent from host A to host B to host D, then forward the modified packet. So the modified packet will match the entry in the second switch, this entry will modify the destination IP address of the packet to host C, then forward to the next hop. The last entry just forwards the packet with host D as the source address and host C as the destination address. Note that even the result of each entry does not directly against the firewall policy. To detect the firewall policy violation using the dependency of rules, the flow path needs to be accurately tracked.

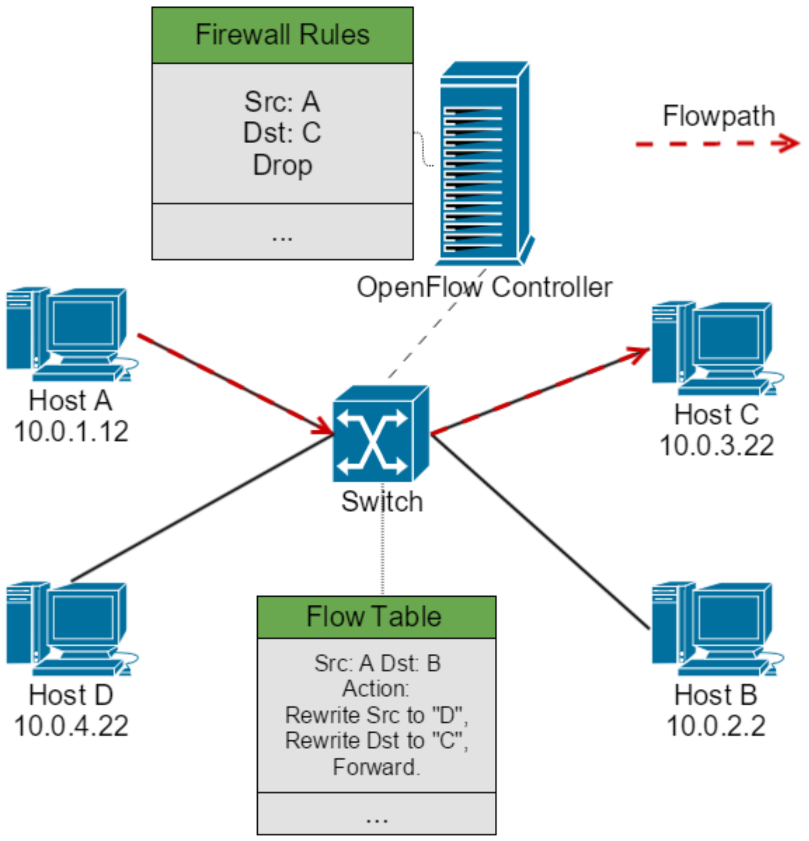

FIGURE 4.1: Inserting one flow entry into the switch's flow table to bypass an SDN firewall.

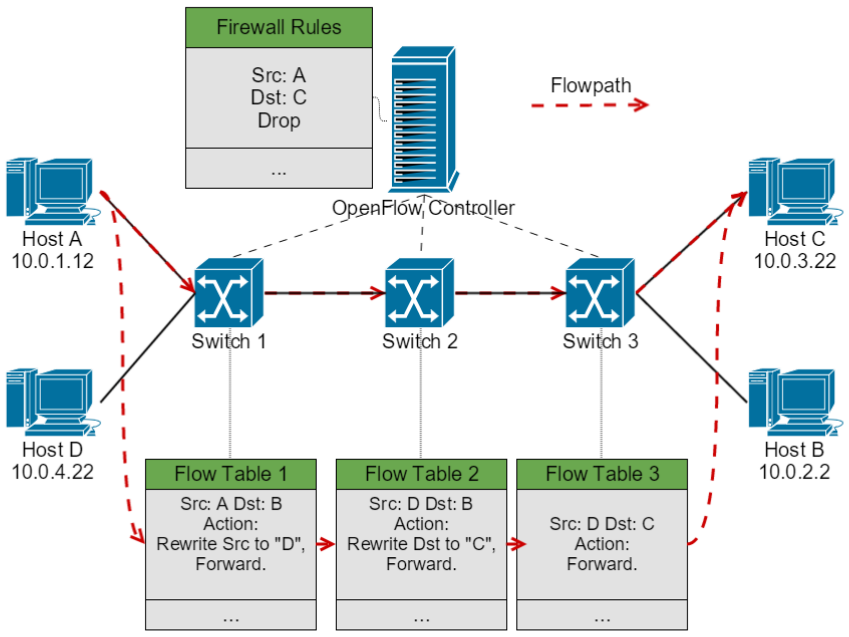

FIGURE 4.2: Inserting three flow entries into the three switches's flow tables to bypass an SDN firewall.

In this kind of firewall bypass scenario, the real source address of the packet is the original source address, and the real destination address is the last modified destination address, and only the communication between the real source address and the real destination address is forbidden by the firewall rules. Obviously, before modify the destination address to the real destination address, the original source need to be modify first, otherwise it will cause a directly conflict with the firewall rules.

Once the firewall policies are bypassed, serious problems could happen to the network. Data leaking is the most direct consequence. For example, an SDN network defines two traffics, one is defined for researchers doing their experiments, the other one is defined for normal users regular use. A firewall is deployed to secure the network traffic from overlapping. Attackers are able to bypass the firewall and steal data from experiments traffic.

4.2.4 System Model and Assumptions

Before proceeding to the proposed detection scheme, it is worthwhile to make some assumptions on our problem:

- The network is assumed to adopt software-defined networking and use OpenFlow Switch Specification Version 1.0.0 to achieve it.
- Since we are focusing on firewall bypass threat, only a few of flow entry actions are considered, and the header field of wildcard are not be cared. To simulate this threat, as described before, we use the set_fields actions which modify the source and destination IPv4 address.

Note that although we adopt this version of OpenFlow in our research, the result is not specific to this version, the same conclusion can be reached with other versions of OpenFlow or other SDN frameworks.

Chapter 5

A Novel Method against the Firewall Bypass Threat in OpenFlow Networks

In this research, our goal is *to design a firewall for SDN networks which can detect and resolve the threat of using flow rules to bypass the firewall policies.* To achieve the goal, we analyze the specification of packet modification behavior in OpenFlow networks and its impact to the traditional firewall deployed in OpenFlow networks. After we identify the problem, we make our own contribution to formalize and model the OpenFlow networks and the firewall bypass threat. We propose a method to accurately detect and prevent firewall bypass threat in OpenFlow networks. Finally, we implement our method and conduct experiments. The result of experiments shows that our method can accurately detect bypass threats in OpenFlow networks. Comparing to previous research work, our method has following two significant features, flow path tracking and active detection.

5.1 Violation Detection

5.1.1 Flow path in OpenFlow networks

In OpenFlow networks, we realize that the forwarding path of a data flow is decided by entries installed in the flow tables of the OpenFlow switches. Although a packet can match multiple flow table entries, the packet must match against each flow entry by prioritization, means entries with higher priority will match before entries with lower priority. Hence only the action of the flow entry with the highest priority will be applied. Practically, two flow entries with the same match header fields cannot have the same

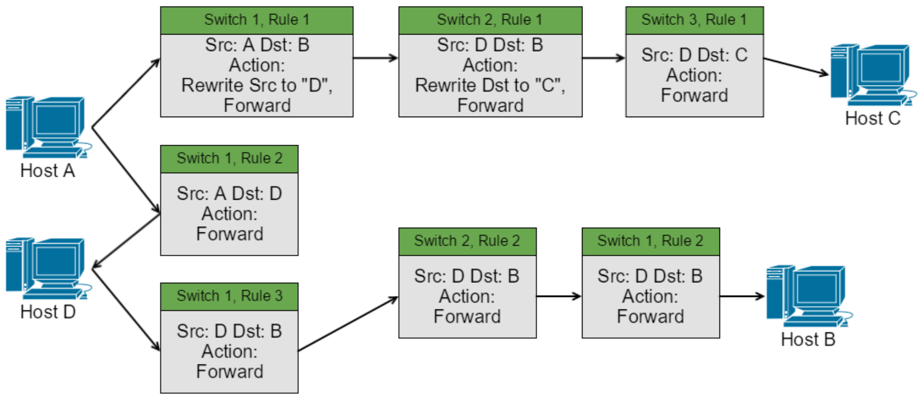

FIGURE 5.1: All flow paths form a directed graph.

highest priority thanks to the automatic overlapping checking mechanism: for a flow entry insert request, the switch must first check is there any flow entries with the same header fields and the same priority existed in its flow table. If a flow entry with the same value has already existed in the flow table, the switch will refuse the insertion request of the overlapped entry. Since flow entries are stored in order, we can locate a flow entry by its switch's id and its order in the flow table.

We use a *rule* to refer a flow table entry. A rule is an OpenFlow-like $< match, action >$ tuple, match represents the header fields of the entry for matching packets and the action represents the actions that can be forward, drop, modify. Thus, we can describe a flow path start form a host node a to a host node b by following (switch, rule) sequence:

$$a \to (s_1, r_1) \to ... \to (s_{n-1}, r_{n-1}) \to (s_n, r_n) \to b. \tag{5.1}$$

As shown in Fig. 5.1 and Fig. 5.2, All flow paths form a directed graph, we call it *Forwarding Graph.*

5.1.2 Forwarding Graph

For a formalization of the pervious, we define a Forwarding Graph $FG = (V, A)$:

- There are two kinds of nodes: $V = \{V_H, V_E\}$.
  - V_H is a set of nodes which represent host devices in the network. These nodes are called *host nodes.* Host nodes are identified by the addresses of represented host devices.

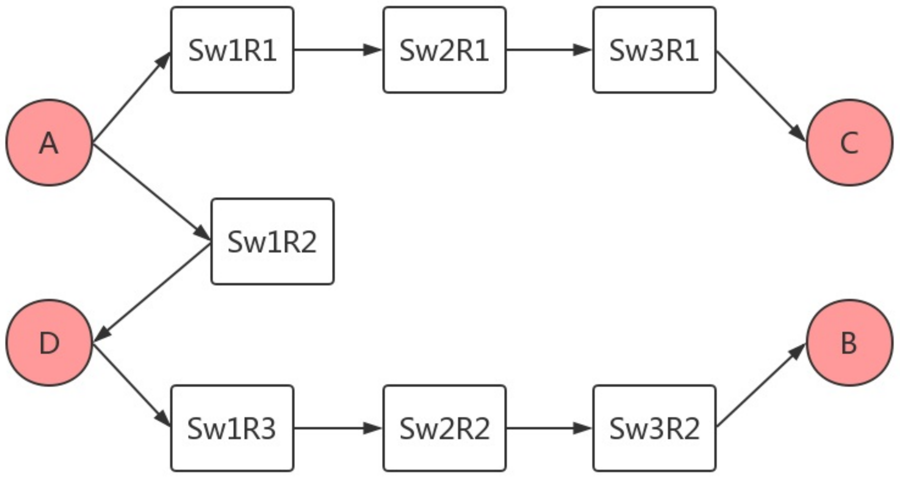

FIGURE 5.2: Forwarding Graph of the network in Fig. 5.1.

  - V_R is a set of nodes which represent rules in the network. These nodes are called *rule nodes*. A rule node is identified by the ID of the switch which the represented rule located and its order in the flow table. The rule node records the header fields of a rule as an *ingress match* tuple like *(ingress source, ingress destination)*, including the original source address and the original destination address. The rule node also records an *egress match* tuple like *(egress source, egress destination)* tuple, including the source address and the destination address after the actions of the rule are applied to the packet. The egress match tuple indicates the action of the rule, for forward action, the egress match remains the same to the ingress match, for drop action, the egress match is set to null.

- A is a set of ordered pairs of vertices, denoted as $A = \{(x, y) \mid x, y \in V\}$, each pair means node x is linked to node y. Node x links to node y when satisfy these constraints:

  - If x is a host node and y is a rule node, x is included in ingress source of y.
  - If x is a rule node and y is a host node, y is included in egress source of x.
  - If x and y are both rule nodes, x's ingress match tuple matches y's egress match tuple.
  - There is a physical link between node x's box and node y's box.

In the Forwarding Graph, if there exists a path:

$$[a, v_1, v_2, v_3, ..., b | a, b \in V_H, v_1, v_2, v_3, ... \in V_R] \tag{5.2}$$

we say b is reachable from a. To compute this reachability in the Forwarding Graph, we modified the Dijkstra's algorithm to only seek rule node when find a shortest path between two host node. This algorithm can guarantee that the result of the reachability computation is correct.

5.1.3 Firewall Deny Space and Firewall Policy Violation

For firewall rules, we introduce a *Fire Deny Space*: S_D, to represent a collection of all connections denied by the firewall rules. In a Firewall Deny Space: $S_D = \{s_1, s_2, ..., s_n\}$ we use a *Firewall Deny Set*: $s = \{V_s, V_d \mid V_s, V_d \in V_H\}$ to represent a single firewall rule which indicates every packet sent form $a \in V_s$ to $b \in V_d$ is denied. V_s is a host nodes set indicating the source of a packet and V_d is a host nodes set indicating the destination of a packet. In a firewall the DENY rules are complementary to ALLOW rules, hence we only care the DENY rules.

Now we can give a formal description of our problem:

- For each Firewall Deny Set s, if $b \in V_d$ is reachable from $a \in V_s$, we can say that there is a firewall policy violation existing in the network.

Obviously, the firewall policy violation is caused by the rules among the violated path.

5.2 Violation Resolution

5.2.1 Initialization

To identify the violation, we first obtain the topology of the network, flow tables of the switched and firewall polices to build our *Forwarding Graph* and *Firewall Deny Space* for the network. Once we finished the initialization, an active detection will be executed to check potentially violation until there are no violation exists. If violations exist, the violated rules will be removed. This is designed for better deployability. Our method does not require the network to be secure before the implementation. Thus, users can deploy the detection at any period of the network.

5.2.2 Update the Forwarding Graph

Adding rules: When a new rule is added, As shown in Algorithm 1, the Forwarding Graph first creates a rule node for it, then finds all links to the new node and form the new node. After finishing the update of the Forwarding Graph, the firewall policy violation will be checked. If there is a violation, the insertion will be reject and the update will be reversed.

Algorithm 1 Flow path track algorithm

```
Input: A Forwarding Graph, $FG$; A Firewall Deny Space, $S_D$; A new rule node, $sw_x r_y$;
  The topology of the network, $Topo$.
Output: A boolean variable, $New\_Node\_Added$
  $FG.AddNode(sw_x r_y,\ Topo)$;
  $New\_Node\_Added \leftarrow TRUE$;
  while $s_d \in S_D$ do
    for $a \in s_d.V_s$ do
      for $b \in s_d.V_d$ do
        if $(FG.has\_path(a, b) = TURE)$ then
          $New\_Node\_Added \leftarrow FALSE$;
          $FG.DeleteNode(sw_x r_y)$;
          $Break$;
        end if
      end for
      if $New\_Node\_Added = FALSE$ then
        $Break$;
      end if
    end for
  end while
  return $New\_Node\_Added$;
```

Deleting rules: Deleting a rule also may cause a firewall policy violation, hence, after deleting the rule node and remove its all links, we check the reachability against the Firewall Deny Space, if a violation is detected, the entire flow path will be removed.

5.2.3 Update the Firewall Deny Space

Adding Rules: When a new firewall rule is added, new violation may arises. Before updating the Firewall Deny Space, the Forwarding Graph will be checked against the new Firewall Deny Set, if violation detected, the entire flow path will be removed.

Deleting Rules: Obviously, deleting a firewall rule will not introduce any new firewall policy violation, so we just delete the corresponding Firewall Deny Set from the Firewall Deny Space.

5.3 Implementation

In this section, we present the detail of our implementation including the environment, and the implementation framework. Then we compare our method with other approaches of resolving firewall bypass threat in SDN and conduct evaluations to test the performance of proposed method.

5.3.1 Implementation Environment

We implement our method and run experiment base on Ubuntu operating system version 14.04 and use Mininet [36] to implement the OpenFlow network virtually, the network is controlled by Floodlight controller [13] , the system is hosted on an Intel Core 2 3.00GHz E8400 CPU with 2.9GB RAM. Specifications of the environment we used are listed in Table 5.1.

Software	**Version**
Floodlight	0.91
Mininet	2.21
OpenFlow	1.0.0

Table 5.1: Specifications of the environment we used.

