COMPUTER NETWORK VERIFICATION AND MANAGEMENT USING CONSTRAINT SOLVERS

SHUYUAN ZHANG

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
ELECTRICAL ENGINEERING
ADVISER: PROFESSOR SHARAD MALIK

APRIL 2016
Abstract

Todays computer networks have become extremely large and complicated. The increased scale is observed in datacenters, which can have hundreds of thousands of network devices. The increased complexity is due to multiple kinds of networking devices (routers, Network Address Translators, firewalls, etc.) that need to work together to execute diverse network functions such as routing, access control, and network address translation. In addition, networking devices need to support multiple protocols to make the network safer and faster. Consequently, it is non-trivial to manage such a large and complex system. Many research works have shown that network configuration is prone to human errors. Thus, it is critical to automate the network configuration and management tasks to achieve more reliable networks.

In my thesis, I mainly focus on three aspects of network management automation. (i) It is important that network administrators can utilize automatic verification and error detection mechanisms to reveal the underlying bugs and warn administrators whenever the network cannot satisfy certain critical properties, such as reachability and absence of forwarding loop. (ii) Firewalls have to be managed automatically in order to guarantee a correct packet access control policy. A simple error in firewall configuration can potentially leak confidential corporate information. (iii) Network changes and updates need to be managed correctly at a low cost. A network is a dynamic system and the state of a network may change frequently because of a new network policy enforcement, a bug-fix of existing network states, an installation of new network devices, etc. It is important to handle these different updates correctly to guarantee a smooth functioning of a network.

This thesis addresses these three problems by proposing a constraint solver based framework that can be applied to these problems and experimentally evaluates its scalability and feasibility. My study demonstrates that the constraint based solution
is capable of handling the verification problem, the firewall management problem, and the update problem for medium sized networks in reasonable time.
Acknowledgements

First and foremost, I am thankful that I have such a great advisor, Professor Sharad Malik. His dedication to research, his pursuit of perfection, his genuine kindness and patience for people has inspired me to grow and become a better person and researcher. Now that I will graduate from Sharad’s group, the values I learned from Sharad will benefit me for the rest of my life.

I am also grateful for all the insightful discussion with my committee members: Aarti Gupta, Sanjai Narain, Jennifer Rexford, David Walker, and Prateek Mittal. I would like to thank Aarti for her mentoring at NEC Laboratory where I was an intern there. I am very impressed by her rigorous thinking and her passion for research. I especially enjoyed brainstorming with her, which has been tremendous help for sharpening my work. I owe a great part of my achievements to Sanjai. Our discussion sparked a lot of research ideas. His help and support allowed me to develop these ideas into interesting projects. I am thankful that I had the opportunities to collaborate with two world-class researchers, Jen and David. They helped me deepen my understanding of the field. Jen also gave numerous thoughtful comments for this thesis. I am honored to have had the opportunity to teach for Prateek, from whom I learned a lot about Information Security. I also appreciate a lot of his comments for my pre-FPO presentation, which helped refine this thesis.

I am incredibly lucky to receive great mentorships from various people throughout my PhD years. Dr. Franjo Ivancic guided me through my first internship when I was at NEC Lab. Alvaro Drukaroff was my very helpful mentor at Google New York and I learned a lot about coding. I would like to thank other members of Sharads research group: Georg Weissenbacher, Nestan Tsiskaridze, Sayak Ray, Yakir Vizel, Daniel Schwartz-Narbonne, Carven Chan, Arnab Sinha, Divjyot Sethi, Yavuz Yetim, Sunha Ahn, Charlie Shucheng Zhu, Burcin Cakir, Pramod Subramanyan, Ameneh Golnari, Zhixing Xu, and Qi Nie. Their suggestions throughout the years had been
extremely helpful for refining my research ideas, improving my presentation skills, and broadening my horizons about various other topics.

Friends are a great treasure to me and I am very glad to have made many friends at Princeton. Having coffee alone is just not the same as having coffee with friends. I cannot list all the people I met but here are some that I played a lot of card games/board games/foozball/XBox with: Xianmin Chen, Wenyu Zhou, Xiao Chen, Xingchen Wang, Ran Zhang, Xuefeng Peng, Mingxuan Wu, Liechao Huang, Jin Cheng, Yifei Yuan, Yu Lin, and Junde Huang. I would also like to thank Jing Ma, my good friend from college for always being there for me.

I would also like to take this opportunity to thank my parents, to whom I owe so much more than words can describe. I know we will stay strong and never give up!

In the end, I would like to thank my best friend, my wife, Ren. We met here at Princeton and I cannot imagine getting through PhD and the rest of my life without her being by my side. She has and always will make the journey full of delightful joys.
To my parents and my wife, Ren.
## Contents

Abstract ........................................................................................................ iii
Acknowledgements ......................................................................................... v
List of Tables .................................................................................................. xi
List of Figures .................................................................................................. xii

1 Introduction ................................................................................................ 1
  1.1 Background .............................................................................................. 1
    1.1.1 Modern Computer Networks .............................................................. 1
    1.1.2 Software-Defined Networks ............................................................... 3
  1.2 Motivation for Network Management Automation .................................... 5
    1.2.1 Data Plane Management vs. Control Plane Management .............. 7
  1.3 Thesis Contributions ............................................................................... 8
  1.4 Thesis Organization ............................................................................... 9

2 Constraint Based Network Modeling .......................................................... 11
  2.1 Constraint Based Modeling ..................................................................... 11
    2.1.1 Boolean Satisfiability ....................................................................... 12
    2.1.2 Integer Linear Programming ............................................................ 14
  2.2 Boolean Model for the Data Plane ......................................................... 15
    2.2.1 Data Plane Abstraction .................................................................... 15
    2.2.2 Overview of the Boolean Model ..................................................... 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.3</td>
<td>Comparison with Other Approaches</td>
<td>21</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Coupling of Temporal and Spatial Information</td>
<td>22</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Solutions to the Temporal Limitations</td>
<td>23</td>
</tr>
<tr>
<td>2.3</td>
<td>Network Modeling</td>
<td>25</td>
</tr>
<tr>
<td>2.4</td>
<td>Switch Modeling</td>
<td>27</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Input Aggregation Module</td>
<td>28</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Matching Table</td>
<td>29</td>
</tr>
<tr>
<td>2.5</td>
<td>Chapter Summary</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>Data Plane Verification</td>
<td>33</td>
</tr>
<tr>
<td>3.1</td>
<td>Network Properties</td>
<td>33</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Forwarding Loop</td>
<td>34</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Reachability Checking</td>
<td>38</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Slice Isolation</td>
<td>40</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Network Equivalence Checking</td>
<td>42</td>
</tr>
<tr>
<td>3.2</td>
<td>Experimental Results</td>
<td>43</td>
</tr>
<tr>
<td>3.3</td>
<td>Conclusion</td>
<td>46</td>
</tr>
<tr>
<td>4</td>
<td>Firewall Management</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>Single Firewall Management</td>
<td>48</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Background</td>
<td>49</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Firewall Equivalence, Inclusion, and Redundancy Checking</td>
<td>51</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Firewall Synthesis</td>
<td>54</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Experiments</td>
<td>56</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Related Work</td>
<td>62</td>
</tr>
<tr>
<td>4.1.6</td>
<td>Conclusions and Future Work</td>
<td>63</td>
</tr>
<tr>
<td>4.2</td>
<td>Distributed Firewall Management</td>
<td>63</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Big Switch Abstraction</td>
<td>65</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Using 0/1 ILP to express propositional logic . . . . . . . . . . . . . . 14
2.2 Notation for the model . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.3 Switch Encoding Variables . . . . . . . . . . . . . . . . . . . . . . 28
4.1 Redundancy Removal Results . . . . . . . . . . . . . . . . . . . . 61
4.2 Problem Formulation: Notation . . . . . . . . . . . . . . . . . . . 68
4.3 Capacity vs. Overhead in Rule Merging . . . . . . . . . . . . . . . 86
List of Figures

1.1 Software-Defined Networks ........................................... 3
2.1 Matching Table .......................................................... 17
2.2 Viewing a Network as a Logic Circuit .............................. 19
2.3 A packet flows through a network ................................... 20
2.4 Multiple packets arrive at the same switch ....................... 22
2.5 Encoding Switch Connectivity: An Illustrative Example ........ 27
2.6 The Encoding Blocks for Switch i ................................. 28
2.7 Matching Table .......................................................... 29
3.1 Weaker version of forwarding loop ................................. 34
3.2 Unreachable forwarding loop ....................................... 35
3.3 Unreachable Loop ...................................................... 37
3.4 Equivalent Networks .................................................. 42
3.5 L = 4 N = 20 ........................................................... 45
3.6 L = 8 N = 80 ........................................................... 45
3.7 L = 10 N = 125 ....................................................... 45
3.8 L = 12 N = 180 ....................................................... 45
4.1 The Overview of Firewall Encoding ................................. 51
4.2 The Encoding for Symbolic Matching Tables ..................... 54
4.3 Mutation 1: Flipping a random bit ................................. 59
4.4 Mutation 2: Deleting one random rule. ........................................ 59
4.5 Mutation 3: Swapping two random rules .................................... 60
4.6 Inclusion Checking ................................................................. 61
4.7 Big Switch Abstraction: Illustrative Sketch .................................. 66
4.8 Problem Illustration ............................................................... 70
4.9 Flow Chart for Our Approach ................................................. 72
4.10 Circular Dependency for Mergeable Rules ................................... 77
4.11 Sub-policy for each route ....................................................... 78
4.12 Execution time vs. # of Rules. k = 8 ......................................... 83
4.13 Execution time vs. # of Rules. k = 16 ....................................... 83
4.14 Execution time vs. # of Rules. k = 32 ....................................... 84
4.15 Execution time vs # Paths ....................................................... 88
4.16 Execution time vs #rule entries per switch ................................. 88
5.1 Example: Routing State 1 .......................................................... 94
5.2 Example: Routing State 2 .......................................................... 94
5.3 Encoding for Switches ............................................................. 99
5.4 An Illustrating Example ........................................................... 100
5.5 # Switches vs. Time ............................................................... 103
5.6 # Switches vs. # Indices .......................................................... 104
5.7 Percent of Updated Switches vs. Time ....................................... 104
5.8 Percent of Updated Switches vs. # Indices .................................. 104
Chapter 1

Introduction

As modern computer networks become exceedingly large and complex, network administrators find the job of managing networks becomes increasingly challenging and error prone. Misconfiguration of networks can not only cause network outages or network downtime, but also lead to security breaches. This thesis addresses automation of network management.