5.3.1.1 Mininet

Mininet is a fast, lightweight but powerful network emulator. Mininet is capable to create a realistic virtual network running functional hosts and switches. Those hosts are able to run customized programs. One of Mininet's most powerful and useful features is Mininet's switches are programmable using the OpenFlow protocol. By default, Mininet runs Open vSwitch [37] [38], an open-source multilayer virtual switch supporting multiple protocols. Open vSwitch requires an OpenFlow controller since it runs in OpenFlow-only mode, which means all data flow can only be processed by defined flow entries or the controller. Mininet comes with three controller classes to support three types of controllers: the built-in OpenFlow reference controller, NOX, and Open vSwitchs ovs-controller. These controllers are out-of-date or have limited function. Fortunately, Mininet also provides the capability of interacting with user's custom OpenFlow controller.

5.3.1.2 Floodlight

We choose Floodlight as our controller because it has a very active community and has a number of sample module applications, such as a leaning switch and a firewall. Floodlight comes with a web based and Java based GUI and most of its functionality is able to access through a REST API.

There are three module applications we valued:

Forwarding: Forwarding module is designed to automatically find the shortest path for table-miss packets, adds the appropriate flow entries into related switches, and lets the switch continue its forwarding. Forwarding module lets Floodlight able to work in networks that contain both OpenFlow and non-OpenFlow switches. For us, Forwarding makes the OpenFlow-only Open vSwitch act like a normal non-OpenFlow layer 2 switch. By default, these flow entries added by Forwarding are assigned with very short timeout and low priority, in order to do not influence user added flow entries and keep the network not redundant.

Static Flow Pusher: Forwarding inserts flow entries in a reactive way, Static Flow Pusher inserts in a proactive way. User can use Static Flow Pusher's REST API to add custom flow entries into specific switches. For example, we can use the following curl command to insert a flow on a switch which ID is 1 that forwards packets sent from a host with IP 10.0.0.1 to a host with IP 10.0.0.2. Serval set_field actions are also provided by Static Flow Pusher.

```
curl -d '{"switch": "00:00:00:00:00:00:00:01", "name":"flow-mod-
1", "src-ip":"10.0.0.1", "dst-ip":"10.0.0.2",
 "active":"true", "actions":"output=normal"}' http://localhost:8
 080/wm/staticflowentrypusher/json
```

Firewall: Firewall enforces ACL rules by monitoring Packet_IN messages. ACL rules here are set of conditions that allow or deny a traffic flow at its ingress switch. Firewall

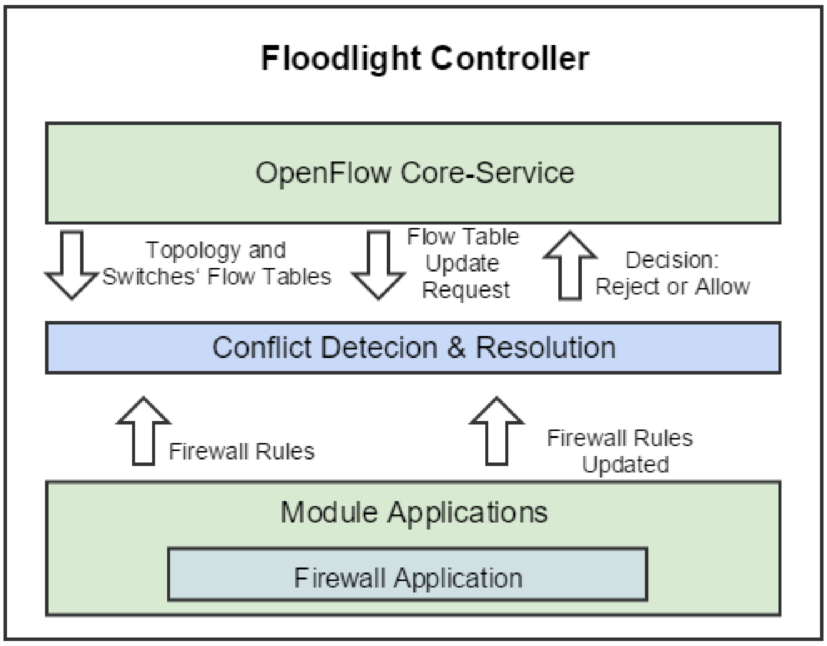

FIGURE 5.3: Implementation frame of our method.

also exposes REST interface for users, for example, the following curl commands order switch 1 to drop all packets sent from the host with IP 10.0.1.12 to the host with IP 10.0.3.22, which is the firewall policy described in pervious bypass scenarios.

```
curl -X POST -d '"src-ip":  "10.0.1.12/32", "dst-ip":"10.0.3.22/32"'
http://localhost:8080/wm/firewall/rules/json
```

5.3.2 Implementation

As shown in Fig. 5.3, our detection module is implemented in the Floodlight controller as a class. The flowchart is shown in Fig. 5.4. The detection module will first obtain the topology message from the controller, reads firewall rules from Firewall Module and flow entries from Static Flow Pusher Module to initialise the Forwarding Graph, FG and the Firewall Deny Space, S_D. Once FG and S_D are built, the violation will be checked to finish initialization. After initialization, when the controller try to update flow entries in the switches, or add new firewall rules, the module will be triggered and run the violation check, decision will be give according to the result of detection.

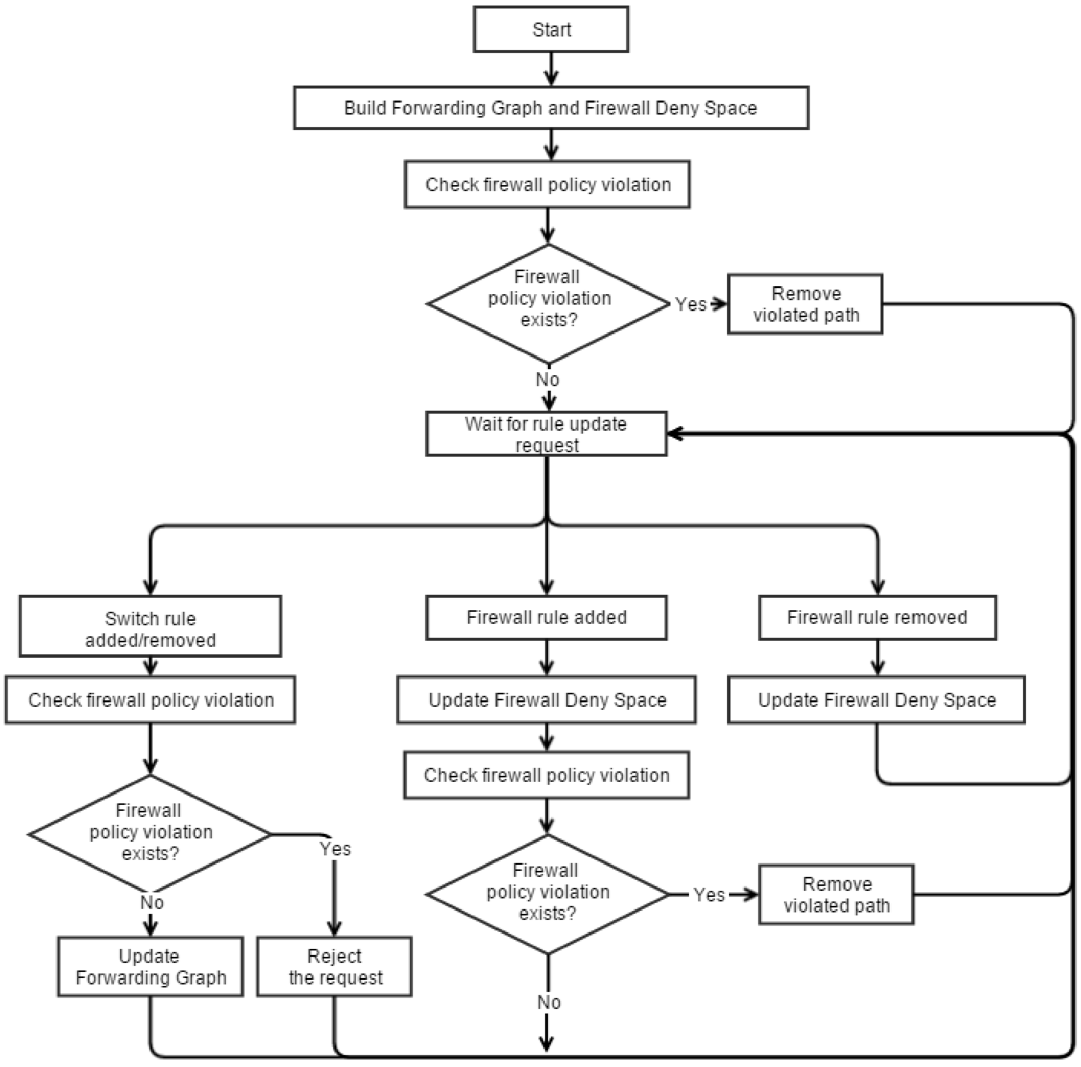
Start
Build Forwarding Graph and Firewall Deny Space
Check firewall policy violation
Firewall policy violation exists?
Yes
Remove violated path
No
Wait for rule update request
Switch rule added/removed
Firewall rule added
Firewall rule removed
Check firewall policy violation
Update Firewall Deny Space
Update Firewall Deny Space
Check firewall policy violation
Firewall policy violation exists?
Yes
No
Firewall policy violation exists?
Yes
Remove violated path
Update Forwarding Graph
Reject the request
No

FIGURE 5.4: Flowchart of our method.

5.4 Performance Evaluation

To test our method's functionality, we simulate two scenarios described in Chapter 4. We first build the same topologies and set the firewall rules to block IPv4 communication between Host A and Host C, then we start a simple HTTP web server application in Host C to reply packets sent from Host A. After the setup, we try to insert those bypass-enable flow entries. The result showes our method is able to reject the insertion request while the original Floodlight firewall allowed. Furthermore, we conduct the simulations with those bypass-enable flow entries installed in the switches before the initialization of our method. The threat is detected and the entries are removed after the initialization,

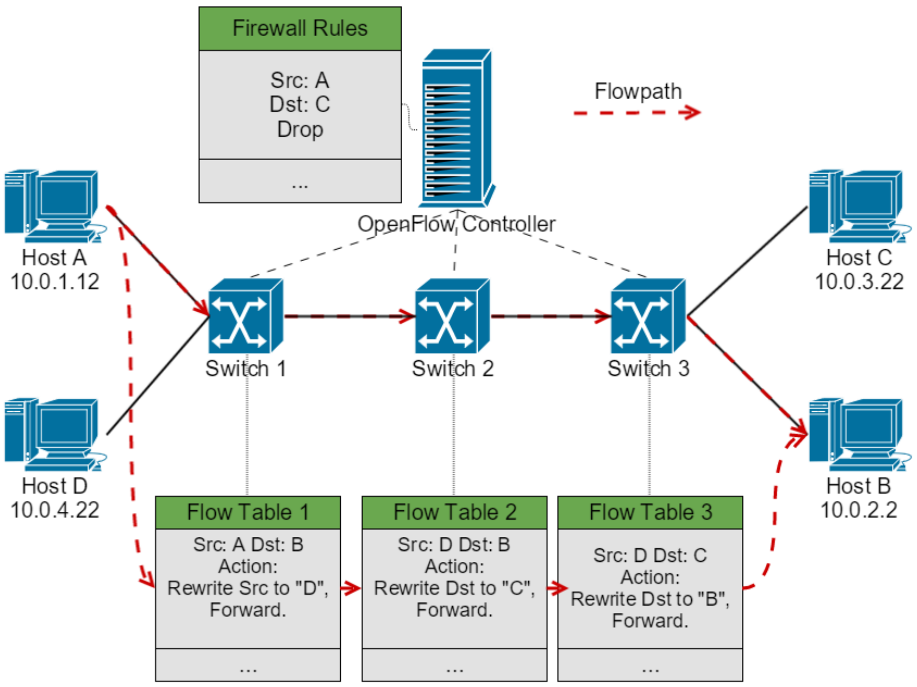

FIGURE 5.5: Example scenario for performance comparison.

thanks to the active detection mechanism. As shown in Table 5.2, our method has two significant features comparing to previous works.

	Flow Path Tracking	**Active Detection**
FortNOX	×	×
J Wang et al.	○	×
FlowGuard	○	×
Our method	○	○

TABLE 5.2: Features comparison.

Considering a scenario shown in Fig. 5.5, suppose the firewall application blocks the communication between host A and host C. There are three flow entries respectively installed in the three switches, these entries are similar to pervious bypass-enable entries, but in switch 3, a new entry will modify the destination IP address to the IP address of host B for all packets sent form host D to host C. By alias rule reduction algorithm introduced in FortNOX, since it simply transforming the rules' match criterion to detect bypass violations without tracking the flow path, a conflict will be detected. However the final destination of packets sent from host A to host B is host B, the packets will not arrive at host C, means the alias rule reduction algorithm detected a fake conflict. Our method is tested in this scenario and still keeps it preciseness.

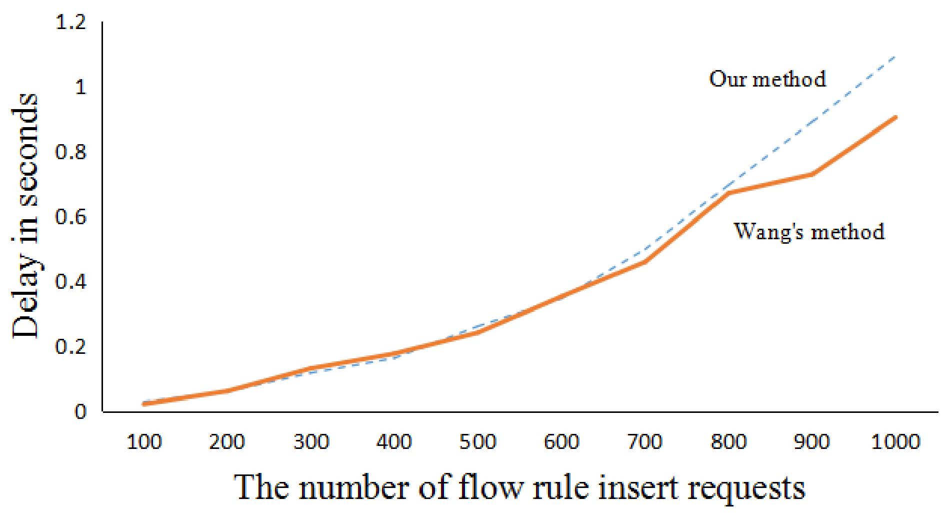

FIGURE 5.6: Delay of evaluating a variety number of flow entries against firewall policies.

Since our method requires the controller to evaluate policy update requests before executing them, the delay is unavoidable while achieving the security. We study the performance in term of extra computational time while increasing the number of inserted flow entries. The topology of the network remains the same to pervious scenario, which consists of three switches and four hosts. We use the `pingall` command provided by Mininet to generate the most of entries. This command goes through the host list and attempts to ping every other host in the topology. `Pingall` is chosen since it triggers Floodlight's Forwarding module to automatically install flow entries. `Pingall` requires that a new rule be written to the switch for both the request and reply of the majority of pings. The only time a new rule is not written is if the rule already exists from a previous ping, i.e. when host A is pinging host B while the flow rules from host B pinging host A is still resident in the switch's flow table. We changed the default timeout of entries produced by Forwarding module so we can easily collect them. We also designed a great deal of entries try to bypass the firewall policy. The overlapping check mechanism is disabled so we can insert more entries than the network needed. This places a really large load on our implement to validate each rule as well as keep an up to date view of the switches' flow tables.