1.1 Background

1.1.1 Modern Computer Networks

With the advent of 21st century, the development of Internet and computer networks in general has also entered a new era. The size and complexity of modern networks have become significantly different from the networks when they were first invented. This difference can be seen in several aspects:

- Thanks to the invention of new Internet services, such as cloud computing, Content Distribution Networks (CDN), and the foundation of new Internet-based corporations, such as Google and Facebook, the size of a single networking system becomes unprecedentedly large and continues to increase. For example,
a data center network we need 2,000 switches to support 16,000 servers using Fat-Tree topology [7] and large data centers can easily go beyond hundreds of thousands of servers and tens of thousands of switches [34, 61]. As the entire IT industry is growing, it is likely that this number will increase in the future.

- One characteristic of modern networks is the large number of networking services and functionalities that the network devices need to support. With the growth of cloud computing and data centers, functionalities such as network virtualization, load balancing, bandwidth capping, etc. have become an essential part of data center networks. Moreover, network security is always critical to the networks and hence, firewalls, intrusion detection systems, and other security devices also play an important role in modern networks. Other services and protocols to improve the performance or functionalities of the networks such as Virtual Local Area Network (VLAN), Multiprotocol Label Switching (MPLS), Network Address Translation (NAT) also contribute to the complexity of today’s networks.

- Further, modern networks may suffer from different implementations of networking protocols across vendors. Even though standard organizations such as Internet Engineering Task Force (IETF) develop and promote a great number of networking protocol standards, specific details that are not formally specified in the standard itself are subject to implementation differences [39]. Moreover, the implementation of vendors may contain bugs and result in nonconformance to the original specification. These different implementation discrepancies can lead to potential inefficiencies of network operation.
1.1.2 Software-Defined Networks

One of the major problems of traditional networks is the lack of flexibility and layered abstraction. Most protocols are pre-installed in the routers or switches and they cannot be easily changed or customized. The rigidity of the network products significantly increases the product cycle of new network offerings. Moreover, the entire methodology of solving network problems is quite ad hoc. Whenever a new network function or service is needed, network researchers explore new protocols to handle the problem but this further adds more complexity to the system. As the network evolves, the number of networking protocols can become increasingly large, which in turn makes network management exceedingly difficult.

Recently, the networking community has put significant effort in promoting Software-Defined Networks (SDN) with the hope that this new network paradigm can reduce the network product cycle and stimulate rapid development of new network technologies. The most representative system of SDN is the OpenFlow platform [13]. The essential idea of SDN is to replace all networking devices with programmable “switches” (or OpenFlow switch for OpenFlow based solution) and use a centralized controller to control the complete behavior of the network as shown in Figure 1.1. The programmable switches are programmed using the control packets...
sent from the controller through a dedicated channel. Their program is a prioritized matching table or flow table that specifies which action to perform on the incoming packets based on their header. The network controller runs the network operating system and network applications written in high-level programming languages to manage the matching table in each switch. Now, the role of switches in networking becomes more like the role of the CPU in a PC. In this setting, on top of generic networking hardware is the networking operating system, on top of which is the network applications or programs.

SDN provides a clear layered abstraction as discussed above and there are many great advantages because of that. The application layer is mainly responsible for the functionalities related to a single network. Whenever a new networking function is needed, programmers can easily write a network application to implement the functionality, which greatly reduces the development-to-product cycle and enables network innovation. The network operating system is responsible for coordinating different applications, allocating network resources, and collecting network information. The network hypervisor is responsible for network isolation. For example, it handles network virtualization. As we can see, every layer is clearly defined and separated. Whenever there is a new need in one layer, we do not need to change much on the others.

Another advantage of SDN is the centralized view of the network. Because of the centralized controller, administrators can have a consistent, timely, and comprehensive view of the entire network state. This centralized view can describe the complete behavior of the network at a single point of time. Thus, it can provide a fast exposure and response of network errors such as configuration bugs and hardware errors.
1.2 Motivation for Network Management Automation

As mentioned in the previous sections, modern networks have become extremely large and complicated and hence, it is no longer a trivial job to correctly manage those network devices. Even though SDN has shown many advantages over traditional networks, it is still not the ultimate perfect solution to solve all problems because there are still human beings involved in the management process. Research has shown that a great deal of network downtime is due to human misconfiguration [29]. For example, lack of accurate documentation, such as ad hoc network changes without formal records, can greatly increase the chance of misconfiguration. SDN will not mitigate such problem because it still requires humans to write and configure applications. Therefore, there is a long way to go to achieve fully automated management for both SDN and traditional networks. The challenges that need to be addressed are as follows:

- First, SDN applications, which are just like any other software, can have bugs as well. Debugging networking systems is even more difficult than debugging normal software because a network is an extremely distributed system and massive events are happening concurrently. Different event interleavings make the problem very large. Further, every programmable switch may have thousands of matching rules in the matching table. If there is a bug, the potential search space is huge. Therefore, traditional testing techniques, such as using ping or traceroute, to solve this kind of problem is difficult to cover the entire search space if not impossible. This type of problem requires advanced techniques that can explore the entire search space in one shot, such as verification.

- Secondly, although network applications relieve some of the burden on administrators, they still have to configure and even customize those applications
for certain types of networking problems. In order to achieve fully automated network management, we argue that we should only require administrators to specify the high-level behavior of the network without even requiring them to write or to choose network applications and this can be difficult. In many cases, it may be very easy to automatically generate a naive solution for certain management problems but those solutions may sacrifice efficiency and performance. It is quite challenging to solve the problems optimally. For example, given a high-level firewall specification, it is very difficult to generate an optimal firewall implementation with the least number of firewall rules installed. This may be needed to meet the capacity constraints of hardware look-up tables. Solving this problem is quite difficult because the packet header space the firewall matches on is huge. A typical firewall matches on source/destination IP addresses, source/destination port, and protocol, which in total are 104 bits. If we allow arbitrary wildcard in the matching rule, every bit in a rule can have 3 cases (0, 1, and wildcard), which leads to the total number of possible rules to be $3^{104} \approx 4 \times 10^{49}$. Searching within such a humongous space is an extremely difficult problem but if we are smart about how we model the problem, we can transform firewall optimization problem into Quantified Boolean Formula problem, which can significantly reduce the problem size.

- Thirdly, when the controller finishes converting the high-level policy to the low-level flow table details, it should send those flow table rules to the remote switches and it is also very challenging to apply network changes cheaply and efficiently. Naive solutions can easily lead to unreachable networks because the update itself can change the routing behavior of the network, which can potentially result into forwarding loops. For traditional networks there has been much research on how to apply network updates for network migrations but they all have limitations such as incapability of handling firewalls. Other
solutions are costly. Currently, SDN relies on dedicated networks such as VLAN to update network changes. It is desirable that we do not rely on VLAN because we can save the resource used to run the spanning tree algorithm and we can use the VLAN tag space for other applications.

Therefore, automating the network management will become increasingly necessary as the size and complexity of the network grows. Further, there are many technical challenges to be addressed to make the network fully automated.

1.2.1 Data Plane Management vs. Control Plane Management

There is a common ground shared by most management tasks, in that they all have to go through the data plane. In networking, the data plane decides the action of all the networking devices on the packets they receive and it usually refers to the matching tables or matching tables of all the network devices. For example, it may include the forwarding tables of routers, MAC-learning tables of switches, Access Control List of firewalls, Network Address Translation table of NAT devices, the matching table of SDN programmable switches, and so on. In contrast, the control plane of a network computes the information in the data plane so the control plane refers to the underlying mechanism (different network algorithms, high-level languages, and so on) that generates the specific matching tables.

Because the data plane completely controls the networking device’s behavior, the data plane defines the complete network behavior at the current point of time and hence, it provides enough information required by many network management tasks. Even though the data plane is computed by the control plane, the data plane has a big advantage over control plane in terms of easiness of solving. The control plane of a network may refer to high-level languages but the data plane is essentially a
collection of Boolean logic based matching tables. Dealing with high-level languages is much more challenging than dealing with matching tables.

Because of the potential performance and scalability benefits, this thesis mainly focuses on the verification and management of data plane.

1.3 Thesis Contributions

My thesis addresses the automation of data plane network management in the following ways. First, we propose a framework that is general enough to handle a variety of management tasks with acceptable computation time. This framework is based on constraint based modeling, which can be either instances of Boolean Satisfiability (SAT) [30], Satisfiability Modulo Theory (SMT) [49], or Integer Linear Programming (ILP) [59] depending on the specific requirement of the problem. The framework models the data plane of a network and captures the complete behavior of the current network state. Based on the data plane state, the framework is powerful enough to answer a variety of data plane questions such as whether there are forwarding loops in the network, whether the network policy implementation is consistent with the desired network behavior, etc.

Secondly, based on the constraint based model, we propose a solution to verify data plane state against several network properties, including reachability checking, forwarding loop absence, slice isolation, etc. We outline a solution to check whether two networks behave exactly the same. This work is valuable to network management to avoid misconfigurations and minimize bugs.

Thirdly, we formulate a variety of firewall management problems using the constraint-based model. A firewall is a critical network security device and is essential for the prevention of malicious traffic. Therefore, correct firewall management
has to be guaranteed. In this set of results, we formally define and solve several firewall management problems as follows:

- The first problem is firewall relationship checking. Given two firewalls, our formulation can check whether the two behave exactly the same for every packet. Based on the firewall equivalence checking, we can remove redundancies for firewalls. Further, we also formulate how to synthesize an optimal firewall based on solving Quantified Boolean Formula (QBF) solving.

- The second problem is how to implement a distributed firewall given its high-level behavior. Our technique enables an easy firewall deployment for network administrators because they only need to specify the high-level behavior of the network.

- The third problem we solve is to check whether the implemented firewall is consistent with the given high-level reachability specification. The solution can verify whether an implemented firewall satisfies all the reachability properties that are specified by the network manager.

Fourth, we propose how to update the data plane state using in-band networks. The central control unit or controller sends control packets to the devices in order to validate the update. However, it is not easy to do this correctly and efficiently. A naive update order can make part of the network unreachable. My solution is based on the constraint based formulation without any assumption on the network topology and network state and it does not consume any of the packet header space resource.

### 1.4 Thesis Organization

The thesis is organized as follows: Chapter 2 explains the details of the constraint-based modeling and discusses the model’s advantages and disadvantages. Chapter 3
discusses how to use the constraint based model to formulate the data plane verification problem to verify network properties such as absence of forwarding loops and reachability checking. Chapter 4 addresses many firewall management problems, such as firewall optimization, firewall policy distribution, and firewall consistency checking. Next, Chapter 5 illustrates how to update network changes efficiently using in-band networks. Finally, Chapter 6 draws some overall conclusions and identifies future work in this direction.
Chapter 2

Constraint Based Network Modeling

This chapter discusses the constraint based modeling that we use to solve various network problems throughout my thesis. First, we will briefly introduce the constraint based problem formulation including Boolean Satisfiability (SAT) and Integer Linear Programming (ILP) which my thesis is mainly based on. Then, we will give an overview of the constraint-based network, which will be followed by specific details about the network modeling.

2.1 Constraint Based Modeling

In constraint based modeling, the problem of interest is modeled as a set of constraints in some theory. A solution to the problem corresponds to a solution for these constraints. For example, SAT is based on a set of propositional logic constraints on Boolean variables. ILP is based on a set of linear arithmetic constraints with all variables being integers and may also have an objective function for certain optimizations. The solvers for those models can solve these constraint problems by providing
a satisfiable assignment for each variable or proving the model is unsatisfiable. Their solution provides a solution to the application problem of interest.

2.1.1 Boolean Satisfiability

All variables in SAT are Boolean and all constraints are in propositional logic \[30\]. The syntax of propositional logic is defined as follows:

\[
\text{formula} : \text{formula} \land \text{formula} | \text{formula} \lor \text{formula} | \neg \text{formula} | (\text{formula}) | \text{atom}
\]

\[
\text{atom} : \text{Boolean-variable} | \text{TRUE} | \text{FALSE}
\]

Other operators such as EQUALITY (\(\equiv\) and \(\leftrightarrow\) are used interchangeably) and IMPLY (\(\rightarrow\)) can be constructed using OR (\(\lor\)), AND (\(\land\)), and NOT (\(\neg\)). A literal is either an atom or the negation of an atom.

As a short example, the following formula describes a simple SAT problem instance.

\[(a \lor b) \land (\neg b \lor c) \land (\neg b \lor \neg d) \land d\]

Because the last clause is \(d\), hence, \(d\) has to be set to true for the formula to be satisfiable. In order to satisfy \((\neg b \lor \neg d)\), \(b\) has to be set to false if \(d\) is true. Then, because of \((a \lor b)\), \(a\) has to be true. Because \((\neg b \lor c)\) has already been satisfied by \(b\) being false, \(c\) can be either true or false. Therefore, a satisfiable solution can be either \(a = \text{TRUE}, b = \text{FALSE}, c = \text{TRUE}, d = \text{TRUE}\) or \(a = \text{TRUE}, b = \text{FALSE}, c = \text{FALSE}, d = \text{TRUE}\). This previous example shows a satisfiable formula and the following example is an unsatisfiable case.

\[(a \lor b) \land (\neg b \lor c) \land (\neg b \lor \neg d) \land d \land (\neg a)\]
This formula is similar to the previous example except that it includes an additional clause \( \neg a \). Based on the previous example, \( a \) has to be set to true but the new clause conflicts with our finding. Therefore, this formula can no longer be satisfied.

SAT formulas can be expressed in several forms and the most popular one is the *Conjunctive Normal Form (CNF)* and many SAT solvers only support CNF formulas. A CNF formula is a conjunction of clauses where a clause is a disjunction of literals. The short example described above is in CNF form because it is a conjunction of a set of disjunctions of literals. It is also possible to convert any set of Boolean logic expressions into CNF formula using the *Tseitin Transformation* [21]. For example, in order to express \( a \oplus b \) or \((a \land \neg b) \lor (\neg a \land b)\) in CNF, we create a new variable \( c \) to represent it using the following formula:

\[
(\neg c \lor a \lor b) \land (\neg c \lor \neg a \lor \neg b) \land (c \lor \neg a \lor b) \land (c \lor a \lor \neg b)
\]

Each clause in the above formula corresponds to one row in the truth table of the formula. Iff \( a \lor b \) is false or \( \neg a \lor \neg b \) is false, \( c \) is false. Iff \( \neg a \lor b \) is false or \( a \lor \neg b \) is false, \( c \) is true. Multi-variable AND or OR operations can be achieved by concatenating several Tseitin Transformations. For example, in order to use \( x \) to represent \( a \lor b \lor c \), we can first use \( x_1 \) to represent \( a \lor b \), use \( x_2 \) to represent \( x_1 \lor c \). Then, \( x \equiv x_2 \).

In order to represent integer variables, we can use bit vectors to replace an integer variable with a fixed number of Boolean variables, just like how we use 32 binary bits to represent an integer in programming. Even though this implicitly adds a range constraint on the represented integer, we can always increase the number of bits to increase the range. Moreover, optimization using SAT can be achieved by using binary search on the upper/lower bound of the objective function as part of the constraint satisfaction formulation.
2.1.2 Integer Linear Programming

ILP is more powerful than SAT because its variables can be integers and linear constraints are more expressive than propositional logic. When the variables are either 0 or 1, ILP (called 0/1 ILP) can be used to express propositional logic. Table 2.1 lists the equivalent expressions for SAT and ILP. All variables in the table are binary.

<table>
<thead>
<tr>
<th>SAT Formula</th>
<th>0/1 ILP Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c = TRUE )</td>
<td>( c = 1 )</td>
</tr>
<tr>
<td>( c = FALSE )</td>
<td>( c = 0 )</td>
</tr>
<tr>
<td>( c = \neg a )</td>
<td>( c = 1 - a )</td>
</tr>
<tr>
<td>( c = a \land b )</td>
<td>( c \leq a, c \leq b, c \geq a + b - 1 )</td>
</tr>
<tr>
<td>( c = a \lor b )</td>
<td>( c \geq a, c \geq b, c \leq a + b )</td>
</tr>
</tbody>
</table>

Table 2.1: Using 0/1 ILP to express propositional logic

In many network problems, we often express an order constraint. For example, if a condition is satisfied, the packet arrives at switch B before it arrives at switch C. If the condition is captured by variable \( a \) and the integer times for when the packet arrives at switch B and C are integer variables \( b \) and \( c \), the constraint becomes \( a \rightarrow (b < c) \). This expression is not propositional logic. SAT can express this using bit vectors to present \( b \) and \( c \), which is quite inefficient because it significantly increases the size of the formula. ILP can express this constraint using a small trick with a pre-condition that \( b \) and \( c \) are bounded. Assume the upper bound for the absolute value of \( |b| \) and \( |c| \) is a constant, \( U \). Since the logical implication \( A \rightarrow B \) is equivalent to \((\neg A \lor B)\), either \( \neg A \) or \( B \) has to be set to true. In ILP, we use two binary choice variables \( m_1, m_2 \) to encode this. We can use the following formula to express \( a \rightarrow (b < c) \) [12].

\[
a + (m_1 - 1) \leq 0 \quad (2.1)
\]

\[
b - c + 2U \times (m_2 - 1) < 0 \quad (2.2)
\]

\[
m_1 + m_2 \geq 1 \quad (2.3)
\]
$m_1$ and $m_2$ controls either $a \leq 0$ (equivalent to $\neg a$) or $b - c < 0$ should be satisfied. When either $m_1$ or $m_2$ is 0, the equation it controls will become trivially satisfied. When they become 1, the equations they control have to satisfy the original requirement. Equation 2.3 constrains that at least one of $m_1$ and $m_2$ should be 1, meaning at least one of $a \leq 0$ or $b - c < 0$ should be valid.

Another advantage of ILP is its capability of supporting an objective function, which makes optimization much easier than for SAT as you do not need to use binary search. Let us review our previous optimization example for SAT and assume we want to minimize $X$. In ILP, this can be easily achieved by setting an objective function $\text{minimize}(X)$. Minimizing the upperbound of a set of variables $\{X_i\}$, which is equivalent to having an objective function of $\text{minimize}(\text{max}(\{X_i\}))$, may not be directly supported by most ILP solvers. We can achieve this by adding an integer variable $\text{UpperBound}$ and a set of constraints $\forall X_j \in \{X_i\}: X_j < \text{UpperBound}$ and then setting an objective function $\text{minimize}(\text{UpperBound})$.