For comparison, we implement the detection algorithm introduced by Juan Wang et al. which is referred to as Wang's algorithm in this thesis. Fig. 5.6 shows the performance of our implement and their algorithm with a varying number of rule insertion requests. Fig. 5.7 illustrates the computational delay required to detect violation between candidate rules and the firewall policy increasing during the detection procedure. As more and more rule insert requests are evaluated, the delay is increasing but still remaining in a acceptable range when comparing to others. These results show the performance

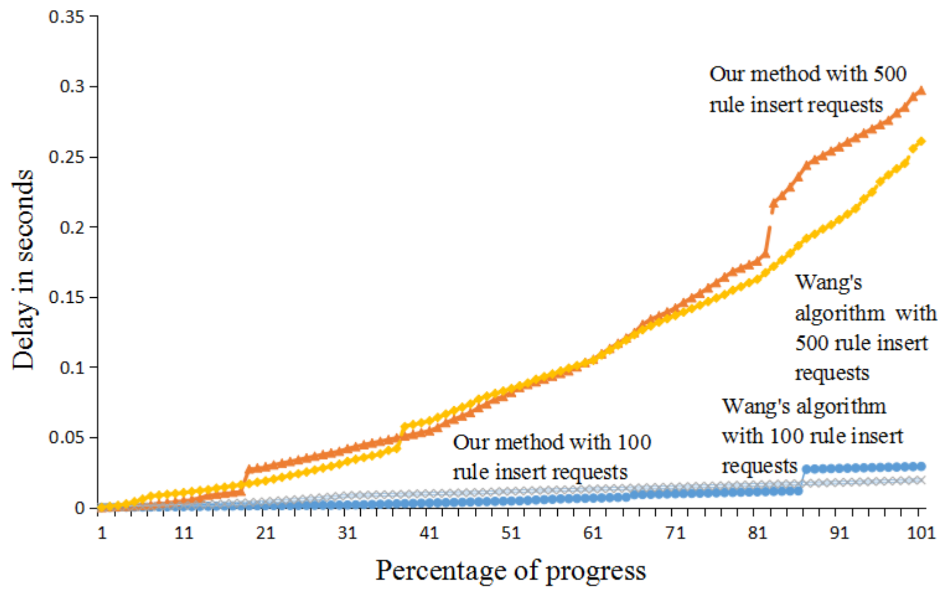

FIGURE 5.7: As more and more rule insert requests are evaluated, the delay are increasing but still remaining in a acceptable range.

along with the increasing numbers of requests, the computing costs by our method are increasing. Note, in our experiments we force our method to evaluate each incoming flow entry against each firewall policy without any whitelisting rules, in practice, this kind scenario could represent a worst case.

5.5 Conclusion

In this research, we address the challenge of building a robust OpenFlow firewall. We introduce a new threat to the firewall in SDN, firewall bypass threat. This threat is caused by violation between firewall rules and flow policies. To detect such violation, we propose a novel detection method based on modeling the network to a directed graph, the problem is transformed to the reachability computation problem. Then, we implement our method in Floodlight and conduct experiments by Mininet. The experiments are designed to evaluate the functionality and accuracy of the proposed detection method. The results of experiments show that our method can accurately and actively detect and resolve bypass threats in OpenFlow networks with acceptable overhand.

Chapter 6

Atomic Predicates Based Data Plane Properties Verification in Software Defined Networking Using Spark

In this research, we propose a new method of fast verification by introducing atomic predicates to deal with the time-consuming problem of existing techniques. Our method starts with the full graph of switches, each with its own forwarding table. Our method analyzes reachability for each ports by modeling the network to a directed graph, the problem is transformed to the reachability computation problem. By adopting atomic predicates, which replaces highly computation intensive operations on predicates by those on sets of integers to speed up verification and reduce the cost of storage. In addition, we propose a parallelized method to compute atomic predicates with Apache Spark. Finally, we implement our method and conduct experiments. The result of experiments shows that our method can compute reachability much faster and space efficiency in OpenFlow networks comparing to our previous research work. Compared to previous research work, our method has two significant features shown in below.

- *Firewall bypass threat detection*: Besides the two essential data plane properties, loop-freedom and nonexistence of black holes, our method is able to detect the firewall bypass threat which is rarely explored in existing data plane property verification methods.
- *Parallelized atomic predicates computation*: We realize that the computation of atomic predicates can be parallelized. We propose a method to preprocess rules

and compute atomic predicates with Apache Spark, which makes the computation of atomic predicates faster and more scalable.

6.1 Proposed Network Model

The goal is to check packet reachability in the network: can a packet with specific header value sent from host a reach host b. To do that. To achieve that, we need a network model to describe packet activation in the packet-switched network. We proposed a network model by representing packets with predicates, and describing network connections with a directed graph.

6.1.1 Network Packet

A network packet is a formatted unit of data routed between a source and a destination on a packet-switched network. Each packet consists of control information and user data. Control information is used by network device to deliver the packet, for example: destination network addresses, error detection codes, and sequencing information. Typically, control information is found in packet headers and trailers. Each packet has a header of h bits. The header bits are partitioned into multiple fields. For an IPv4 packet header, it contains fields such as TTL, protocol type, IP address. Notice that not all fields are needed by a switch to determine the packet where to go. Thus, we only focus on those reachability relevant fields, in particular, source IP address and destination IP address in this paper. The value of each field is represented by a sequence of bit variables, for example, in an IPv4 packet header, the destination IP address fields value is a sequence of 32 bits 1 and 0. The set of all possible sequences represents the set of all packets, namely, the *packet space*.

As we mentioned above, when a packet arrives at a OpenFlow switch, the packet header will be check against each entry in the flow table. If the packet header matched the header fields of a entry, the switch will process the packet according to the action defined in the entry. In this paper, we focus on 3 actions, which are forwarding, rewriting and dropping. Obviously, packets with identical header values will be processed identically, those packets set we call it equivalence class. Formally, an equivalence class is defined as follows.

Definition (Equivalence Class): Given a packet set P, if any $p_1, p_2 \in P$ are treated identically in the network, we call the set P a *equivalence class*. $\diamondsuit$

We use the value of packet header field to represent a set of packets, particular, we use IP address fields values to represent the set of all packets which sharing the corresponding values. For example, by destination IP address value 10.1.1.1/32, we represent all packets sent to 10.1.1.1/32. Or, by source IP address value 10.0.0.2/32, we represent all packets sent from 10.0.0.2/32. We abstract away the data portion of a packet because we assume it does not affect packet forwarding. To represent sequences, we use predicates.

A predicate is a Boolean function which is evaluated to be true or false depending on the values of its variables. We use predicates as indicator functions. For example, given a set X and a subset A of X, we use following predicate to represent set A:

$$p(x|x \in X) = \begin{cases} true, \ if \ x \in A; \\ false, \ if \ x \notin A. \end{cases}$$

The predicate is denoted p in the following contents. Note that if $A = X$, then $p = true$; if $A = \emptyset$, then $p = false$.

Given a 4-bit sequences set $X = \{v_1v_2v_3v_4|v_i = \{1,0\}, i \in \{1,2,3,4\}\}$ and a subset $A = \{1000, 1001\}$. We have predicate p_1 representing subset A.

$$p_1 = \begin{cases} true, \ if \ v_1v_2v_3v_4 \in A; \\ false, \ if \ v_1v_2v_3v_4 \notin A. \end{cases}$$

A predicate also can be represented as a Boolean expression. For example, for predicate p_1, we have Boolean expression: $p_1 = v1 \wedge \neg v_2 \wedge \neg v_3$. In the form of Boolean expressions, predicates can conjunct or disjunct with each other. For example: given a predicate $p_2 = v_1 \vee \neg v_1$, $p_1 \wedge p_2 = (v1 \wedge \neg v_2 \wedge \neg v_3) \wedge (v_1 \vee \neg v_1) = p_1$. The conjunction and disjunction of predicates represent the intersection and union of the sets represented by predicates.

Now, we give a definition of predicate in this paper: a predicate is a Boolean formula where each variable represents one bit in IP address value sequences. A predicate represents a set of packets for which the predicate evaluates to *true*.

A predicate that equals to *true* represents the set of all possible packets, namely the packet space. A predicate that equals to *false* represents an empty set. The conjunction of predicates represents the intersection of the corresponding packet sets.

Predicates are represented by binary decision diagrams (BDDs). BDD is a rooted, directed, acyclic graph, which consists of several decision nodes and terminal nodes. Conjunction and disjunction on predicates can be efficiently implemented by graph manipulation algorithms on BDDs [39], [40], [41].

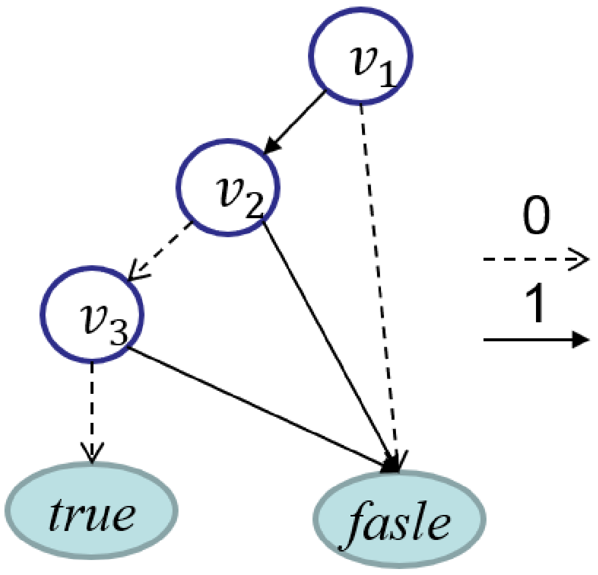

FIGURE 6.1: An example of BDD subgraphs representing a prefix match field $100x$.

Fig. 6.1 shows an simple example of BDD subgraphs representing predicate p_1. The field has 4 bits represented by variables v_1, v_2, v_3, v_4. A dotted edge denotes an assignment to false and a solid edge denotes an assignment to true. Notice that different variables orders may generate different BDD graphs with different numbers of nodes, thus we order variables by the order of bits in the original sequence.

The IP addresses can be easily converted to predicates by preforming Algorithm 2. We preprocess each IP address to a binary string, then we examine each bit of the string to form a predicate. If the bit equals 1 or 0, the corresponding predicates' variable will be set to true or false respectively, otherwise the variable will be skipped. The predicate is formed by logical conjunction of each variables.

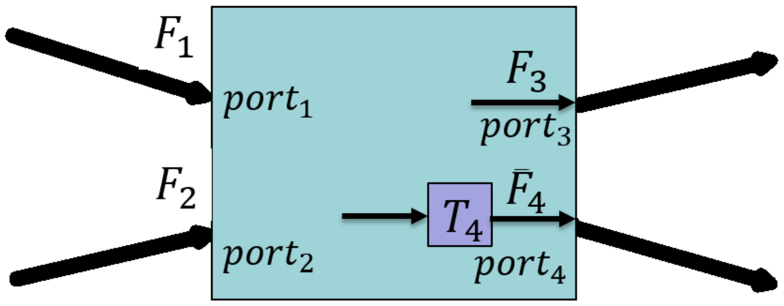

FIGURE 6.2: An example of the box model

Algorithm 2 Converting IP addresses to predicates.

Input: A binary IP address string, IP_str.

Output: A predicate, P.

```
$P \leftarrow \emptyset$; {$P$ is a predicate.}
$v \leftarrow \emptyset$; {$v$ is a predicates' variables.}
for $i \leftarrow 1$ to $len(IP\_str)$ do
  switch ($IP\_str[i]$)
  case 1:
    $P \leftarrow P \wedge v[i]$;
  case 0:
    $P \leftarrow P \wedge \neg v[i]$;
  case $x$:
    Skip;
  end switch
end for
```

With predicate, we can represent the packets sets. We use a box model to represent a switch. Fig. 6.2 shows an example of a box with 2 input ports and 2 output ports. $F_1, ..., F_4$ are predicates represented packets set for each port. T_4 is a packet transformer which can modify packet head information. To describe the transmission path of packet flows, namely, flow paths, we construct a directed graph of all possible flow paths in the network. We call it *forwarding graph.*

6.1.2 Forwarding Graph

We model a packet network as a directed graph of port nodes. Each port node is an input port or an output port of a packets forwarding device, which is referred to as a switch. Each switch has several input port nodes and output port nodes connected to another switch's port nodes. Inside of each switch, the input port node is connected to

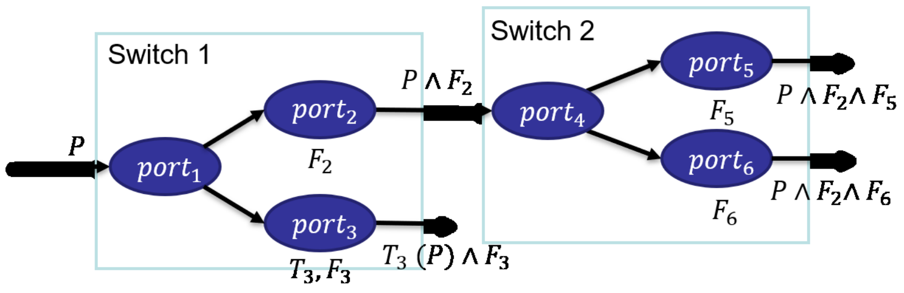

FIGURE 6.3: An example of forwarding graph.

all output port nodes. Port nodes are identified by port number. Each port node has a predicate representing the flow table.

We aggregate those predicates assigned to the same port into a forwarding predicate, F_i where i denotes the port. For those rules with *set* action which can modify packet header, we refer it as packet transformer T_i where$i$ denotes the specific port. T_i is a function that maps a predicate to another predicate to represent the procedure of modify an input packet set to an output packet set and send out from port i. Given a packet transformer T and a predicate P specifying its input packet set, $T(P)$ denotes the transformed predicate specifying the output packet set.

For a formalization of the pervious, we define a Forwarding Graph $FG = (V, A)$:

- V: a set of nodes which represents input ports and output ports of packet forwarding devices (e.g. switches) in the network. Forwarding predicates and packet transformers are stored in corresponding port nodes.
- A is a set of ordered pairs of nodes, denoted as $A = \{(x, y) \mid x, y \in V\}$. We call a ordered pair a *link*, which means port x is physically connected to port y, or x,y belong to the same switch.

Fig. 6.3 shows an example of forwarding graph. In this example, there are 2 switches, each has 3 port nodes. switch 1 is connected to switch 2 by link $(port_2, port_4)$. Given a predicate P which specifies a set of packets, packets set sent out from port 5 is represented by predicate $P \wedge F_3 \wedge F_5$.

With the forwarding graph and the predicates, we are able to compute the packets reachability. However, conjunction and disjunction operations on predicates are computation-intensive, in the worst case, the computation time is $O(2^n)$ [10], where n is the number of variables in the predicates. To simplify the computation and speed up the verification, we adopt the concept of *atomic predicates* [10].

6.1.3 Atomic Predicates

To speed up the computation of predicates, we want each predicate to be unique to avoid overlaps, also keeping the total number of predicates minimum. This idea leads us to atomic predicates. Atomic predicates are a set of predicates, which can use combination of a small number of predicates to represent all predicates of other predicates set. Each predicate is unique, and the total number of atomic predicates is the minimum. The formal definition is shown in below.

Definition (Atomic Predicate): Given a set of predicates $P = \{p_1, p_2, ..., p_n\}$, if a set of predicates, represented as $\{a_1, ..., a_m\}$, satisfies the following five properties, we call predicates $a_1, ..., a_m$ the atomic predicate of P, and predicate set $\{a_1, ..., a_m\}$ the atomic predicate set of P, denoted $A(P)$:

(1) $\forall i \in 1, ..., m,\ a_i \neq fasle$;

(2) $\vee_{i=1}^{m} a_i = ture$;

(3) $if\ i \neq j,\ a_i \wedge a_j = fasle$;

(4) $if\ p \in P,\ p \neq fasle,\ then\ p = \vee_{i \in S(p)} a_i,\ where\ S(p) \subseteq \{1, ..., m\}$;

(5) m is the minimum number such that the set A satisfies the above four properties.

In the definition, property 1 ensures that no atomic predicate represents an empty set. Property 2 ensures that the atomic predicates set can represent the set of all possible variable sequences. Property 3 ensures that each atomic predicate is unique. Property 4 means that we can use combinations of atomic predicates to represent any predicates of original predicates set which is not equal to $false$. Note that if $p = true$, then $S(p) = 1, ..., m$; if $p = false$, then $S(p) = \emptyset$. Correspondingly, an atomic predicate also can be represented in the form $q_1 \wedge q_2 \wedge ... \wedge q_n,\ where\ q_i \in \{p_i, \neg p_i\}$. Property 5 ensures the atomic predicate set has minimum number of atomic predicates.

If each atomic predicate is identified by an integer $i \in \{1, ..., m\}$, predicate p corresponds to a set of integers $S(p) \subset \{1, ..., m\}$. Thus, the conjunction and disjunction operations of predicates are replaced by the intersection and union operations of sets of integers, which are more efficient. Moreover, the set of atomic predicates is much smaller than original predicates, which can help us reducing storage costs of predicates. For example, given a predicates set $P = \{p_1, ..., p_5\}$ and its atomic predicates set $A(P) = \{a_1, ..., a_4\}$, $p_1 = a_1$ and $p_2 = a_2 \vee a_3$, we can use integer set $S(p_1) = \{1\}$, $S(p_2) = \{2, 3\}$ to represent predicate p_1 and p_2. Thus, $p_1 \wedge p_2$ can be computed by $S(p_1) \cap S(p_2)$.

We combine the atomic predicates with the forwarding graph introduced before. The benefit of using atomic predicates is that we can use sets of integers to represent and compute bit sequences, which are IP address sequences in this paper. Thus, the atomic predicates are only related to how we represent IP address sequences. Fig. 6.4 shows an example of forwarding graph with atomic predicates. Instead of using original predicate, we use a integer set which identify the atomic predicates. In this example, given a predicates set P, which has a set of atomic predicates, represented as $A(P) = \{a_1, a_2, a_3, a_4\}$, the integer identifier set is $S(P) = \{1, 2, 3, 4\}$. In the forwarding graph, the original predicate P is replaced by a integer set $\{1, 2, 3, 4\}$. Each integer denotes a atomic predicate p_i, that means $P = p_1 \vee p_2 \vee p_3 \vee p_4$. Packets set sent out from port 5 is represented by predicate p_1.

To compute the atomic predicates set, we need to consider 2 situations, atomic predicates set for one predicate, and atomic predicates set for multiple predicates. It is easy to compute the atomic predicates set for only one predicate. Given a predicate p, according to the definition, its atomic predicates set $A(\{p\})$ is shown in below:

$$A(\{p\}) = \begin{cases} \{true\}, if P = false or true; \\ \{P, \neg P\}, otherwise. \end{cases} \tag{6.1}$$

Given a single predicate p and its atomic predicate set $A(\{p\}) = \{a_1, ..., a_m\}$. If $p = false$, according to property 5, we start from $m = 1$. According to properties 1, 2, we have $a_1 = true$ which satisfies other properties. Since property 5 requires the minimum m, $A(\{p\}) = \{true\}$. Similarly, If $p = true$, then $A(\{p\}) = \{true\}$. If $p \neq false$ and $p \neq true$, starting from $m = 1$, we have $A(\{p\}) = \{p\}$ according to property 4, which does not satisfy other properties. If $m = 2$, we have $a_1 = p$ according to property 4 and $a_2 = \neg p$ according to properties 1, 2, which satisfies other properties. Hence $A = \{p, \neg p\}$.

As for atomic predicates set for multiple predicates, for example, predicate sets P_1 and P_2, we first obtain their own atomic predicate sets $A(P_1) = \{a_1, ..., a_l\}$ and $A(P_2) = \{b_1, ..., b_m\}$. Then we compute the atomic predicate set of P_1 and P_2 $\{c_1, ..., c_n\}$ by following formula.

$$\{c_i = a_j \wedge b_k | c_i \neq false, j \in \{1, ...l\}, k \in \{1, ..., m\}\} \tag{6.2}$$

Now, given a predicates set $P = p_1, ...p_n$, to compute its atomic predicates set $A(P)$, we first compute the atomic predicate set for each predicate in P. Then, we need to keep computing the atomic predicate set of each two atomic predicate sets by performing

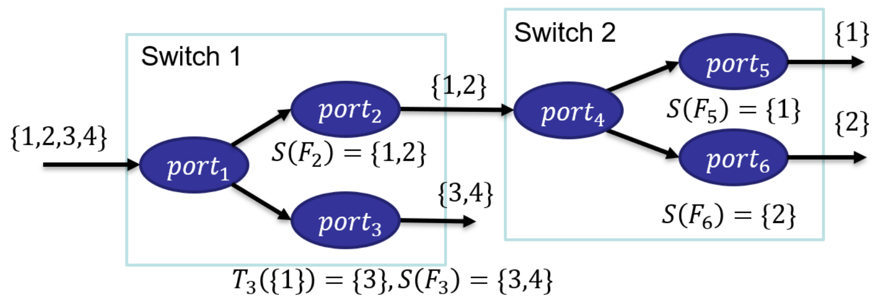

FIGURE 6.4: An example of forwarding graph with atomic predicates.

formula 6.2 until there is only one atomic predicate set left, which is the atomic predicates set that we wanted $A(P)$. Note that the computation is iterative in nature.

6.2 Proposed Data Plane Verification Scheme

As we mentioned before, the goal is to check packet reachability in the network. We achieve that with proposed network model.

Our proposed method has 5 phases, *preprocessing, computing atomic predicates, building Forwarding Graph, computing packet reachability* and *updating.* To improve verification rate in large-scale network with a large number of rules, we use parallel computing in preprocessing, computing atomic predicates phases.

Parallel computing is a type of computation that divides large problems into smaller ones, which can then be solved at the same time. Apache Spark is one of the most popular parallel computing programming frameworks. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. Spark also provides the iterative processing capability for machine learning and graphical analysis applications, which is architecturally impossible for MapReduce [22]. Spark has therefore fulfilled the need of parallel execution requirements of analytics professionals and is an irreplaceable tool in big data community [23].

Resilient distributed dataset (RDD) is the most important element to achieve Spark's function. RDD is a distributed data structure physically partitioned inside a cluster across multiple machines, but logically considered as a centralized entity.

There are two types of operations for expressing application logic in Spark: transformations, which perform operations on an input RDD to produce one or more new RDDs, and actions, which launch the series of transformations to return the final result RDD. For example, there are two operations we valued, *map* and *reduce*.

Transformation *map*: *map* is a transformation operation. Map operation applies a function to each element of the original RDD and then returns the result as a new RDD. In the Map, programmer can define his own application logic. The same logic will be applied to all the elements of RDD. Transformation map takes one element as input, then process it according to a custom function and returns one result element at a time. Thus, the original and transformed RDDs will typically have the same number of elements.

Action *reduce*: *reduce* is an action operation. Reduce aggregates the elements of an RDD according to a custom function. It is a wide operation as it is shuffling data from multiple partitions and reduces to a single value. The custom function should have two arguments of the same data type and the return type of the function also must be same as argument types.

6.2.1 Preprocessing

To convert a flow table into forwarding predicates and packet transformers, we convert the rules in the flow table first. Then, we check each forwarding rule list for each port, merge each rule into one forwarding predicate. We use Spark to convert rules into predicates and merger them into one predicate for each output port. A flowchart of converting a flow table with n rules into k predicates is shown in Fig. 6.5. The process has two phases, *Map Phase* and *Reduce Phase*.

Map Phase: The flow table in a OpenFlow switch is a list of rules. In this paper, we define a OpenFlow rule as $r :< in\_port, M, act, pri, sw >$. , where in_port is incoming switch port, $M \to \{0, 1, x\}^l$ is matching field where l is the length of the match fields and x is a wildcard bit. In our paper, M includes source and destination IP address; pri is an integer field serving as a tie-breaker in matching; act includes three actions as follows: a) $fwd(out\_port)$ represents forwarding packets out from the specified port; b) $set(addr, out\_port)$ represents forwarding packets out of the port after converting their headers information, in our scheme, M into $addr$; c) for the sake of consistency $drop$ represents forwarding packets to a particular port $port_D$.

We record a rule by using a predicate rule:$< P, P', Outport, pri >$. P is the predicate converted from header fields M. If the predicate is evaluated to be true with a 1-0 sequence x, that means packets with header value x match the rule. Each predicate is represented by a BDD, which consists of $n \leq L$ decision nodes for each variable and 2 terminal nodes for $true$ and $fasle$ where L is the number of bits in an IP address. P' denotes the $set(addr, out_port)$ action which modifies the packet header, if $P' \neq true$, implying that the original matching fields P' should be modified to P', a predicate converted from *addr*. If not, means the *act* is $fwd(out_port)$ or *drop*.

With Spark, we can accelerate the preprocessing phase. Once we read all rules into an RDD, we use *map* to convert rules to predicates. The IP addresses can be easily converted to predicates by preforming Algorithm 3. We preprocess each IP address to a binary string, then we examine each bit of the string to form a predicate. If the bit equals 1 or 0, the corresponding predicates' variable will be set to *true* or *false* respectively, otherwise the variable will be skipped. The predicate is formed by logical conjunction of each variables.

Algorithm 3 Converting IP addresses to predicates.

```
Input: A binary IP address string, $IP\_str$.
Output: A predicate, $P$.
1:  $P \leftarrow \emptyset$; {$P$ is a predicate.}
2:  $v \leftarrow \emptyset$; {$v$ is a predicates' variables.}
3:  for $i \leftarrow 1$ to $len(IP\_str)$ do
4:      switch ($IP\_str[i]$)
5:      case 1:
6:          $P \leftarrow P \wedge v[i]$;
7:      case 0:
8:          $P \leftarrow P \wedge \neg v[i]$;
9:      case $x$:
10:         $Skip$;
11:     end switch
12: end for
```

Reduce Phase: As for a flow table, we use *reduce* to convert multiple predicates into one predicate.

In an OpenFlow switch, a packet may match multiple rules, switch will choose the rule with highest priority. Notice rules with longer match fields has higher priority. To compute a forwarding predicate for each port, we first convert each rule's header fields M to a predicate stored by BDD. Then, we sort the rules in the flow table in descending order of priority and use Algorithm 4 to compute the forwarding predicate for each port.

At the beginning, we set all forwarding predicates to *false*, which indicates no packet is allowed to forward. Then we check each forwarding rule list for each port, merge each rule into one forwarding predicate. As for those packet transformers, we record the original predicates which represent the packets sets matched and the modified predicates which represent the modified packets sets.

Algorithm 4 Converting a flow table to forwarding predicates.

Input: A sorted list of rules; A set of ports, $1, ..., n$

Output: A set of forwarding predicate, $F_1, ..., F_n$. and transformer $T_1, ..., T_n$

```
1:  for $i \leftarrow 1$ to $n$ do
2:      $F_i \leftarrow false$;
3:  end for
4:  $fwd \leftarrow false$
5:  for $i \leftarrow 1$ to $m$ do
6:      if $P'_i = true$ then
7:          $F_{Outport_i} \leftarrow F_{Outport_i} \vee (P_i \wedge \neg fwd)$;
8:      else
9:          $T_{Outport_i} \leftarrow T_{Outport_i} \vee (P_i \wedge \neg fwd)$;
10:         $T_{Outport_i}(P_i) \leftarrow P'_i$;
11:     end if
12:     $fwd \leftarrow fwd \vee P_i$;
13: end for
```

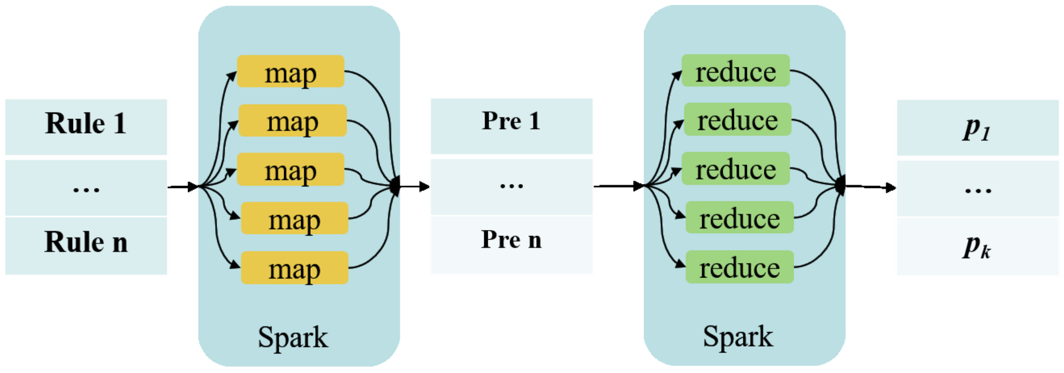

FIGURE 6.5: Flowchart of preprocessing with Spark.

6.2.2 Computing Atomic Predicates with Spark

The algorithm is pretty straightforward. Given a predicates set $P = \{p_1, ...p_n\}$, to compute its atomic predicates set $A(P)$, first, we compute the atomic predicates set for each predicate in P. Then, we compute the atomic predicates set of each two atomic predicates sets until there is only one atomic predicates set left, which is $A(P)$. A flowchart of computing the atomic predicate set of k predicates is shown in Fig. 6.6 with two phases, *Map Phase* and *Reduce Phase*.

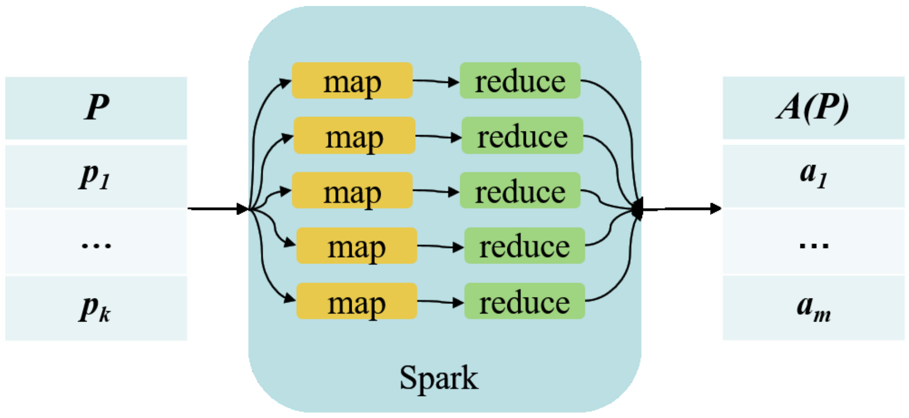

FIGURE 6.6: Flowchart of computing atomic predicates set with Spark.

Map Phase: The main task is to compute the atomic predicates set for each forwarding predicates. We first transform all predicates to a base RDD that can be operated on in parallel. The elements of the base RDD are predicates. Then we use *map* to transform each predicate to its atomic predicates set by performing the formula 6.1. A new RDD will be created. The elements of the new RDD are atomic predicates sets of the predicates.

Reduce Phase: The outputs from the map phase are aggregated in reduce phase. Action *reduce* computes formula 6.2 which takes two atomic predicates sets and returns one in parallel. Reduce phase returns only the result of the reduce which is the atomic predicates set of all predicates. Since the basic idea of atomic predicates is using sets of integers to represent predicates, what we need to do is to compute atomic predicates of all predicates presented in our network model. In our network model, a packet filter is represented by one forwarding predicate and a packet transformation is represented by two predicates, the original predicates and the modified predicates. There is no need to change the algorithm of computing atomic predicates, we just need to compute atomic predicates of all three types of predicates. Notice that the output predicate of transformer $T(P)$ also need to be computed, otherwise they may not be represented by the current atomic predicates set.

6.2.3 Building Forwarding Graph

The goal here is to construct a directed forwarding graph of all possible flow paths in the network. Nodes in this forwarding graph represent ports of switches and edges represent the dependency of the ports. A node representing an input port will be connected to all nodes which representing output ports in the same switch. For the node representing an output port, it will be connected to nodes which representing other switch's input ports if it can send packets to by a data link. The packet sets that can go through a port are represented by corresponding forwarding predicates and transformers. For the sake of simplicity, in the following paper we refer to all these nodes as ports which they represent for.

6.2.4 Computing the Reachability

We check the reachability by computing the reachability tree from each port with the forwarding graph. A reachability tree is rooted at a source port and consists every flow pathes from the source port. The path is represented by nodes which stores a port identifier and the packet set that can reach to the port from the source port.

After converting all forwarding predicates to sets of integers, we can compute reachability trees for verifying data plane properties. Computing the reachability tree from a source port detects forwarding loops for all packets injected into the source port, if any. More generally, reachability trees can be used to verify safety and progress properties specified in a temporal logic: for example, verifying that all packets injected into the source port traverse a specified sequence of required waypoints in the network. Note that loops and black holes can be found in this process.