## 2.2 Boolean Model for the Data Plane

As discussed in Chapter 1, dealing with data plane is much easier than dealing with control plane. In this section, we will provide a high-level overview of how to model the data plane.

### 2.2.1 Data Plane Abstraction

The data plane can be cleanly extracted from the network. The operating systems of many networking devices support the commands to extract the matching tables from the devices. For example, Cisco’s iOS supports “show ip route” to dump the routing table. For SDN, the controller can easily keep track of the matching table in a switch since it is the central unit that initiates all the switch update.
In this work, the data plane refers to two things:

1. The physical connection of all the networking devices.

2. The snapshot of the matching tables of all networking devices in the network at a single instance of time.

The physical connection of the networking devices describes how switches are connected together and it defines the topology of the network. Without the topology information, we cannot reason about how packets flow through the networks.

The snapshot of all matching tables provides the information on how each switch will react to each packet it receives. In real life, obtaining a consistent data plane snapshot may be difficult. For example, in order to set up a route for a packet in SDN, the controller has to install rules for all the switches along the route but different update latencies can cause switches being updated at different time. If we take the snapshot when the switches are being updated, we only capture an intermediate state, which may not accurately reflect the state of the network. If we perform network debugging or verification on such intermediate states, we may raise false alarms. Fortunately, there has been some research to solve this consistent snapshot problem [62] and in this work, we assume all the data plane snapshots are consistent.

The matching table in the data plane takes the incoming packet as the input and forwards the packet to each egress port. The table is abstracted as a collection of prioritized matching rules as shown in Figure 2.1. Each matching rule is defined by its priority, its matching field, and its action field. When a matching table receives a packet, it matches the packet against the matching field of all the rules in the order of their priority. High priority rules will be matched before low priority rules. The first matched rule’s action will be applied to the packet and the switch will forward the packets or modify the packet header according to the action field of the matched rule. Now, we will explain the details of the matching field and the action field.
The matching field describes which packets match the rules. It is defined as a collection of packet header constraints. For example, in MAC address based routing, the matching field may be the MAC address of the packet. In IP address based routing, the matching field may constrain on the destination IP address of the packet. In SDN, the matching field can include several packet fields, including source/destination IP addresses, source/destination port numbers, protocol numbers, and many other fields.

The constraints on the header fields may be exact matching, wildcard matching, or range matching.

- **Exact Matching:** Exact matching specifies that the matching field should match exactly the packet. For example, a matching field may constrain that it only matches IP destination address 192.168.0.1.

- **Wildcard Matching:** Wildcard matching is used for matching a set of values and it is useful for prefix matching. In this thesis, we use “x” to represent a wildcard and it can match both “0” and “1”. For example, 192.168.0.1 is binary value: “11000000,10101000,00000000,00000001”. Then, 192.168.0.0/8
can be expressed as “11000000,xxxxxxxx,xxxxxxxx,xxxxxxxx”. In this work, we also support general wildcard matching, such as matching “01xx01xx”. This is beyond prefix matching where the wildcard string can only be in the suffix.

- **Range Matching**: Range matching can be used to match a range of values and this is useful for the port matching. For example, we can specify that the destination port, \( port \), of the packet must be within the range \([1000, 65535]\). This is equivalent to \( 1000 \leq port \leq 65535 \).

**Action Field**

The action field has two parts: one modification field to specify how the rule changes the packet header and one forwarding field to specify which egress ports the rule forwards the packet to.

- **Modification Field**: This field specifies how to transform the packet header. For example, NAT may dynamically change the source IP address and port value. MPLS may require packet encapsulation and decapsulation to add or remove labels. In this chapter, we specify the modification field to be a function that can apply any Boolean operations on the packet header.

- **Forwarding Field**: “Forwarding actions” specifies which port the matching table forwards the packet to. The set of actions considered in this thesis includes simple forwarding (forward to a specific egress port), broadcasting (forward to all egress ports), multicasting (forward to a subsets of egress ports), forwarding to incoming ports, and blocking (drop the packet).

Since the matching field abstraction is compatible with a wide range of networking devices, for simplicity, we use “switch” to represent all network devices, including routers, switches, firewalls, NAT boxes, OpenFlow programmable switches, etc.
2.2.2 Overview of the Boolean Model

Our Boolean model is based on the idea that the data plane abstracted in the previous subsection is similar to a hardware circuit. Since we only consider the data plane snapshot, the data plane considered here is static and stateless, which resembles a combinational circuit. Even though each port or link in a network is bi-directional, we can split one bi-directional link into two uni-directional links as shown in Figure 2.2.

In the figure, all the vertices are switches and all the edges are network links. Then, all the ingress ports become the input of a switch and all the egress ports become the output. Inside a switch is the look up table, which resembles comparison logic for lookup, multiplexing logic for routing, and mapping logic for header modification. Therefore, a switch can be modeled as a combinational circuit. Similarly, on a higher level, the network is composed of several switches and hence, can be modeled as a composition of several combinational circuits.

Since combinational circuits are based on propositional logic, we can use a single propositional logic formula to represent a network. We can use a Boolean variable to represent whether a packet is able to reach that link. For the example in Figure 2.2, we can use a binary variable $v_1$ to represent that there is a packet sent into the network’s ingress link of switch A and variable $v_2$ represents if the packet can reach the network.
Figure 2.3: A packet flows through a network

egress port at switch A. If two switches are connected by the same link, they can share the same variable for their corresponding ports. For example, we use variable $v_{12}$ to represent if the packet can reach the link from switch B to switch A. Then, a valid assignment of each link variable will represent how the packet traverses through the network. Figure 2.3 shows a valid assignment of each variable. In the figure, a grey link represents a variable being assigned to 0 while a black link is assigned to 1. We can see that the set of variables that are assigned to 1 ($v_3, v_{12}, v_9, v_{16}, v_6$) forms a directed link from network ingress port at switch B to network egress port at switch C. This directed link captures the packet flows through the network along the route B→A →D →C. How each switch decides which port to forward the packet to depends on the specific matching table inside each switch and the information of the packet header sent to the ingress links. The details about the model of each switch and packet header will be discussed in the following sections.

Based on the propositional network model, we can specify corresponding logic constraints on the model to perform certain network tasks. For different tasks, we only need to add different constraints while the underlying network model is the same.
How to use the model for different network problems will be discussed in the following chapters.

### 2.2.3 Comparison with Other Approaches

There are several related efforts in the area of formal network/data plane verification. We briefly review them here and place our work relative to them.

There is a set of efforts based on using finite state systems. As in our work, they consider the network state to be fixed, i.e., a snapshot of the dynamic network. However, they consider transitions in the packet state. The packet state is defined as \((h, p)\) where \(h\) is the packet header information and \(p\) is the packet location. The state transitions are determined by the network state, i.e., the rules in the switches. These efforts can further be partitioned into two groups - one based on model checking this finite state system \([9, 51]\) and one based on ternary symbolic simulation of this system \([28]\). The model checking efforts use CTL to model the network properties or management tasks and use standard model checkers to check them. In contrast, we avoid the state space analysis involved in model checking through innovative propositional logic modeling.

A key advantage of our approach over ternary symbolic simulation (the *Header Space Analysis* approach) is a uniform framework for checking properties. For example, in order to check forwarding loops, their approach needs to tag each packet with all the ports that it has visited. For large networks this overhead can be substantial. As we will discuss in the next section, our method couples temporal and spatial information together, we do not need to bookkeep this information. Overall, the processing of ternary vectors inherits the limitations of a Disjunctive Normal Form (DNF) representation including a possibly exponential growth of packets in going through a prioritized matching chain.\(^1\) While our method builds a single formula,

\(^1\)This is because each matching rule passes down to lower priority rules the set difference of the current set of packets in DNF with what is matched by this rule.
it avoids any size explosion through the standard technique of adding intermediate variables.

The effort that is closest to our work is the Anteater project that also encodes the property check as a Boolean formula. However, unlike our model, it does not build a single formula for the network and then check different properties for it. Rather, they build a custom formula for checking the reachability between two ports. They also show how a forwarding loop involving a specific switch $s$ can be checked by splitting this switch into two switches $s$ and $s'$ and the checking the reachability of $s'$ from $s$. Further, for checking loops for the complete network this check needs to be done for each switch separately.

2.2.4 Coupling of Temporal and Spatial Information

Although a network can be modeled as a combinational circuit, there is a major difference between a network model and a traditional combinational circuit model. The propositional logic model for a network can describe the entire lifetime of a packet but in hardware verification, one Boolean model usually represents a single clock unit. In network model, temporal information is based on the location information, i.e. the information about time and location is tightly coupled together. Because every link
is directed, we can infer how a packet flows through a network just based on the
direction of the adjacent links. This highlights a great advantage of our model: it does not need to store the history of locations a packet has traversed. We do not need to use extra storage and time to store and retrieve the packet history. Instead, we only use one set of variables to represent the packet at a location through its path through the network.

However, this also raises a tricky issue: what if multiple packets arrive at the same link at the same time or the same packet arrives at the link at different times, such as in the case of a forwarding loop? Since we only use one variable to encode a link, it can represent at most one packet arriving at the link at one time and the information about other packets or other time will get lost. A small example is shown in Figure 2.4. In the figure, the packet get multicasted at switch B and the two multicasted packets join at switch C. If switch C decides to forward both multicasted packets to the egress port, we have to choose which packet should be allowed to pass through the switch and the information about the other packet will be lost. What if multiple packets arrive at the same switch as shown in the Figure? We can only match one packet against the matching table at switch C.