[H] **Algorithm 5** Building Reachability Tree

```
Input: A start port, $s$; Integer sets representing Forwarding rules for each port, $S(P_1), ..., S(P_m)$;
    Transformer predicates, $T_1, ..., T_m$; The topology of the network, $Topo$.
Output: A Reachability Tree, $Re\_Tree$
 1: $Re\_Tree.AddNode(s, S(P_s))$;
 2: $visited(s) \leftarrow TRUE$;
 3: $DFS(s, S(P_s))$;
 4: return $Re\_Tree$;
 5: $DFS(t, temp)$;
 6: for $v \in Topo.Next(t)$ do
 7:   if $!visited(v)$ then
 8:     if $\exists T_v$ then
 9:       $temp \leftarrow T_v(temp)$;
10:     end if
11:     $ints \leftarrow temp \cap S(P_v)$
12:     if $int \neq \emptyset$ then
13:       $Re\_Tree.AddNode(v, ints)$;
14:       $visited(v) \leftarrow TRUE$;
15:       $DFS(v, ints)$;
16:     else
```

```
17:         if v is an output port&&t is an input port then
18:             Send Black hole detected message;
19:             Break;
20:         end if
21:     end if
22:   else
23:     Send Loop detected message;
24:     Break;
25:   end if
26: end for
```

The reachability tree from a port s to all other ports in the network is computed by performing a depth-first search (DFS) shown in Algorithm 5. During the DFS, when a search branch visiting a port x, if one of the following 3 conditions happens: 1. the integer set which represents packets that can reach port x is empty; 2. port x is an output port and it has no links connected to any input port; 3. port x has been visited before in the search. The search branch is terminated and will backtrack and continue DFS until no more port node in the network can be reached. Notice that if condition 3 happens, it means we detect a loop. When DFS finishes, a reachability tree from port s to all other ports in the network is created. Once we obtain the reachability tree, we can check the packet reachability.

Fig. 6.7 shows the reachability tree of Fig. 6.4. Node $port_1$ labeled by $1, 2, 3, 4$ forwards only packets in atomic predicates $p_1 \vee p_2 \vee p_3 \vee p_4$. $port_3$ labeled by $1, 3, 4$ transforms p_1 to p_3 then forwards packets set represented by $p_1 \vee p_3 \vee p_4$. Each node stores the reachable ports and final set of integer for each port. Node also records wither the integers set is transformed or not.

6.2.5 Application Cases

With the reachability tree, we can check packet reachability to verify 2 data plane properties, loop-freedom, nonexistence of black holes. We also can detect the firewall violation, firewall bypass threat with packet reachability.

6.2.5.1 Firewall Model

To detect firewall bypass threat, we need a model for firewall rules to work with our proposed network model. For firewall rules, we also use predicates to present packet

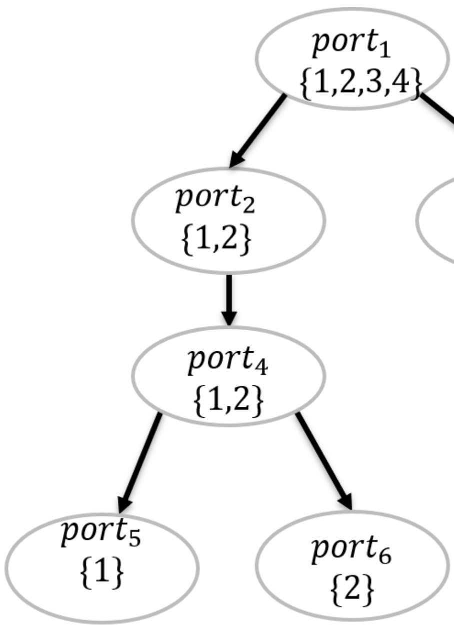

FIGURE 6.7: An example of a reachability tree.

sets. A firewall rule list is a collection of all connections denied by the firewall rules. In a firewall the DENY rules are complementary to ALLOW rules, hence we only care the DENY rules. We define a firewall rule as $< port_{src}, port_{dst}, pred_F >$, which indicates every packet with header values satisfied predicate $pred_F$ sent form $port_{src}$ to $port_{dst}$ is denied.

Considering the example shown in Fig. 4.2, we need to record both source address and destination address in our predicates. We define $pred_F = pred_{src} \wedge pred_{dst}$ where $pred_{src}$ and $pred_{dst}$ are predicates indicate source address and destination address respectively. If a address has h bits, we use variable set v_1, v_2, $...v_h$ to represent each bit of source address, and use another variable set v_{h+1}, v_{h+2}, $...v_{h+h}$ for destination address. The number of nodes in the BDD graph representing $pred_F$ is $\leq 2 + 4h$ [10].

6.2.5.2 Problem Definition

With the firewall model, we can formally define those 2 data plane properties, loop-freedom, nonexistence of black holes and the firewall violation, firewall bypass threat in our proposed network model.

Loops: A loop in the graph is equivalent to at least one strongly connected component in the directed graph. Two port nodes x and y belong to a strongly connected component, if there is a path from x to y and a path from y to x.

Firewall bypass threat: Given a firewall rule: $< port_{src}, port_{dst}, pred_F >$, $S(pred_F)$ denote the integer set of atomic predicates representing $pred_F$. If there exist a predicate

p, $S(p) \subseteq S(pred_F)$, and p can reach $port_dst$ from $port_src$ by transforming, the firewall rule is bypassed.

Black holes; A set of packets that are dropped due to no forwarding entry. For a switch node with output ports $port_1, ..., port_n$, let $S(F_1), ..., S(F_n)$ be the set of atomic predicate identifiers for each output port, $S(P)$ be the set of all atomic predicate identifiers. There is a black hole in the switch if: $S(p) - \cup_{i=1}^{n} S(F_i) \neq \emptyset$.

6.2.5.3 Verification with Reachability Tree

With the reachability tree indicating packet reachability, we can detect loops, black holes to verify loop-freedom, nonexistence of black holes. Combining with the firewall model, we also can detect firewall bypass threat.

Loop Detection: Loop detection is performed by computing the reachability tree for every port. During the DFS, if a port has been visited before in the search, that means there is a loop.

Black Hole Detection:Black Hole can be detected by calculating following set for each switch: $S(p) - \cup_{i=1}^{n} S(F_i)$. If the set is not empty, that means there is a black hole in this switch.

Detect Firewall Bypass Threat: The detection can be processed with an existing reachability tree when adding a new firewall rule or after the system initialization. Once we create a reachability tree, we can check the reachability of each firewall rule to detect the violation in case of new firewall rules are added. If violations exist, the administrator will be informed. To identify the violation with an existing reachability tree of $port_{src}$, we use Algorithm 6. First, we check whether the reachability tree of $port_{src}$ contains $port_{dst}$. If $port_{dst}$ exist in the tree, which means $port_{dst}$ is reachable from $port_{src}$, we will check the reachability of the denied packet set $pred_F$. $pred_{dst}$ will be conjuncted with atomic predicates, if the conjunction is not empty, then we traverse the reachability tree and track those atomic predicates, if they are transformed, then track the transformed atomic predicates. Finally, if those atomic predicates can reach $port_{dst}$ from $port_{src}$ by transforming, that means the firewall rule is bypassed.

[H] **Algorithm 6** Identifing the violation with a existing reachability tree

```
Input: A Reachability Tree, $Re\_Tree$; A firewall rule including $port_{src}$, $port_{dst}$, $pred_{src}$
    and $pred_{dst}$; Atomic Predicates of forwarding rules, $AP_1, ..., AP_n$).
Output: Whether the violation exists or not
1: if $port_{dst}$ exists in $Re\_Tree(port_{src})$ then
```

```
2:     if $x \in Re\_Tree(port_{src}).port_{dst}.value$ and $AP_x \wedge pred_{dst} \neq \emptyset$ {$value$ includes
       atomic predicates identifier.} then
3:        $final \leftarrow x$;
4:     end if
5:     if $AP_y \wedge pred_{src} \neq \emptyset$ then
6:        $start \leftarrow y$;
7:     end if
8:     for $i \in Path(port_{src},\ port_{dst})$ {$Path$ records the path also whether which atomic
       predicates identifier transformed or not.} do
9:        if $i = port_{dst}$ then
10:          if $start = final$ then
11:             return $Violation\ exists.$;
12:          else
13:             return $No\ violation\ detected.$;
14:          end if
15:       else
16:          if $start\ is\ transformed$ then
17:             $start \leftarrow transformed\ start$;
18:          end if
19:       end if
20:    end for
21: end if
```

Once we finished the initialization, an active detection will be executed to check potentially violation. This is designed for better deployability. Our method does not require the network to be secure before the implementation. Thus, users can deploy the detection at any period of the network.

6.2.6 Dynamic Update

An important requirement of network verification is to support dynamic changes to the network in real time. Network changes like link and rule changes require addition and deletion of predicates. We adopt the method presented in [10] to achieve rules updating by updating predicates and the corresponding graphs accordingly. These methods are also used for addition/deletion of predicates after a network link addition/deletion which causes predicates changes. There are three events as follows.

Link update: When a link goes up or down, the set of predicates and the set of atomic predicates are not changed. However, reachability trees from source ports may be affected by a link update event. We check each reachability tree to locate the corresponding nodes, then update the forwarding graph. A switch up event can be handled as a multi-links up event and a switch down event can be handled as a links down event.

Forwarding rule update: When a new rule is added, or deleted, it may change one or more predicates of some ports. Our method will check if a forwarding predicate or packet transformer is changed by the rule update; if so, it computes a new predicate for the port and updates the set of atomic predicates. If the set of atomic predicates remains unchanged, the reachability tree will be updated. Otherwise, the reachability tree will be completely recomputed. The updated predicates and reachability tree will remain temporary before being checked against the firewall rule during building, if violation detected, the update will be canceled and the administrator will be informed.

Firewall rule update:: When a new firewall rule is added, new violation may arises. Before updating the firewall rule, the reachability tree will be checked against the new firewall rule, if violation detected, the administrator will be informed. Obviously, deleting a firewall rule will not introduce any new firewall policy violation.

6.3 Implementation and Performance Evaluation

We implement our method and run experiment based on Ubuntu Linux operation system version 18.04, the system is hosted on an Intel Core i5-6500 3.20GHz CPU with 7.7GB RAM.

Table 6.1: Experimental Datasets Information.

	No. of hosts	**No. of rules**
Dataset 1	4	20
Dataset 2	8	88
Dataset 3	16	368
Dataset 4	32	752

We run Spark in local mode. Spark has an advanced Directed Acyclic Graph (DAG) execution engine that supports acyclic data flow and in-memory computing. It can run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk. In local mode, Spark spawns all the execution components - driver, executor, and master in the same single Java Virtual Machine (JVM). The default parallelism is the number of threads as specified in the master URL. We run Spark locally with as many worker threads as logical cores on our machine, in our experiment, 4 threads. This is the only mode where a driver is used for execution.

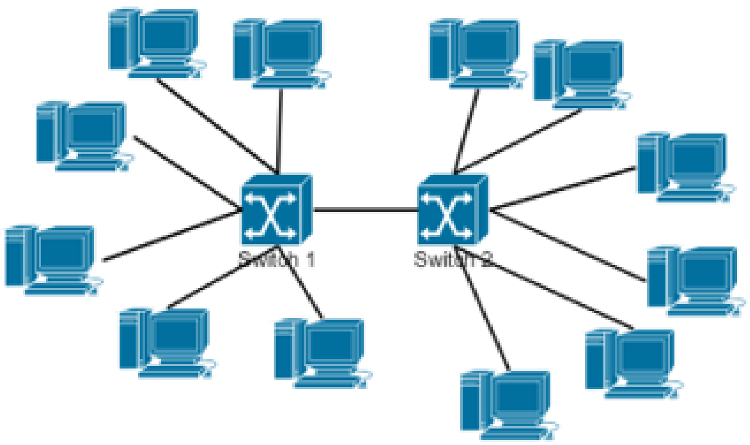

FIGURE 6.8: A topology example of the datasets.

We generate four datasets to evaluate the performance. Those datasets share the same topology, shown in Fig. 6.8, two switch connected to each other and serval hosts. We use a network simulator Mininet [36] and an open source OpenFlow controller Floodlight [13] to generate the data. Mininet is a fast, lightweight but powerful network emulator. Mininet is capable to create a realistic virtual network running functional hosts and switches. Mininet also provides the capability of interacting with user's custom OpenFlow controller. We choose Floodlight as our controller because it has a number of sample module applications, such as a leaning switch and a firewall. Floodlight comes with a web based and Java based GUI and most of its functionality is able to access through a REST API. Floodlight is able to automatically find the shortest path for table-miss packets, adds the appropriate flow entries into related switches, and lets the switch continue its forwarding.

Floodlight also has a firewall module which enforces ACL rules by monitoring Packet_IN messages. ACL rules here are set of conditions that allow or deny a traffic flow at its ingress switch.

The detailed statistics are list in Table 6.1. In all of our experiments, we use our reachability algorithms to compute reachability of each pair of ports and measure the average time overhead of building the forwarding graph and computing reachability of on pair of ports.

For efficiency comparison, we modify the detection method in our previous research [42] which is a simplified HSA-based method designed to verify firewall bypass threat in SDN networks. We choose it since it shares the similar proposes with the proposed method which are packet reachablity verification and firewall bypass threat detection. To our best knowledge, it is capable to represent similar HSA-based methods. The previous research uses a graph-based network model also called as forwarding graph. In the

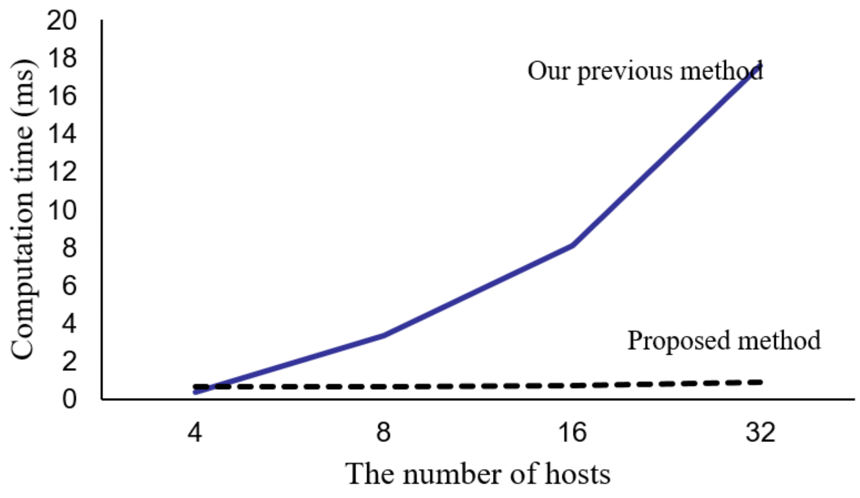

FIGURE 6.9: Computation time (ms) of initialization.

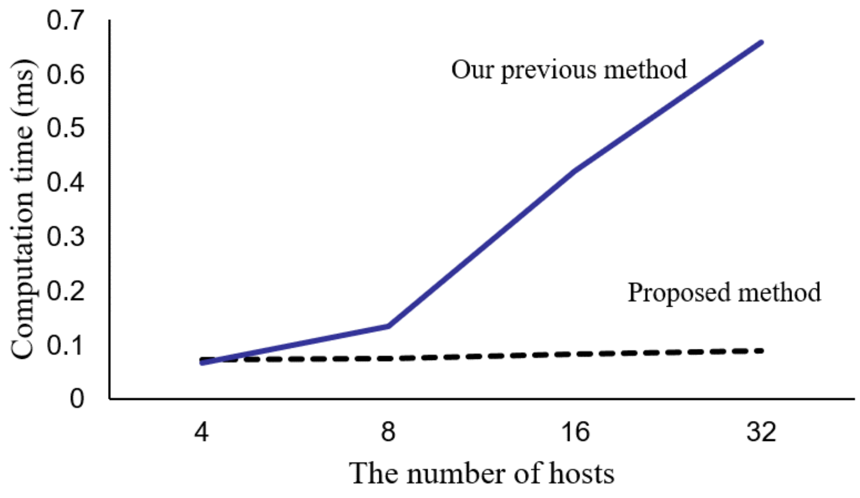

FIGURE 6.10: Computation time (ms) of reachablity computation.