2.2.5 Solutions to the Temporal Limitations

The solution to the above issue is fourfold:

- First, we only examine one packet at a time, meaning we only inject one packet into the network and only allow one packet traversing through the network. In this case, we can reduce the amount of packets that can cause conflicts. However, this does not solve the problem completely because as shown in the previous example, a packet can get multicasted and arrive at the same switch later.
- Therefore, secondly, we need to use a non-deterministic flooding technique. For this technique, we non-deterministically select one egress port to forward the packet instead of forwarding it to all egress ports. This will prevent the case shown in the example from happening. Whenever a packet is sent to an ingress port, it will form a directed non-branching link towards its destination and this non-branching link can guarantee that there is no more than one packet arriving at the same location unless there is a forwarding loop.

- Thirdly, we need to make sure the network does not have forwarding loops. As mentioned in the previous sub-section, since each set of link variables is only able to encode one packet for the first time it arrives at the link, our model will fail at handling the case that a packet arrives at a link multiple times. Our modeling detects when a forwarding loop is present and prevents a packet from re-entering the loop.

- Then, one may ask “will the non-deterministic flooding limit the capability of the model?” The answer is “NO” because of the fourth point: we only consider a set of networking properties or tasks that focus on a single non-branching packet trace. A single packet trace is defined as a tuple, which is composed of the packet header information and a packet trace: \((\text{PktHeader}, (\text{Sw}_1, \text{Sw}_2, \ldots))\). The packet trace is a total switch order in which the packet visits switches throughout the network. By “single packet trace” properties, we mean that it only needs a single non-branching packet trace to prove the properties are violated, i.e. the counter-example of the property is a single packet trace. Single packet trace properties include a wide range of important network properties, including absence of forwarding loops, packet reachability, etc. For example, absence of forwarding loop specifies that there should be no loop in the network such that the data plane can ever forward a single non-multicasted packet to
the same destination more than once. As long as there is one packet that reaches a switch it has visited before, there must exist one forwarding loop in the network. It means we only need one looping packet trace to prove that property is violated.

Next, we will discuss how to encode the model using Boolean formulas for network and switches.

### 2.3 Network Modeling

The relevant notation in the encoding is listed in Table 2.2.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N)</td>
<td>The formula to encode the network</td>
</tr>
<tr>
<td>(S_i)</td>
<td>The formula to encode switch (i)</td>
</tr>
<tr>
<td>(l_i^{\text{in}})</td>
<td>The set of variable to encode the packet value at the network ingress port (i)</td>
</tr>
<tr>
<td>(l_i^{\text{out}})</td>
<td>The set of variable to encode the packet value at the network egress port (i)</td>
</tr>
<tr>
<td>(l_{i,j}^{\text{sw,in}})</td>
<td>The set of variables to encode the packet value at switch (i)'s ingress port (j)</td>
</tr>
<tr>
<td>(l_{i,j}^{\text{sw,out}})</td>
<td>The set of variables to encode the packet value at switch (i)'s egress port (j)</td>
</tr>
<tr>
<td>(N^{\text{pkt}})</td>
<td>The number of bits in the packet header</td>
</tr>
</tbody>
</table>

Table 2.2: Notation for the model

The network is represented as a Boolean formula \(N\) with the input being the set of global ingress ports represented by the set of Boolean variables \(\{l_i^{\text{in}}\}\) and output being \(\{l_i^{\text{out}}\}\). The subscript \(i\) is the index of the ingress/egress port. By default, \(l_{i,0}^{\text{in}}\) and \(l_{i,0}^{\text{out}}\) are called the validity variable of the link \(i\). If the validity variable is 1, it means there is a packet arriving at the corresponding link. All the other variables for a link represent the packet header information of the packet that arrives at the link, which are \(\{l_{i,1}^{\text{in}}, \ldots, l_{i,N^{\text{pkt}}}^{\text{in}}\}\) and \(\{l_{i,1}^{\text{out}}, \ldots, l_{i,N^{\text{pkt}}}^{\text{out}}\}\). \(N^{\text{pkt}}\) is the size of the packet header. All these variables are called the packet variable of a link and they may contain both
integer variables and Boolean variables. For example, we can use 64 Boolean variables if \( N^{pkt} = 64 \), to represent source IP addresses and destination IP addresses and we can use 2 additional integer variables to represent the source port and destination port (representing integers in SAT has been discussed in section 2.1.1). This encoding can also include other fields including MAC addresses, IP protocol index, etc. In addition, we can also represent packet encapsulation by duplicating the entire packet header. If the fields we are interested in are just IP addresses, we can use 64 variables for the original header and we can add another 64 variables (i.e. \( N^{pkt} = 128 \)) to support packet encapsulation.

The switch links \( l_{sw,in}^{i,j} \) and \( l_{sw,out}^{i,j} \) are encoded exactly the same as the global ingress/egress ports. The subscript \( i \) encodes the index of the switch which the link belongs to. The subscript \( j \) encodes the index of the link which this set of variables represent. Now, we will discuss how to encode the switch connection. Since the switches are connected through those unidirectional links, we need to make sure that two switches have the same value for the link shared by them. If switch \( i \)’s ingress link \( x \) is connected to switch \( j \)’s egress link \( y \), then \( \forall k : l_{sw,in}^{i,x,k} \leftrightarrow l_{sw,out}^{j,y,k} \) and \( \forall k : l_{sw,in}^{j,y,k} \leftrightarrow l_{sw,out}^{i,x,k} \). For convenience, we can simply share the two set of variables in the real encoding. This constraint also holds for all switches and the network’s global ingress/egress ports.

The reason why we include the packet header bits in each link is to support packet modification in the switches. However, if no switch can change the packet header, for example, if there is no NAT in the network, then because of the single-packet view described in section 2.2, we are sure that all packets in the network will have the same header. In that case, the link variable only needs to specify whether a packet reaches the link and thus, we can use the validity variable to represent the entire link. In addition, we use a global packet header variable \( pkt \) to capture the packet header information to reduce the size of the formula. A small example is shown in
Figure 2.5: Encoding Switch Connectivity: An Illustrative Example

Figure 2.5 Assume all the light links representing the corresponding link variables are assigned to 1 and all the black links are assigned to 0. In the example, the packet is sent to the network ingress port 1. Switch A floods the packet to its port 2 and port 3. Switch B forwards the packet to go through port 2. Switch C forwards it to port 3. Switch D completely blocks the packet. In sum, the packet sent to network port 1 is able to reach 4 links. As shown in this example, this Boolean encoding of the network can represent the entire lifetime of a packet through the network.

If the input and output of $N$ are the ingress/egress links, the majority of the packet forwarding functionality is encoded inside the switch and we have $N = \bigwedge_i S_i$. When the network receives a packet from one global ingress port (i.e. one $l^{\text{in}}_{i,0}$ being true), the network encoding $N$ is able to calculate what the output will be by propagating the value through the switch encoding.

### 2.4 Switch Modeling

The notation related to switch variables is provided in Table 2.3 and a small example is shown in Figure 2.6. A switch is composed of an Input Aggregation Module, and a matching table. Thus, $S_i = I_i \land M_i$. 27
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_i$</td>
<td>The variable to encode the Input Aggregation Module for switch $i$</td>
</tr>
<tr>
<td>$M_i$</td>
<td>The variable to encode the matching table for switch $i$</td>
</tr>
<tr>
<td>$s_i$</td>
<td>The variables to represent the packet selected by the Input Aggregation Module</td>
</tr>
</tbody>
</table>

Table 2.3: Switch Encoding Variables

2.4.1 Input Aggregation Module

The Input Aggregation Module is a packet selection module to select which packet should pass through and match against the matching table when multiple packets arrive at the same switch. Even though we are assuming there is only a single packet trace in the network, multiple packets can arrive at the same switch in the case of a forwarding loop. For example, if a packet visits $A \rightarrow B \rightarrow C$, and then returns back to B. Switch B will see two packets, one from A and one from C. Therefore, when there is a forwarding loop in the network, we need to use the Input Aggregation Module to select one incoming packet to go through. Details about how to check a forwarding loop is postponed to Chapter 3. The output of the Input Aggregation Module is $s_i$, and similar to those link variables, it is composed of $1 + N_{\text{pkt}}$ variables.

The packet selection is done through an integer input selection variable, $E_i^{\text{in}}$. Assume switch $i$ has $N_i^{\text{port}}$ ports. Then,

$$
\forall k \in [1, N_i^{\text{port}}]: s_{i,k} = \left( \bigvee_{j \in [1, N_i^{\text{port}}]} (l_{i,j,k}^{\text{sw, in}} \wedge (E_i^{\text{in}} \leftrightarrow j)) \right)
$$

28
with additional constraint of

\[ s_{i,0} = \left( \bigvee_{j \in [1, N_{\text{port}}]} l_{i,j,0}^{\text{sw,in}} \right) \]

to ensure that if there is a packet reaching the ingress ports, the validity variable of \( s_i \) must be one.

### 2.4.2 Matching Table

As discussed in section 2.2.1, a matching table matches the incoming packet against all the rules in order of priority inside a matching table and calculates the output packet header information and the set of egress ports the matching table should send the packet to. Each rule is composed of a matching field and an action field. Each action includes a modification field and a forwarding field. The overall structure of the encoding is outlined in Figure 2.7. Each rule has two inputs, the internal packet of the switch, \( s_i \), and another Boolean variable \( r_{k-1} \) to indicate if \( s_i \) has matched higher
priority rules. \( r_{k-1} = 1 \) indicates that \( s_i \) has matched one of the rules \( m \in [1, k-1] \).

The output of a rule is a set of variables, \( t_k \), which has the same structure as the link variables and \( s_i \). \( t_{k,0} \) is equal to 1 if and only if \( r_{k-1} = 0 \) and \( s_i \) matches the matching field specified by the rule presented by variable \( u_k \). \( r_k \) and \( t_{k,0} \) are defined as follow:

\[
t_{k,0} = (u_k \land \neg r_{k-1}) \quad \text{and} \quad r_k = (r_{k-1} \lor t_{k,0})
\]

The rest of the variables in \( t_k \) are determined by the action field of the rule. An Output Aggregation Module is attached at each egress port to aggregate all the packets from the rules with an action to forward the packet to the corresponding egress port.

### Matching Field

\( u_j \) is composed of several matching expressions for either wildcard matching or range matching. If a field \( k \) is a wildcard, then

\[
u_{j,k} = \begin{cases} 
    s_{i,k} & \text{if field } k \text{ is 1} \\
    \neg s_{i,k} & \text{if field } k \text{ is 0} \\
    1 & \text{if field } k \text{ is x}
\end{cases}
\]

If the field \( k \) is a range matching, then

\[
u_{j,k} \leftrightarrow (s_{i,k} \leq \text{UpperBound}) \land (s_{i,k} \geq \text{LowerBound})
\]

and \( u_j \leftrightarrow (\land_k u_{j,k}) \).

### Action Field

The output of the modification field is \( t^0_j \). The modification field describes a logic function of how to transform \( \{s_{i,1}, ..., s_{i,N\text{pkt}}\} \) to \( \{t^0_{j,1}, ..., t^0_{j,N\text{pkt}}\} \). Since our Boolean model can support any propositional logic, we support any logic transformation on the header fields as long as we can express it using propositional logic. For example,
we can assign \( t_{j,1} \) to 0 or 1. We can assign \( t_{j,1}^0 \) to be \( \neg s_{i,1} \) to model bit inversion. We can also assign \( t_{j,1}^0 \) to be any logic combination of \( s_i \), such as making \( t_{j,1}^0 = (s_{i,2} \land s_{i,1}) \lor (\neg s_{i,2} \land \neg s_{i,1}) \).

The forwarding field describes which egress port the rule should send the processed packet \( t_j^0 \) to and this framework also supports a wide variety of forwarding actions:

- **Simple forwarding:** Each rule can simply connect the output to the Output Aggregation Module of the specified egress port to model the simple forwarding action.
- **Broadcasting:** A rule can broadcast or flood a packet \( t_j^0 \) to \( t_j^1, \ldots, t_j^{N_{port}} \) but flooding has two modes: non-deterministic mode and all-flooding mode. As discussed in section [2.2] if we consider single-packet trace, a rule should only flood the packet to one of its egress ports non-deterministically as discussed in Section [2.2.5]. In that case, we use a selection variable \( E_{j}^{\text{flood}} \) to choose which port to send to, i.e. adding \( \bigvee_k \left( (E_{j}^{\text{flood}} \leftrightarrow k) \land (t_j^0 \leftrightarrow t_j^k) \right) \) as a global constraint.
- **Multicasting:** Multicasting is similar to broadcasting but with the constraint that the set of ports we constrain on \( E_{j}^{\text{flood}} \) is the egress ports this rule multicasts to instead of broadcasting to all egress ports.
- **Forwarding to incoming port:** A rule can also forward a packet to its incoming port. The encoding is also done similar to the non-deterministic flooding but with the constraint that \( E_{j}^{\text{flood}} \leftrightarrow E_{i}^{\text{in}} \).
- **Blocks a packet:** A firewall may decide to block a packet based on the policy. A blocking action can be implemented by not connecting the output of the rule to any egress port’s Output Aggregation Module. In order to support the kind of firewall shown in Figure [2.6], we can only have a binary bit as the output and we use a disjunction to aggregate all the outputs of the rules with a “permit” action. If a packet does not match any “permit” rules, it is automatically rejected.
The Output Aggregation Module attached at each egress port aggregates all the packets sent to the egress port together. Since a packet can only match one rule at maximum, there is at most one set of $t_k$ variables with $t_{k,0}$ being 1. Assume the inputs to the Output Aggregation Module of $l_{i,j}^{\text{sw, out}}$ are $\{t_{i,j}^k\}$. Then, we have $\forall m: l_{i,j,m}^{\text{sw, out}} = \bigvee_k (t_{i,j,0}^k \land t_{i,j,m}^k)$.

## 2.5 Chapter Summary

We present a propositional logic framework for network data plane modeling. This framework provides the following advantages: it avoids the state space traversal of model checking based methods and avoids the limitations of a DNF representation inherent in ternary symbolic simulation. Because of the tight coupling of temporal and spatial information, our Boolean model does not need to bookkeep the packet history. However, on the other hand, our model precludes the case that multiple packets arrive at the same location. The model has strong capability of packet modification and processing.

In the next few chapters, we will discuss how to use the Boolean model to solve a range of networking problems.
Chapter 3

Data Plane Verification

Data plane verification deals with properties related to the outcome of individual packets through the network. This is of particular interest since any network fault must manifest itself as an undesirable outcome for some packet through the network. In this chapter, we will explain how to use the Boolean model formulated in Chapter 2 to verify network data plane. This chapter is organized as follows: we will explain what networking properties are covered in this work and how they are incorporated into the Boolean model we described in the last chapter in section 3.1. Section 3.2 discusses the details about all the experiments we ran to demonstrate the scalability and feasibility of this work. In the end, we draw conclusion in section 3.3. This chapter is mainly based on my publication [63].

3.1 Network Properties

This section discusses the network properties studied in this work. As discussed in the last section, all the properties are single packet trace properties, i.e. we only need one packet trace as the counterexample to prove the properties are violated. Moreover, all properties are functional, meaning they all deal with whether a packet can follow the desired route to reach the destination. We do not consider the properties related
Figure 3.1: Weaker version of forwarding loop

to packet latencies, queueing, and other quantitative aspects of a network. These are beyond the scope of this work.

3.1.1 Forwarding Loop

Property Definition

The definition of a forwarding loop is that there exists a packet in the network that can arrive at the same network location more than once. A weak version of forwarding loop considers the network location to be a switch, i.e. a network device. At this granularity, it is possible that the packet is forwarded to different egress ports for the different times it arrives at the switch where it loops. It is possible that the packet only loops through the network once and then gets forwarded as normal again. For example, in Figure 3.1, the packet enters the network through switch B and arrives at B again when it goes through switch C and A. However, for the second time it arrives at B, the packet gets forwarded to the host. There are several scenarios that can cause this situation to happen. Maybe the packet header gets modified along the route and switch B forwards the modified packet to a different port.

In contrast, in the stronger version of forwarding loop definition, the network location refers to the network ports instead of switches and with a constraint that the packet header is also the same no matter how many times it loops through the
network. In this case, we can guarantee that the packet does indeed get stuck in the loop. If the data plane is stateless, meaning a switch cannot keep a record of how many times a packet visits it, the switch will always have the same forwarding action for the same packet and it will guarantee to form a loop.

Even in the presence of a forwarding loop, the looping packets will eventually get discarded because there is a *Time-to-Live (TTL)* field in the packet header. Whenever a switch routes a packet, it will first check if the TTL field is non-zero. If it is, the switch decrements the field by 1 and forwards it to the next hop. If the value is zero, the switch immediately drops the packet. In most cases, a forwarding loop still reduces the efficiency of a network because every switch along the loop has to forward the packet over and over again and this consumes the CPU resource and packet storage at each switch.

Further, there may be forwarding loops that are not reachable. An example is shown in Figure 3.2. The network’s data plane state supports a packet to loop through switch A, B, and C but the packet is dropped at every network ingress port due to reasons such as firewall blocking. Hence, there is no valid packet that can traverse into the network to trigger the loop. Even though there is a real forwarding loop in the data plane, the loop is never reachable in real world and thus not a cause for concern.
In this work, we will mainly focus on the weak version of forwarding loops because it fits into our Boolean model well. Since the stronger version requires the model to represent the same packet arriving at the same port for multiple times, it is expensive for our Boolean model to support that.

**Encoding**

In addition to the network encoding, in order to verify the properties, we need to add another set of property formulas to the entire encoding. The encoding of forwarding loop checking is based on how to find the counterexample of the property. Since forwarding loop checking is a single packet trace property, we only need to find a packet that can loop through the network. There are a few constraints to add to the formula:

- **Constraint 1:** There should only be one packet entering in the network. This is due to the temporal constraint described in Section 2.2.5. The variables to represent the set of network ingress ports are \( l_{i,0} \). The formula to encode this constraint is \( \bigvee_i l_{i,0} \land \neg \bigvee_{i \neq j} (l_{i,0} \land l_{j,0}) \) for SAT. The first part \( \bigvee_i l_{i,0} \) is to specify there is at least one packet at the ingress ports and the second part \( \neg \bigvee_{i \neq j} (l_{i,0} \land l_{j,0}) \) specifies no pair of ingress ports can have valid packets at the same time. The formula is \( \sum_i l_{i,0} = 1 \) for ILP.

- **Constraint 2:** No packet exits from the network, i.e. there should be no packet at any egress ports. Since we are trying to find a looping packet, that packet should never exit the network. The formula to encode this is \( \bigwedge_i \neg l_{i,0} \) for SAT or \( \sum_i l_{i,0} = 0 \) for ILP.

- **Constraint 3:** The previous two conditions allow a corner case that a firewall blocks a packet. Therefore, se should constrain that no switch blocks packets.
We can constrain that if there is a packet accepted by the Input Aggregation Module, none of the matching rules with a “Drop” action should be matched.