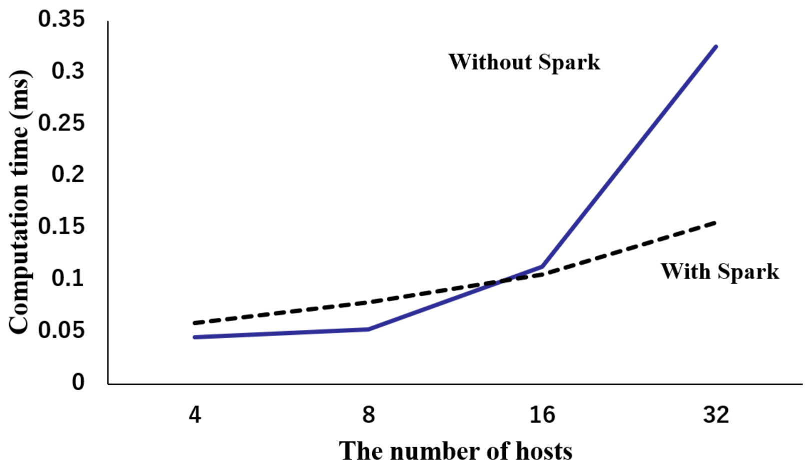

FIGURE 6.11: Computation time (ms) of computing atomic predicates.

previous forwarding graph, each node represents a rule with a IP address bit sequence. As we mentioned before, HSA-based methods suffer the same disadvantage that when the number of rules increases, the number of nodes increases accordingly. It will result in a huge graph which is difficult to achieve fast verification and update. In this paper, we use ports instead of rules as nodes to build the graph model. Thus, the graph size will remain relatively smaller comparing to the previous research. Meanwhile, since we use predicates instead of bits sequences to represent rules, we can achieve faster rule verification. Note, in our experiments, we force our method to evaluate each incoming flow entry against each firewall policy without any whitelisting rules, in practice, this kind scenario could represent a worst case. The results from this experiment are shown in Fig. 6.11 and Fig. 6.10. It can be seen that proposed method is much faster than the previous method whether in initialization or reachablity computation, especially when dataset scale up. This is thanks to atomic predicates which compress the redundancy of rules so the forwarding graph will have much less nodes than the previous method, therefore, speed up the computation.

We compare the computation time of preprocessing and computing atomic predicates with or without Spark. The results are shown in Fig. 6.11. Due to Spark's own system overhand, it does not outperform the original computation with small dataset, but it is faster while dealing with bigger dataset. And note that, we run Spark in local model. Within nowadays data center, where likely has Spark deployed in clusters due to its popularity, the proposed method is able to achieve faster computation and better scalability.

TABLE 6.2: Storage overheads of reachability trees of all ports (bytes).

	Proposed Method	**Previous Method**
Dataset 1	6400	5930
Dataset 2	7820	8875
Dataset 3	18323	325945
Dataset 4	32517	720986

We also compare the storage overhead of our proposed method and previous method. We calculate the memory usage for storing reachability trees of all ports in proposed method and the whole forwarding graph, which consists all ports' reachability information, in previous method. The results are presented in Table 6.2. It can be seen that proposed method is more space efficient than previous method along with increasing rules in datasets.

To test the firewall violation detection mechanism, we adopt the method used in [32]. Our goal is to measure the delay caused by detecting the firewall bypass threat. We simulate the scenarios described in Chapter 4 by modifying the datasets we described

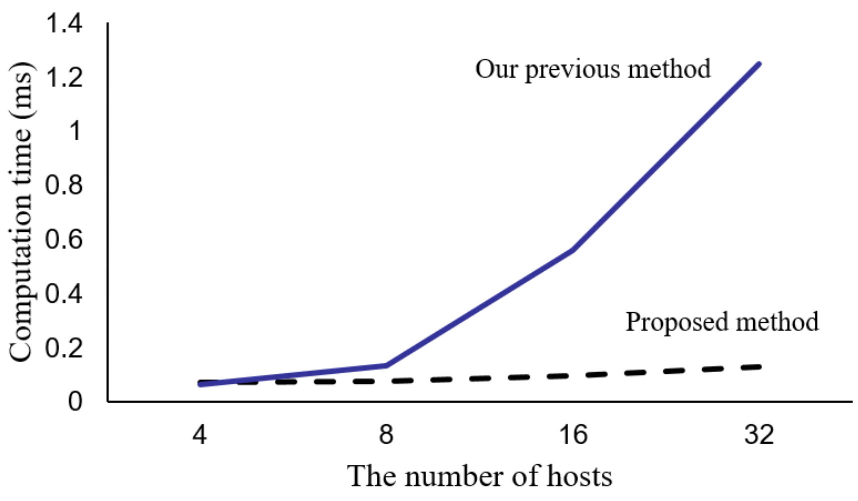

FIGURE 6.12: Computation time (ms) of detecting firewall violation computing time.

above. We add a set of candidate forwarding rules with transform operation to allow 2 hosts to connect to each other. Then we set a firewall rule to block the communication between those 2 hosts. We check those candidate rules against the firewall rule to simulate rule update events. The result is shown in Fig. 6.12. The simulation results show that our method is able to detect the firewall bypass threats and reject those update requests. Since the algorithm is similar to computing reachability, it is not hard to tell that our method is faster than the previous method.

6.4 Conclusion

Network errors in data plane are very common in nowadays network. New security challenges also come out with new technology. In this paper, we propose an atomic predicates based data plane properties verification method to verify data plane properties for SDN using Spark. By modeling the network as a directed graph, the problem of packet reachability verification is formulated to the graph reachability computation problem. We adopt the concept of atomic predicates so that we can compute packet reachability efficiently by compressing the redundancy of rules by representing a very large set of packets by a unique predicate, which can be identified by a single integer. This reduces the number of nodes in the forwarding graph and the computation time of packet behavior. We also adopt the parallel process computational framework Spark to accelerate the computation and achieve scalability. With the results of packet reachability verification, we can detect loops, black holes to verify 2 data plane properties, loop-freedom and nonexistence of black holes. Combining with the firewall model, we also can detect a type of access control violation called firewall bypass threat.

Our experimental results demonstrate that proposed method is much faster and space efficiency for computing reachability than our previous method. Compared to other data plane properties verification methods, proposed method is not only capable of verifying data plane properties including loop-freedom, nonexistence of black holes, but also can detect firewall bypass threat. We also discussed the performance of computing atomic predicates with Spark. Compared to the method without Spark, the method with Spark shows its advantage in scalability with bigger datasets and clusters.

Chapter 7

Atomic Predicate Based Network Property Verification for SDN Networks

In this research, we address the challenge of designing a fast network property verification tool for large scale SDN networks. We propose a novel detection method based on modeling the network to a directed graph. The problem is formulated to the reachability computation problem. We adopt the concept of atomic predicates and AP Tree so that we can detect loops and black holes efficiently. We also adopt the parallel process computational framework Spark to accelerate the computation and achieve scalability. Our experimental results demonstrate the proposed method is much faster for computing reachability than our previous method.

7.1 Network Model and Problem Statement

In this paper, we represent packets with atomic predicates, and describe network connections with a directed graph.

7.1.1 Network Model

In SDN networks, a data packet is a formatted unit of data routed between a source and a destination. Each packet has a header of h bits. The header bits are partitioned into multiple fields. We only focus on those reachability relevant fields, in particular, Internet Protocol (IP) address in this paper. The value of each field is represented by a sequence of bit variables. When a packet arrives at a switch, the packet header will be parsed to match against forwarding rules in the flow table which is a list of rules. If the packet header matched the header fields of a rule, the switch will process the packet according to the actions defined by the rule. Packets with identical header values will be processed identically.

We use the value of packet header field to represent a set of packets, particular, we use IP address fields values to represent the set of all packets which sharing the corresponding values. For example, by destination IP version 4 address value 10.1.1.1/32, we represent all packets sent to 10.1.1.1/32. We abstract away the data portion of a packet since we assume that it does not affect packet forwarding. To represent sequences, we use predicates.

7.1.2 Predicate

A predicate is a Boolean function which is evaluated to be true or false depending on the values of its variables. We use predicates as indicator functions.

A predicate also can be represented as a Boolean expression. For example, for predicate p_1, we have Boolean expression: $p_1 = v1 \wedge \neg v_2 \wedge \neg v_3$. In the form of Boolean expressions, predicates can conjunct or disjunct with each other. For example: given a predicate $p_2 = v_1 \vee \neg v_1$, $p_1 \wedge p_2 = (v1 \wedge \neg v_2 \wedge \neg v_3) \wedge (v_1 \vee \neg v_1) = p_1$. The conjunction and disjunction of predicates represent the intersection and union of the sets represented by predicates.

Now, we give a definition of predicate in this paper: a predicate is a Boolean formula where each variable represents one bit in IP address value sequences. A predicate represents a set of packets for which the predicate evaluates to $true$.

A predicate that equals to $true$ represents the set of all possible packets, namely the packet space. A predicate that equals to $false$ represents an empty set. The conjunction of predicates represents the intersection of the corresponding packet sets.

Predicates are implemented as binary decision diagrams (BDDs). BDD is a rooted, directed, acyclic graph, which consists of several decision nodes and terminal nodes. Conjunction and disjunction on predicates can be efficiently implemented by graph manipulation algorithms on BDDs [39], [40], [41].

IP addresses can be easily converted to predicates. We preprocess each IP address to a binary string, then we examine each bit of the string to form a predicate. If the bit equals 1 or 0, the corresponding predicates' variable will be set to true or false respectively, otherwise the variable will be skipped. The predicate is formed by logical conjunction of each variables.

We model an SDN network as a directed graph of switches. Each switch has serval input ports and output ports connected to other switches or hosts. Switches are identified by switch IDs. Each output port has a predicate representing the packets set that can go through this port according to corresponding rules. Those predicates are called forwarding predicates.

For a formalization of the pervious, we define a Forwarding Graph $FG = (V, A)$:

- V: a set of nodes which represents switches and hosts in the network.
- A is a set of ordered pairs of nodes, denoted as $A = \{(x, y) \mid x, y \in V\}$. We call a ordered pair a *link*, which means node x is physically connected to node y.

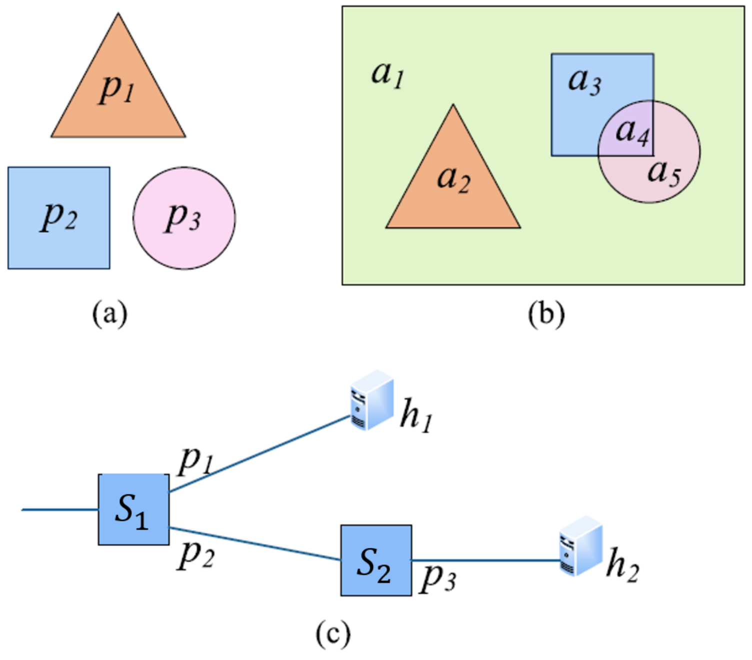

FIGURE 7.1: (a) Three predicates. (b) The packet header space and five atomic predicates. (c) A sample network including the three predicates.

7.2 Problem Statement

Since SDN allows network applications to operate with switches in the networks directly, it faces a variety of management challenges. In this paper, we focus on checking reachability and verifying loop-freedom and nonexistence of black holes. The goal is to check packet reachability in the network. In our network model, *reachability* means that can a packet with specific header values sent from node x reach node y. If the packet can reach node y through a certain path, we say node y is *reachable* from node x. Like the other tools, the reachability check is the cornerstone of our method. As the result of computing the reachability of a packet flow, we can obtain the whole transmission path of the flow, which is used for network properties analysis.

The set of packets that can travel through a sequence of switches can be represented by the conjunction of the forwarding predicates. For example, packet set arrived at h_2 form s_1 in Fig. 7.1 can be computed and represented by $p_2 \wedge p_3$. Since computing conjunction is computation-intensive. To enhance the performance we use atomic predicate to simplify the computation.

7.2.1 Atomic Predicate

The conjunction and disjunction of predicates are highly computation-intensive since they operate on multidimensional sets. To simplify the computation and speed up the verification, the concept of *atomic predicate* is introduced in [10]. The main idea is to use combination of a small number of predicates to represent any predicate of another predicates set. This kind small number of predicates are called as atomic predicates. The set of all atomic predicates is called as atomic predicates set. Each atomic predicate is unique, and the total number of atomic predicates is the minimum.

The formal definition is shown in below.

Definition (Atomic Predicate): Given a set of predicates $P = \{p_1, p_2, ..., p_n\}$, if a set of predicates, represented as $\{a_1, ..., a_m\}$, satisfies the following five properties, we call predicates $a_1, ..., a_m$ the atomic predicate of P, and predicate set $\{a_1, ..., a_m\}$ the atomic predicate set of P, denoted $A(P)$:

(1) $\forall i \in 1, ..., m, \ a_i \neq fasle$;

(2) $\vee_{i=1}^{m} a_i = ture$;

(3) $if \ i \neq j, \ a_i \wedge a_j = fasle$;

(4) $if \ p \in P, \ p \neq fasle, \ then \ p = \vee_{i \in S(p)} a_i, \ where \ S(p) \subseteq \{1, ..., m\}$;

(5) m is the minimum number such that the set A satisfies the above four properties.

If each atomic predicate is identified by an integer $i \in \{1, ..., m\}$, predicate a corresponds to a set of integers $S(P) \subset \{1, ..., m\}$. Thus the conjunction and disjunction operations of predicates are replaced by the intersection and union operations of sets of integers, which are more efficient. Moreover, the set of atomic predicates is much smaller than original predicates, which can reduce storage costs of predicates.

With proposed network model, forwarding graph and predicates, we can compute the reachability of a packet with specific header values. As an illustration, Fig. 7.1(a) shows three forwarding predicates p_1 (triangle), p_2 (square), and p_3 (circle), each of which represents a set of packets that are evaluated to true by a predicate. Each forwarding predicate specifies a set of packets that can pass the corresponding rules. Fig. 7.1(b) shows the three predicates in the packet space. All packets in this example are specified by five atomic predicates, a_1 to a_5. Each predicate is equal to the disjunction of a subset of atomic predicates. For example, $p_2 = a_3 \vee a_4$. According to the definition, $a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5 = true$. Atomic predicate also can be represented in the form $q_1 \wedge q_2 \wedge ... \wedge q_n,\ where\ q_i \in \{p_i, \neg p_i\}$. This form characterizes the behaviors of all packets it evaluates to true by the atomic predicate. For example, $a_4 = \neg p_1 \wedge p_2 \wedge p_3$ represents the packet behavior along each port: all packets evaluated to true by a_4 can pass the flow table of p_2 and p_3 but cannot pass p_1. We call it the packet behavior expression.

Combine the packet behavior expression and forwarding graph, we can obtain the forwarding path for a packet set. In the network shown in Fig. 7.1(c), Let p_1 specify the set of packets that can be forwarded at switch s_1 to its output port to host h_1, p_2 specify the set of packets that can be forwarded at switch s_1 to its output port to switch s_2, and p_3 specify the set of packets that can be forwarded at switch s_2 to its output port to host h_2. A packet specified by $a_4 = \neg p_1 \wedge p_2 \wedge p_3$ is forwarded at s_1 by the path $s_1 > s_2 > h_2$. A packet specified by $a_5 = \neg p_1 \wedge \neg p_2 \wedge p_3$ is forwarded to h_2 if it is at s_2, but will be dropped if it is at s_1.

7.2.2 AP tree

To efficiently obtain the packet behavior expression, we use Atomic Predicate (AP) tree. The AP Tree is a data structure introduced in [11] to quickly classify a packet to an atomic predicate.

An AP Tree is a binary tree which nodes are labeled by predicates. Starting from the root, at each internal node, the input packet is evaluated by the predicate in the label. If the result is true, the packet continues to be evaluated in the left subtree. Otherwise it goes to the right sub-tree. An AP Tree with $(k+1)$ levels is constructed from evaluating each of the k predicates at each level of internal nodes. A leaf node is then labeled by $q_1 \wedge q_2 \wedge ... \wedge q_k,\ q_i \in \{p_i, \neg p_i\}$, which specifies the set of packets reaching the leaf. Fig. 7.5(a) shows the AP Tree of the three predicates in Fig. 7.1(b). Shaded nodes indicate leaf labels that are false, in other words, representing empty set. By searching this AP tree, we can quickly obtain the packet behavior expressions for atomic predicates.

7.2.3 Loop-freedom and Nonexistence of Black Holes

We verify specific two essential network properties, loop-freedom and nonexistence of black holes. In SDN network, a loop could occur when mistakenly installing forwarding rules. If a packet is sent into a looped path, it can loop until its time to live (TTL) value expired. On the other hand, a black hole could happen when mistakenly deleting a forwarding rule. If a packet is forwarded to a switch with no match forwarding rules, it will be dropped without informing the source. With the network model, we define loops, black holes as follow.

Loops: A loop in the graph is equivalent to at least one strongly connected component in the directed graph. Two nodes x and y belong to a strongly connected component, if there is a path from x to y and a path from y to x.

Black holes; A set of packets that are dropped due to no rule matches. For a switch node connected to other switches, if a packet cannot be determined at it, we conduct that there is a black hole in this switch.

7.3 Proposed Method

As we mentioned before, the goal is to check packet reachability in the network. We achieve that with proposed network model.

Our proposed method has 5 phases, *preprocessing, computing atomic predicates, AP Tree construction, computing packet reachability* and *updating.* To simplify the computation and speed up the verification, we combine the concept of atomic predicates and Spark into *preprocessing, computing atomic predicates, AP Tree construction* phases.

7.3.1 Preprocessing

To convert a flow table into forwarding predicates, we convert the rules in the flow table first. Then, we check each forwarding rule list for each port, merge each rule into one forwarding predicate.

With Spark, we can accelerate the whole preprocessing. Once we read all rules, we use *map* to convert multiple rules to predicates parallely. As for a flow table, we use *reduce* to convert multiple predicates into one predicate.

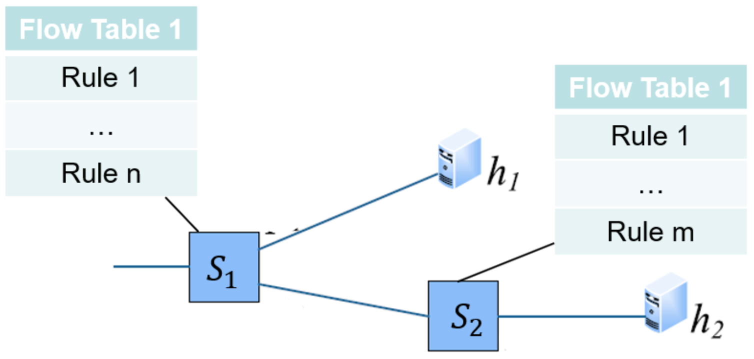

FIGURE 7.2: A small network consists two switches.

For example, Fig. 7.2 shows a small network consists two switches s_1, s_2, each has a flow table with serval forwarding rules. We use Spark to convert rules into predicates and merger them into one predicate for each output port. The flow chart is shown in Fig. 7.3.

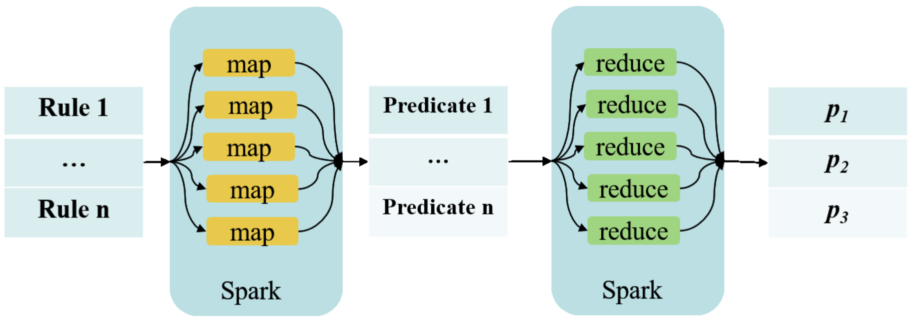

FIGURE 7.3: Flow chart of preprocessing with Spark.

7.3.2 Computing Atomic Predicates

Given a predicates set $P = p_1, ...p_n$, to compute its atomic predicates set $A(P)$, first, we compute the atomic predicates set for each predicate in P. Then, we compute the atomic predicates set of each two atomic predicates sets until there is only one atomic predicates set left, which is $A(P)$.

To compute the atomic predicate set, we use *map* to generate the atomic predicate set for each forwarding predicate and then use *reduce* to compute the final atomic predicate set via performing the computation of two atomic predicate sets. The flow chart is shown in Fig. 7.4.

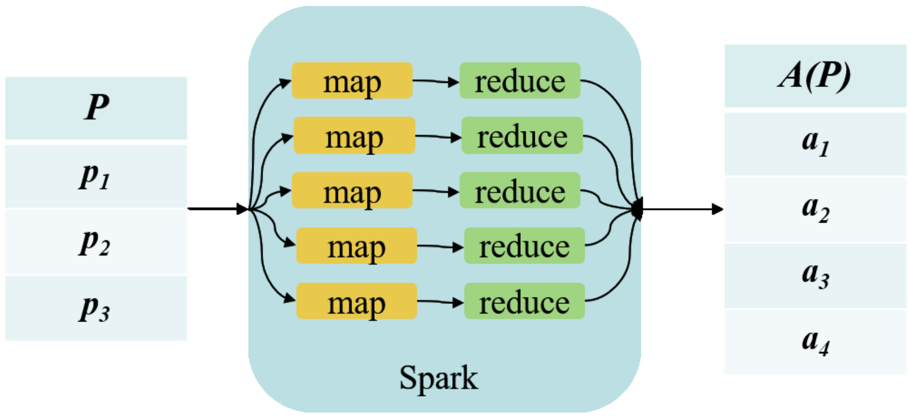

FIGURE 7.4: Flow chart of computing atomic predicates set with Spark.

7.3.3 AP Tree Construction

To classify a packet to an atomic predicate and obtain the packet behavior expression, we simply searches the AP Tree by evaluating the packet until the leaf labeled by the atomic predicate is found.

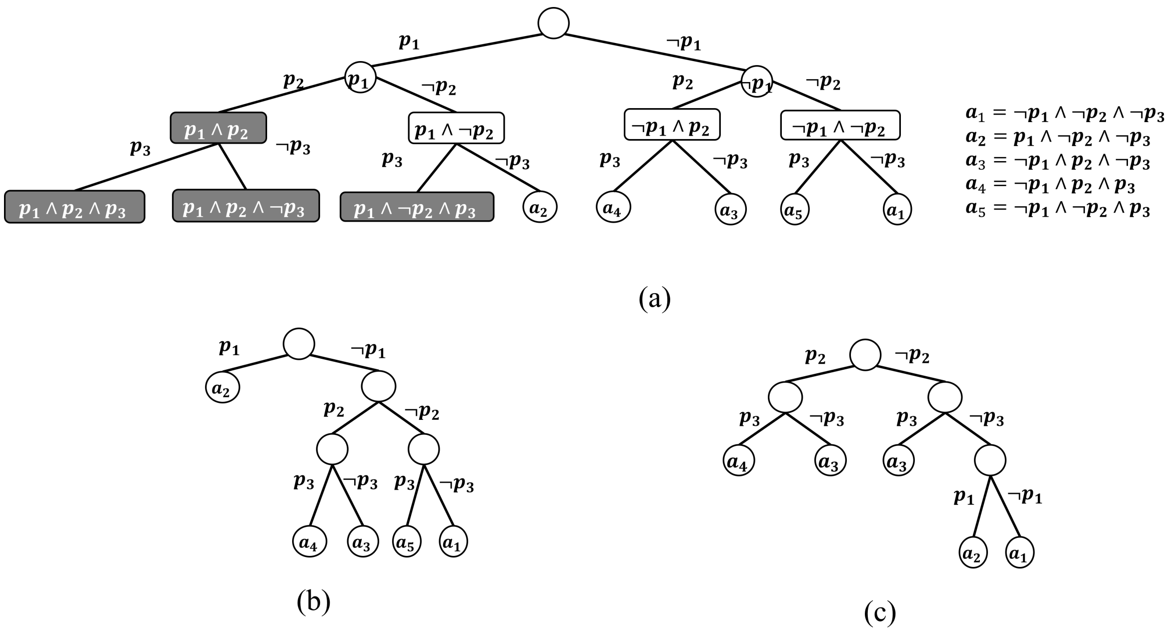
p_1
$\neg p_1$
p_2
$\neg p_2$
p_3
$\neg p_3$
$p_1 \wedge p_2$
$p_1 \wedge \neg p_2$
$\neg p_1 \wedge p_2$
$\neg p_1 \wedge \neg p_2$
$p_1 \wedge p_2 \wedge p_3$
$p_1 \wedge p_2 \wedge \neg p_3$
$p_1 \wedge \neg p_2 \wedge p_3$
$a_1 = \neg p_1 \wedge \neg p_2 \wedge \neg p_3$
$a_2 = p_1 \wedge \neg p_2 \wedge \neg p_3$
$a_3 = \neg p_1 \wedge p_2 \wedge \neg p_3$
$a_4 = \neg p_1 \wedge p_2 \wedge p_3$
$a_5 = \neg p_1 \wedge \neg p_2 \wedge p_3$
(a)
(b)
(c)

FIGURE 7.5: AP Tree of predicates in Fig. 7.1 (b). (a) Original AP Tree. (b) Pruned AP Tree. (c) Optimized AP Tree.

The depth of a leaf is defined to be the number of predicates evaluated to reach the leaf [11]. In the worst case, for an AP tree with 2^k atomic predicates, finding a leaf needs to evaluate all k predicates. Reducing the average depth of leaves can reduce the query time. Therefore construction optimization is important to AP tree performance.

AP Tree can be optimized by removing a sub-tree if all leaves in the subtree are labeled by false since no packet can reach any of these three leaves. For example, in Fig. 7.5(a), $p_1 \wedge p_2 \wedge p_3, p_1 \wedge p_2 \wedge \neg p_3$, and $p_1 \wedge \neg p_2 \wedge p_3$ are all false according to the relationships in Fig. 7.1(b), which specify empty sets of packets. After the "empty" nodes are removed, internal nodes with only one child will be removed since there are no need to check them. Fig. 7.5(b) shows the pruned AP Tree has average depth (1+3+3+3+3)/5 = 2.6. The average depth of the AP Tree also may be different if predicates are placed in a different order. In Fig. 7.5(c), the predicates are placed at three levels in the order of p_2, p_3, p_1, resulting the average depths being 2.4. One effective ordering is placing predicates which are equal to the disjunction of a subset of fewer number of atomic predicates at lower levels. For example, in Fig. 7.5(c), p_1 is placed at the lowest level since it is equal to a_2.

Let $R(p)$ denote the subset of atomic predicates whose disjunction is p. $|R(p)|$ denotes the cardinality of $R(p)$. For example, since $p_1 = a_2$, $R(p_1) = \{a_2\}$, $|R(p_1)| = 1$. For $p_2 = a_3 \vee a_4$, we have $R(p_2) = \{a_3, a_4\}$, $|R(p_2)| = 2$. To reduce average leaf depth, the AP Tree is constructed by placing all predicates onto the tree in descending order of $|R(p_i)|$, where $|R(p_i)|$ is counted for each predicate p_i.

To reduce the average depth of leaves, once we obtained the atomic predicates set, we compute the subset of atomic predicates $|R(p_i)|$ for each predicate p_i. Then use Spark to sort the predicates by $|R(p_i)|$ and construct the AP tree.

7.3.4 Computing Packet Reachability

To compute the packet reachability, the network information, switches information, and the atomic predicates are needed.

Thanks to the packet behavior expression, for any forwarding predicate p_i, we can easily check whether the predicate evaluates to $true$ or $false$ for the packet. Recall that p_i represents a set of forwarding rules of an output port. Hence we can determine at any switch whether the packet is dropped and which port it is forwarded to. Starting from the ingress switch, we find the output port to which the packet is forwarded and then determines the next-hop switch, until the packet stops. If the packet is a multicast packet, it may be forwarded to multiple ports. We continue to find the forwarding ports on the next-visited switches until the packet reaches the destination or is dropped. The packet paths are thus obtained.

Fig. 7.6(a)shows an example for computing reachability for packets specified by atomic predicate a_4. Consider a packet which arrives at the switch s_1 and it is classified to atomic predicate a_4 by searching the AP Tree. The representation, $\neg p_1 \wedge p_2 \wedge p_3$, of a_4

shows that the packet is forwarded to s_2 because p_1 is false and p_2 is true for the packet. Similarly at s_2, the packet is forwarded to h_2 because p_3 is true for the packet.

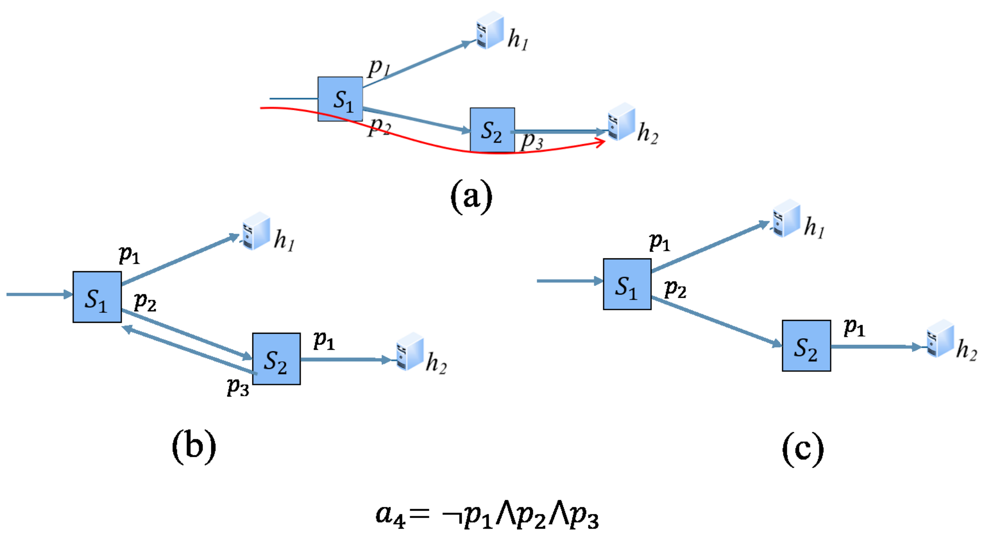

Figure 7.6: Computing reachability for packets specified by a_4.(a) Normal path. (b) Loop. (c) Black hole.

Loop Detection: Loop detection is performed while computing the reachability. If the next-hop switch has been visited before in the search we conduct there is a loop. For example, Fig. 7.6(b) shows a loop between s_1 and s_2 for packet specified by a_4.

Black Hole Detection: If the packet behavior cannot be determined at a none egress switch, which still has output link to other switches, we conduct there is a black hole in this switch. For example, Fig. 7.6(c) a black hole at s_2 for packet specified by a_4.

7.3.5 Updating

An important requirement of network verification is to support dynamic network changes, including link and rule changes, both of which require addition and deletion of predicates. We use the method presented in [10] to convert a rule insertion or deletion to predicate change. If there is no predicate change after a rule update, AP Tree does not need to be updated. Otherwise, the old predicate will be removed, and the updated predicate will be added into the AP Tree. These methods are also used after addition/deletion of a network link which requires addition/deletion of predicates.

Add a Predicate: When a new predicate p is added, for each leaf node representing an atomic predicate a in the current AP Tree, we compute $a \wedge p$ and $a \wedge \neg p$. If none of them is false, two children are added to the leaf node, representing $a \wedge p$ and $a \wedge \neg p$ respectively. If one and only one of the two conjunctions is false, the label of the leaf node is replaced by the other conjunction. If both conjunctions are false, this leaf node will be unchanged.

Delete a Predicate: To delete an existing predicate p from the AP Tree, we do not remove all internal nodes labeled by p. This is because after the removal of a node, merging the two sub-trees rooted at its children is very difficult. Instead, we still keep p in the AP Tree, but mark it as deleted in the list of all predicates. A query packet is still processed by the AP Tree to find its leaf node representing its atomic predicate. It is still evaluated by the deleted predicates to determine which sub-tree to visit next. However, when computing packet behaviors, all deleted predicates will be ignored.

7.4 Implementation and Performance Evaluation

We implement our proposed approach and run experiment based on Ubuntu Linux operation system version 18.04, the system is hosted on an Intel Core i5-6500 3.20GHz CPU with 7.7GB RAM.