**Unreachable Forwarding Loop:** The constraints discussed above can guarantee to find a forwarding loop but it does not necessarily return a reachable loop. For example, just based on the listed conditions, the SAT solver can return a solution as shown in Figure 3.3. In the figure, the packet that enters the network has a destination IP address: 10.0.0.1 but the packet that loops through the network actually has a destination IP address of 10.0.0.2. This can happen if the Input Aggregation Module at switch B accidentally selects the packet sent from switch A to pass through and ignores the packet from the network ingress port. If this happens, the real packet sent to the network is cut off from the rest of the network. Therefore, we need a technique that can make switch B always choose the packet that is sent to the network ingress port.

The solution is to add a flag bit to all the packet encodings to indicate if a packet has entered a loop. In Figure 3.3, we define the flag bit such that the bit is set to 0 for packets injected into the network (with dst: 10.0.0.1) because they have not entered a loop. The packet that is actually looping (with dst: 10.0.0.2) has entered a loop and has a flag bit 1. If the Input Aggregation Module sees two valid packets at the ingress ports, there must be a forwarding loop due to the single packet trace.
assumption. Then, the module always chooses the packet with the flag bit 0 and sets
the flag bit of the output of the Input Aggregation Module to be 1 since it enters a
loop. Since the Input Aggregation Module sees the flag bit as 1 for the packet sent
from switch A, it recognizes that this packet has looped back and does not select it
for further processing.

In summary, if the SAT solver or ILP solver returns a solution that satisfies all
the previous constraints, it represents a forwarding loop. Since there is no switch
blocking packets (because of constraint 3), as long as there is a packet injected into
the network, there are only two possibilities for the packet. It can either exit the
network, which is not allowed by the constraint 2, or get stuck in a forwarding loop.
Further, the loop must be reachable since the switch where it starts looping (switch B
in the previous example) always chooses the packet from the network ingress ports. If
the solvers cannot find a satisfiable solution, no forwarding loop exists in the network.

3.1.2 Reachability Checking

Property Definition

Reachability checking is to check whether a set of packets with headers: \(\{\text{pkt}_1, \text{pkt}_2, \ldots\}\),
can arrive at the specified network locations as desired. There are two flavors of
reachability checking:

- **Flavor 1:** Every packet in \(\{\text{pkt}_1, \text{pkt}_2, \ldots\}\) can reach from a network ingress
  port to a destination. This is a "\(\forall\)" property.

- **Flavor 2:** There exists some packet in \(\{\text{pkt}_1, \text{pkt}_2, \ldots\}\) that can reach from a
  network ingress port to a destination. This is an "\(\exists\)" property.

The destination in the definition can refer to both switches and network ports. For
example, we can also check whether a packet can reach from a switch to a port.
Flavor 1 is useful in checking the connectivity between a source ingress port and a destination. For example, the network administrators may want to check whether all UDP packets with a specific destination IP address are able to arrive at the corresponding location. Flavor 2 is useful in the following scenario: assume a network that blocks all SSH traffic. The Flavor 2 can be used to find whether there exists an SSH packet that is permitted between two locations.

Further, a generalized reachability checking can also support **Waypointing** and **Blacklisting**. Waypointing specifies that besides the destination, a packet should also visit certain other locations along the route to the destination. Waypointing can be used in scenarios such as that an administrator wants to guarantee all packets have to go through a firewall before arriving at the egress ports. On the contrary, blacklisting constrains that the specified packets should never visit certain locations. For example, there may be some switches that are likely to be compromised by hackers and all packets have to detour and avoid those switches.

**Encoding**

In order to check the reachability of flavor 1, we need to find a counterexample that a packet injected to the source location with the specified header constraint does not reach the destination. Assume the source location is $l_{i}^{in}$ and the destination is $l_{dst}^{dst}$. If the destination is a switch $j$, then $l_{dst}^{dst} \leftrightarrow s_{j}$. If it is egress port $k$ at switch $j$, then, $l_{dst}^{dst} \leftrightarrow l_{out}^{out}_{j,k}$. The constraint on the packet header is specified by function $C(.)$. We need to satisfy the following constraints to specify the property:

- **Constraint 1:** We need to inject a packet to $l_{i}^{in}$ with header constraint $C(.)$. This can be encoded as $l_{i,0}^{in} \wedge C(l_{i}^{in})$.

- **Constraint 2:** We need to constrain that the specified destination does not receive a packet because we need to find a counterexample as discussed above. Then, we can use the following formula $l_{0}^{dst} = FALSE$. 

39
As long as there exists a satisfiable solution to the above constraints, the solution must be a counterexample to the property.

Checking the reachability of flavor 2 is a little different from checking for flavor 1. Since flavor 1 specifies a $\forall$ property, then finding the counterexample is a “$\exists$” problem. SAT or ILP are essentially $\exists$ problems because they try to find if there exists a satisfiable solution to the given formula. Therefore, finding the counterexample of flavor 1 reachability fits well in our Boolean model. However, flavor 2 is already an $\exists$ property. We are able to directly check the property. Therefore, we use the following constraints to check flavor 2 reachability:

- **Constraint 1**: Similarly, we need to inject a packet to $l_{i,0}^{in}$ with header constraint $C(.)$, which can be encoded as $l_{i,0}^{in} \land C(l_{i,0}^{in})$.

- **Constraint 2**: Then, we need to specify the corresponding destination receives a packet, which can be encoded as $t_0^{dst} = TRUE$.

### 3.1.3 Slice Isolation

**Property Definition**

In a virtualized environment, network systems partition resources into different **slices** so that each tenant can use their own slice of the network resource without noticing other users sharing the same physical network. A slice can be viewed as a collection of the tuples of packet header and packet locations, such as ports. Each slice should be absolutely isolated from others to create a completely private environment to avoid privacy leaking. The **slice isolation** property checks that a packet from one slice does not cross over to another slice. For example assume $s$ is a switch, $p_{s,1}^{in}$ is an ingress port of $s$, $p_{s,2}^{out}$ is an egress port of $s$, and $h_1, h_2$ are two packet header. Let $(h_1, p_{s,1}^{in})$ be in Slice 1 and $(h_2, p_{s,2}^{out})$ in Slice 2. We need to guarantee that $h_1$ at $p_{s,1}^{in}$ is not processed by switch $s$ into $h_2$ at $p_{s,2}^{out}$.
Encoding

The basic idea of checking slice isolation is to find a counterexample that violates the property. The encoding is as follows:

- **Constraint 1:** We need to add a slice index variable $l_{i,j}^{in,slice}$ to each link variable set, $l_{i,j}^{in}$, to indicate which slice the packet at the link belongs to. For example, assume slice 1 is defined as $\{(h_1, p_{in}^{s1}), \ldots\}$. Then, the constraint is $\land ((l_{i,1}^{in,slice} \leftrightarrow 1) \land (l_{i,1}^{in} \leftrightarrow h_1))$.

- **Constraint 2:** Similar to other properties, we need to inject a packet to an ingress port. The formula to encode this constraint is $(\lor_i l_{i,0}^{in}) \land (\neg \lor_{i \neq j} (l_{i,0}^{in} \land l_{j,0}^{in}))$ for SAT and $(\sum_i l_{i,0}^{in}) = 1$ for ILP.

- **Constraint 3:** Check if there is a leaking from one slice to another slice. We need to check this at each switch locally. The counterexample represents that the incoming packet belongs to a different slice than the outgoing packets. The slice index the incoming packet belongs to can be encoded as $Index_{i}^{in} = \lor ((c_{i}^{in} \leftrightarrow j) \land (l_{i,j}^{in,slice}))$. The $c_{i}^{in}$ variable is to choose which packet the $Index_{i}^{in}$ represents. Similarly, the slice index of the egress ports is $Index_{i}^{out} = \lor ((c_{i}^{out} \leftrightarrow j) \land (l_{i,j}^{out,slice}))$. Then, finding the counterexample is $\lor_i (Index_{i}^{in} \neq Index_{i}^{out})$.

Because constraint 3 constrains the violating condition, the satisfying solution must return a packet that leaks from one slice to another. Therefore, the constraints above can guarantee to find a reachable slice isolation violation.
3.1.4 Network Equivalence Checking

Property Definition

In addition to the previous properties that focus on a single network, our framework can also check whether two networks can have exactly the same forwarding behavior for every packet entering the networks. This can be useful when the network administrators want to experiment with network topology or different security policies. This can also be used to remove redundant rules in the data plane because if a rule is redundant, removing it would not change the forwarding behavior of a network.

In order for two networks to have the same forwarding actions for each packet, their network ingress/egress ports must match each other, i.e. for every port in network A, there must exist a counterpart in network B and vice versa but their internal topology and switch connections do not have to be the same. We define the mapping of the ports between two networks as a set of tuples, $M = \{(l_{i,j}^{in,A}, l_{k,l}^{in,B})\}$. We define two networks are equivalent if for every packet entering the networks through their matched ingress ports, the two networks will send them to the same matched egress ports.
For example, as shown in Figure 3.4, both networks send the packet arriving at the switch B to the egress port connected to switch A and switch C. Therefore, the two networks are equivalent although their internal structures are different.

Encoding

Similar to the previous properties, in order to find a counterexample, we look for a packet that causes the two networks to have different forwarding actions. The encoding is as follows:

- **Constraint 1:** The ingress ports of the two networks should completely match each other. \( \bigwedge_{(l_{in,A}^{i,j}, l_{in,B}^{k,l}) \in M} (l_{in,A}^{i,j} \leftrightarrow l_{in,B}^{k,l}) \).