We run Spark in local mode. Spark has an advanced DAG (Directed Acyclic Graph) execution engine that supports acyclic data flow and in-memory computing. It can run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk. In local mode, Spark spawns all the execution components - driver, executor, and master in the same single JVM. The default parallelism is the number of threads as specified in the master URL. We run Spark locally with as many worker threads as logical cores on our machine, in our experiment, 4 threads. This is the only mode where a driver is used for execution.

Table 7.1: Experimental Datasets Information.

	No. of hosts	**No. of rules**
Dataset 1	4	20
Dataset 2	8	88
Dataset 3	16	368
Dataset 4	32	752

We generate four datasets to evaluate the performance. Those datasets share the same topology, shown in Fig. 7.7, two switch connected to each other and serval hosts. We use a network simulator Mininet [43] and an open source OpenFlow controller Floodlight [13] to generate the data. Mininet is a fast, lightweight but powerful network emulator. Mininet is capable to create a realistic virtual network running functional hosts and switches. Mininet also provides the capability of interacting with user's custom OpenFlow controller. Floodlight high-performance, easy to use, open source OpenFlow controller. Floodlight comes with a web based and Java based GUI and most of its functionality is able to access through a REST API. Floodlight is able to automatically find the shortest path for table-miss packets, adds the appropriate flow entries into related switches, and lets the switch continue its forwarding.

The detailed statistics are listed in Table 7.1. In all of our experiments, we use our reachability algorithms to compute reachability of each pair of ports and measure the average time overhead of building the forwarding graph and computing reachability of on pair of ports.

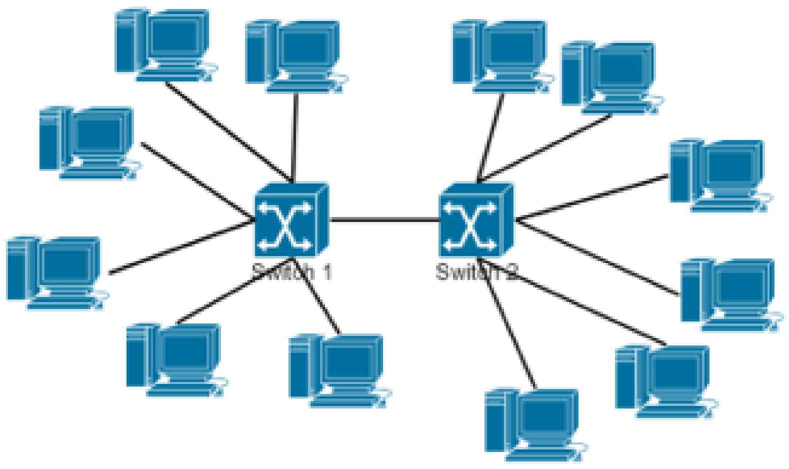

FIGURE 7.7: A topology example of the datasets.

For efficiency comparison, we choose our previous research denoted by *APSpark* [44]. APSpark is a similar method designed to verify firewall bypass threat in SDN networks. We choose it since it shares the similar proposes with the proposed method which are

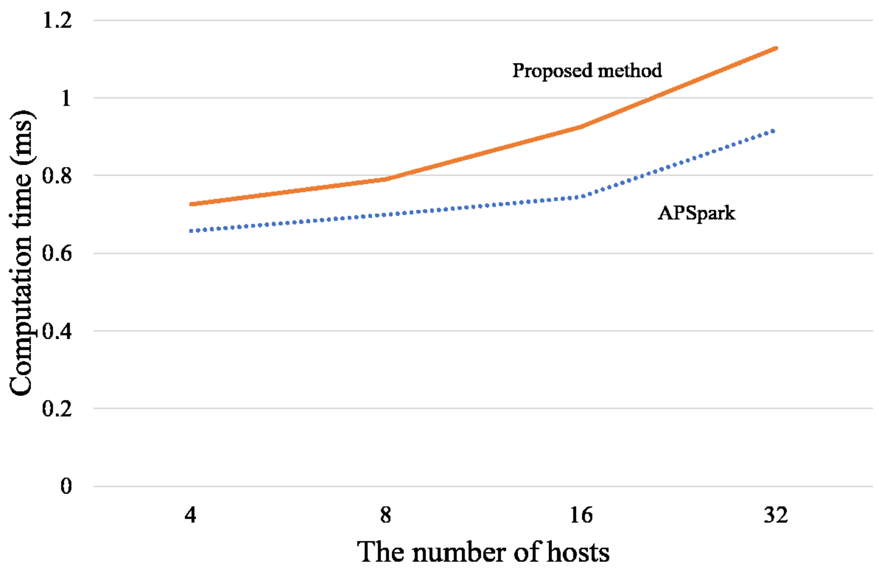

FIGURE 7.8: Computation time (ms) of initialization.

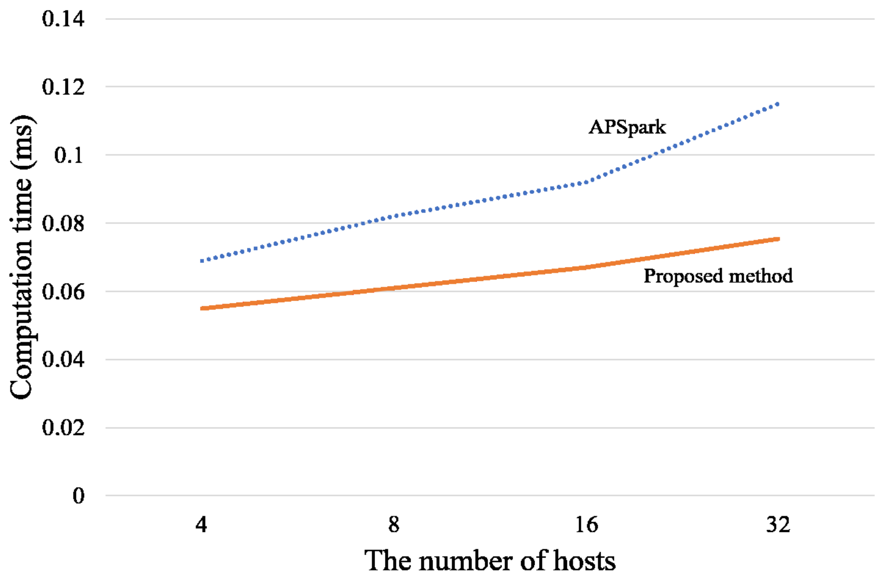

FIGURE 7.9: Computation time (ms) of reachablity computation.

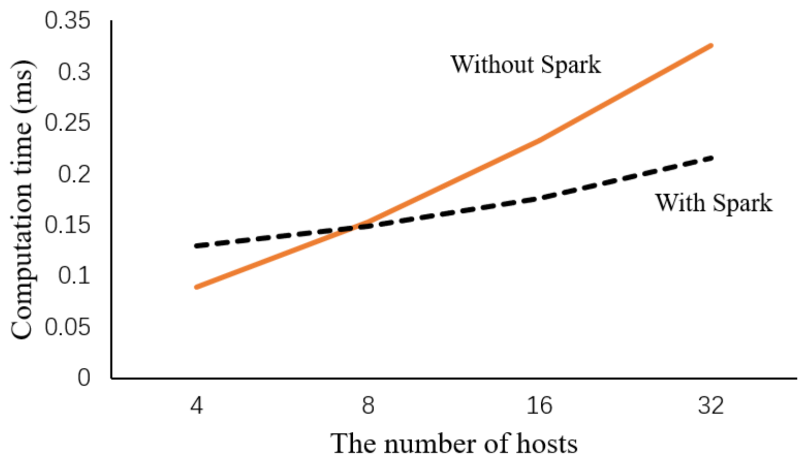

FIGURE 7.10: Computation time (ms) of computing atomic predicates.

packet reachablity verification. It uses the similar network model and combines AP and Spark to achieve fast verification, but it doesn't adopt AP tree to organize the AP data. The results from this experiment are shown in Fig. 7.8 and Fig. 7.9. It can be seen that the proposed method is faster than APSpark method in the reachablity computation, especially when dataset scales up thanks to AP tree structure, but requires more time to initialize. Note that the initialization only need to be done at the first time when the method is deployed.

We also compare the computation time of preprocessing rules and computing atomic predicates with or without Spark, which is the basic AP tree method. The results are shown in Fig. 7.10. Due to Spark's own system overhand, it does not outperform the original computation with small dataset, but it is faster while dealing with bigger dataset. And note that, we run Spark in local model. Within nowadays data center, where likely has Spark deployed in clusters due to its popularity, the proposed method is able to achieve faster computation and better scalability.

7.5 Conclusion

Network errors caused by configuration of flow tables are very common in nowadays network. Packet reachability verification tool is very valuable to detect those errors. In this paper, we have addressed the challenge of designing a fast packet reachability verification tool for large scale SDN networks. We have proposed a novel detection method based on modeling the network to a directed graph. The problem is formulated to the graph computation problem. We have adopted the concept of atomic predicates and AP tree so that we can verify reachability efficiently. We also adopt the parallel process computational framework Spark to accelerate the computation and achieve scalability. Our experimental results have demonstrated our proposed method is faster for computing reachability than our previous method.

Chapter 8

Conclusion

8.1 Conclusion

Configuration errors of flow tables are very common in nowadays network. In this study, we address the challenge of designing a fast network property verification tool for large scale SDN network. We propose three detection methods based on modeling the network to a directed graph. The problem is formulated to the reachability computation problem. We adopt the concept of atomic predicates, the AP Tree and the parallel process computational framework Spark so that we can detect several network properties efficiently and achieve scalability. Our experimental results demonstrate our proposed methods are effective and efficient.

Acknowledgements

First and foremost, I express my gratitude to my supervisors, Professor Jie Li and Shigetomo Kimura for the time they have spent on discussing and guiding my research during my master period. Their guidance is very precious not only for this thesis, but also for my future work.

I am grateful to my instructors, Professor Keisuke Kameyama, Yongbing Zhang, Hirotake Abe and Noboru Kunihiro for their advices, and feedback.

Last but not least, I am very thankful to my laboratory mates, my friends and my family, for their sincere support throughout my master study.

Bibliography

[1] Openflow switch specification 1.0.0. http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf.

[2] Ilora Maity, Ayan Mondal, Sudip Misra, and Chittaranjan Mandal. Tensor-based rule-space management system in sdn. *IEEE Systems Journal*, 2018.

[3] Peng Qin, Bin Dai, Benxiong Huang, and Guan Xu. Bandwidth-aware scheduling with sdn in hadoop: A new trend for big data. *IEEE Systems Journal*, 11(4): 2337–2344, 2015.

[4] Nick McKeown, 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):69–74, 2008.

[5] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, volume 14, pages 87–99. USENIX Association, 2014.

[6] David Oppenheimer, Archana Ganapathi, and David A Patterson. Why do internet services fail, and what can be done about it? In *USENIX symposium on internet technologies and systems*, volume 67. Seattle, WA, 2003.

[7] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: verifying network-wide invariants in real time. In *in Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27. USENIX Association, 2013.

[8] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 99–111. USENIX Association, 2013.

[9] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126. USENIX Association, 2012.

[10] Hongkun Yang and Simon S Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, 2016.

[11] Huazhe Wang, Chen Qian, Ye Yu, Hongkun Yang, Simon S Lam, Huazhe Wang, Chen Qian, Ye Yu, Hongkun Yang, and Simon S Lam. Practical network-wide packet behavior identification by ap classifier. *IEEE/ACM Transactions on Networking (TON)*, 25(5):2886–2899, 2017.

[12] James Mccauley. Pox: A python-based openflow controller, 2014.

[13] Project Floodlight. Floodlight openflow controller. Project Floodlight, 2013.

[14] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *2014 IEEE 15th International Symposium on*, pages 1–6. IEEE, 2014.

[15] SDN Ryu. Ryu sdn framework, 2016.

[16] TV Lakshman, T Nandagopal, R Ramjee, K Sabnani, and T Woo. The softrouter architecture. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Networking*, volume 2004. Citeseer, 2004.

[17] Lily Yang, Ram Dantu, T Anderson, and Ram Gopal. Forwarding and control element separation (forces) framework. *Internet Engineering Task Force*, 2004.

[18] Guru Parulkar. Open networking foundation,, 2011.

[19] Wolfgang Braun and Michael Menth. Software-defined networking using openflow: Protocols, applications and architectural design choices. *Future Internet*, 6(2):302–336, 2014.

[20] A Ghaffa and R Soomro. Big data analysis: Ap spark perspective. *Global Journal of Computer Science and Technology: Software & Data Engineering*, 15(1), 2015.

[21] Ning Zhang, Peng Yang, Ju Ren, Dajiang Chen, Li Yu, and Xuemin Shen. Synergy of big data and 5g wireless networks: opportunities, approaches, and challenges. *IEEE Wireless Communications*, 25(1):12–18, 2018.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[23] Weiwei Shi, Yongxin Zhu, Philip Yu, Jiawei Zhang, Tian Huang, Chang Wang, and Yufeng Chen. Effective prediction of missing data on apache spark over multivariable time series. *IEEE Transactions on Big Data*, (1):1–1, 2017.

[24] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. On controller performance in software-defined networks. In *Presented as part of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, 2012.

[25] Yi Liu, Cheng Lei, and Hongqi Zhang. Mr-verifier: Verifying open flow network properties based on mapreduce. In *2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 546–553. IEEE, 2015.

[26] Ehab Al-Shaer and Saeed Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, pages 37–44. ACM, 2010.

[27] Seuk Son, Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. Model checking invariant security properties in openflow. In *Communications (IC-C), 2013 IEEE International Conference on*, pages 1974–1979. IEEE, 2013.

[28] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review*, 41(4):290–301, 2011.

[29] Hongkun Yang, Simon S Lam, Hongkun Yang, and Simon S Lam. Scalable verification of networks with packet transformers using atomic predicates. *IEEE/ACM Transactions on Networking*, 25(5):2900–2915, 2017.

[30] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A security enforcement kernel for openflow networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 121–126. ACM, 2012.

[31] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008. ISSN 0146-4833. doi: 10.1145/1384609.1384625. URL http://doi.acm.org/10.1145/1384609.1384625.

[32] Juan Wang, Yong Wang, Hongxin Hu, Qingxin Sun, He Shi, and Longjie Zeng. Towards a security-enhanced firewall application for openflow networks. In *Cyberspace Safety and Security*, pages 92–103. Springer, 2013.

[33] Hongxin Hu, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. Flowguard: building robust firewalls for software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 97–102. ACM, 2014.

[34] Allan Liska. *Building an Intelligence-Led Security Program*. Syngress, 2014.

[35] Noureddine Boudriga. *Security of mobile communications*. CRC Press, 2009.

[36] Mininet Team. Mininet: An instant virtual network on your laptop (or other pc). *Google Scholar*, 2012.

[37] Ben Pfaff and Bruce Davie. The open vswitch database management protocol. 2013.

[38] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 117–130, 2015.

[39] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.

[40] Shin-ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *Proceedings of the 30th international Design Automation Conference*, pages 272–277. ACM, 1993.

[41] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47. IEEE Computer Society Press, 1993.

[42] Yicong Zhang, Jie Li, Lin Chen, Yusheng Ji, and Feilong Tang. A novel method against the firewall bypass threat in openflow networks. In *Wireless Communications and Signal Processing (WCSP), 2017 9th International Conference on*, pages 1–6. IEEE, 2017.

[43] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[44] Yicong Zhang, Jie Li, Shigetomo Kimura, Wei Zhao, and Sajal K Das. Atomic predicates based data plane properties verification in software defined networking using spark. *IEEE Journal on Selected Areas in Communications*, 2020.

List of Publications

1. **Yicong Zhang**, Jie Li, Shigetomo Kimura, Wei Zhao, Sajal K. Das: "Atomic Predicates Based Data Plane Properties Verification in Software Defined Networking Using Spark", *IEEE Journal on Selected Areas in Communications - Special Issue on Network Softwarization & Enablers*, 2020.

2. **Yicong Zhang**, Jie Li, Lin Chen, Yusheng Ji and Feilong Tang, "A Novel Method against the Firewall Bypass Threat in OpenFlow Networks," *Proceedings of 9th International Conference on Wireless Communications and Signal Processing (WCSP)*, pp. 1-6, October 11-13, 2017, Nanjing, China.