- **Constraint 2:** We need to inject one packet into the network, which is exactly the same as how we encode it in forwarding loop checking.

- **Constraint 3:** The counterexample looks for whether there is an egress port that has different outcomes. We can use the following to encode this:
  \( \bigvee_{(l_{in,A}^{i,j}, l_{in,B}^{k,l}) \in M} (l_{in,A}^{i,j} \neq l_{in,B}^{k,l}) \).

Because constraint 3 specifies to find a conflicting egress state, the satisfying solution must be a counterexample that violates the equivalence between the two networks.

3.2 Experimental Results

We have implemented a tool called NetSAT based on our approach and conducted a series of experiments to test its efficacy. The first test benchmark used is the publicly available Stanford backbone network [28]. The second set of benchmarks are synthetically generated. In these benchmarks, we used the Fat-Tree topology [7] which is a commonly used topology in data-center networks. The size of a Fat-Tree
network is proportional to the number of ports each switch has. If all switches have $L$ ports, the total number of switches $N$ is given by $N = 5L^2/4$ and the number of global network ports is $L^3/4$ [7]. Each network port is assigned one IP prefix to represent the IP addresses it owns. The matching rules in the network were generated by randomly selecting a pair of global network entry ports and using breath-first search to find a shortest path between the two ports. We use $P$ to denote the number of ports selected or the number of different routes in the network. The matching field of each rule is based on the IP addresses of the source and the destination with some randomly generated bits. We use $H$ to denote the number of bits in the packet header.

All experiments use Minisat [18] as the SAT solver and run on Ubuntu Linux with kernel 3.2.0 and Intel Xeon processor running at 3.2GHz without using any parallelism.

**Stanford Backbone Network:** The Stanford network has 16 routers and includes full network complexity (VLAN tags, ACLs, etc.). This expands the rule set to about 15,000 rules and results in a formula with about 6.2 million CNF variables and 32 million CNF clauses. It takes about 100 seconds to return satisfiable for both forwarding loop and reachability checking. It takes about 5 seconds for unsatisfiable cases for reachability checking as in this case the packet gets dropped very quickly as it traverses through the network.

**Synthetic Benchmarks:** For the synthetic benchmarks, we ran a few sets of experiments to study the effect of the total number of rules, total number of switches, and the header size of the packet on the execution time for property checking. We report the execution time for the unsatisfiable case as that is the slower case for this benchmark set. In addition, we also ran Header Space Analysis (HSA) [28] on the same testbenches to compare the performance with our method.

For the first set of experiments, we ran both NetSAT and HSA on three different sizes of networks with $L = 4, 8, 10$ for forwarding loops. Both methods successfully
return no forwarding loops found for all cases. We tried different $H$ ranging from 64 to 192. We increased the number of routes, $P$, to demonstrate the scalability of both approaches. All results are shown in Figures 3.5 to 3.7. From all figures, we can see that NetSAT consistently performs faster than HSA. For $L = 8, 10$, NetSAT almost performs two orders of magnitude better than HSA. The reason is that our single-packet-trace based properties do not require to bookkeep the packet trace. Generally, NetSAT demonstrates a larger variance in the execution time because SAT solving algorithm sometimes uses random tie-breaking. And as the size of the network increases from $L = 4$ to 10, the execution time increases almost four orders
of magnitude because the size of the formula becomes much larger. The number of routes also plays an important role in determining the execution time.

Secondly, we ran flavor 2 reachability checking on a network with $L = 12$ with different $P$ and $H$ values. Even though the size of the network becomes much larger, both techniques finish within 10 seconds. The reason is for reachability checking, we specify the ingress port to inject the packets, which significantly reduces the searching space for both methods. In this experiment, HSA performs faster than NetSAT. The reason may be that HSA incurs less overhead for smaller problems.

3.3 Conclusion

In this chapter, we present how to check a range of properties for data plane verification using the framework we discussed in the last chapter. We formulated the checking of forwarding loops, reachability, slice isolation, network equivalence using constraint-based modeling. The essential idea of the verification technique is to find a counterexample that violates the properties. We built a tool based on SAT, called NetSAT and we compared the performance between NetSAT relative to HSA. For checking forwarding loops, NetSAT is consistently two orders of magnitude faster than HSA. The most dominating factors that affect the execution time are the size of the network and the total number of routes. Moreover, checking reachability is much faster than checking forwarding loops since the search space is much smaller.
Chapter 4

Firewall Management

In this chapter, we will mainly focus on the automatic management for firewalls. Firewalls are very common in modern networks and most private networks utilize a firewall to protect and monitor incoming and outgoing traffic, using a set of rules developed by a network administrator. Firewalls help filter contact between a network and the rest of the Internet, dropping packets that do not satisfy the access rules outlined by an administrator. These rules are designed to examine certain fields of a packet header, and subsequently either forward the packet if it is on the “PERMIT” list, or drop the packet if it is not included in the set of acceptable rules. For large, complex networks, a single firewall instance may not be enough to satisfy all the security requirements and hence, a network of firewalls or equivalently a distributed firewall is needed to protect the network 23. Since firewalls are critical to modern networks, it is important to implement them correctly and efficiently.

This chapter is divided into two parts: the management and optimization for single firewall discussed in Section 4.1 and the management for distributed firewalls in Section 4.2. The chapter summary is in Section 4.3.
4.1 Single Firewall Management

One issue that often plagues firewalls is that the rules are not always optimized, with many firewalls containing redundancies in the rule set [6]. Additionally, firewalls often contain errors, such as rules being erroneously shadowed by higher priority rules, which are difficult to detect due to the lack of available policy verification tools, thus compromising the security of the network [60]. Single-firewall verification entails using formal analysis for ensuring that a set of firewall policy rules are consistent with a specification, and includes checking for redundancies within a firewall and equivalence checking between firewalls. Single-firewall verification can assist in discovering superfluous policy rules that can slow down the travel time of packets to and from a network. Verification can also help ensure that even if a rule set does not contain any conflicts, it is prioritized in the correct sequence to avoid undesired packet denial. Additionally, one can use inclusion checking to determine if one firewall has a stricter policy than another firewall by permitting only a subset of packets allowed by the second firewall.

In this section, we present a SAT based method of firewall verification for equivalence checking between two firewall rule sets to check if they have the identical behavior for every incoming packet. We also use our SAT-based verification tool for inclusion checking between two firewalls, which is very helpful in checking if one firewall has a stricter policy than another firewall. We also demonstrate the application of verification in detecting redundancy in a firewall. We additionally present a single-firewall synthesis method based on a Quantified Boolean Formula (QBF) formulation and use a QBF solver to synthesize an optimal firewall. This section is organized as follows: Section 4.1.1 presents the background of this section, i.e. an overview of the firewalls that we are targeting. Section 4.1.2 and Section 4.1.3 discuss the technical details on encoding the firewalls and verification properties as logic formulas. Following the discussion of the encoding, we present experimental data that tests our
encoding and formulation in Section 4.1.4. Next, we discuss the related work and finally draw conclusions in Section 4.1.5 and Section 4.1.6.

4.1.1 Background

Firewalls

The firewalls discussed in this section are abstracted as a one-input one-output device that can model both software-based and hardware-based firewalls. A firewall takes packets from the network as an input and returns the decision about whether the packets should be dropped or permitted. Therefore, a firewall can be thought of as a simpler version of a switch as described in Section 2.4 but with only a single packet input and a binary output. The main component of a firewall is still a matching table, which is composed of strictly prioritized matching rules. Similar to the prioritized matching table discussed in Section 2.4, each matching rule has two fields, one matching field and one action field but the action of a firewall is simply “DROP” or “PERMIT” action, which specifies whether the packet should be dropped or not.

The matching table is considered to be completely static during verification. Although stateful firewalls usually change state during operation, they can be regarded as static between two matching rule updates. Hence our firewall is a snapshot of stateful firewalls at a single instance of time.

Firewall Verification and Synthesis

Firewall Equivalence Checking: One of the major properties that is studied in this work is firewall equivalence. Given two firewalls, we determine if they have identical behavior, i.e. they DROP/PERMIT exactly the same set of packets. This is an important property because it can be used in firewall optimization to compare whether a given firewall and an optimized firewall are identical. The fundamental
principle in firewall equivalence checking is trying to find the difference between two firewalls. If no differences exist between them, the two are equivalent.

**Firewall Inclusion Checking:** Another relationship between two firewalls is inclusion. If firewall A permits only a subset of the packets permitted by firewall B, then we say firewall A is stricter than firewall B since it drops more packets. Often, when we want to replace one firewall with another, we have to make sure that the new firewall does not sacrifice the security requirements of the old one and is at least as strict as the replaced firewall.

**Firewall Rule Redundancy Checking:** A firewall may also have redundant rules within its own matching table \[32\]. We define a redundant rule as one that once removed from the rule set produces a matching table that is logically equivalent to the original ruleset. For example, one rule may accept packets destined to 10.0.0.0/32 while another rule accepts all packets destined to 10.0.0.0/24. In this case, if the first rule is removed, all packets destined to 10.0.0.0/32 are still accepted by the latter rule, rendering the first rule redundant.

**Firewall Synthesis:** The number of rules can affect the performance of firewalls, especially for firewalls based on sequential matching. Therefore, network administrators desire to have as smallest number of rules as possible in the firewall. Firewall synthesis is concerned with synthesizing a firewall with exactly the same behavior as a given firewall such that the synthesized firewall’s specifications have the smallest number of rules installed. Here, we use a *Symbolic Firewall*, which is parameterized in the number of the rules. The advantage of a symbolic firewall is that we can use variables to program the firewall and an optimal synthesis determines the values of the variables and thus the rules. We describe all the technical details for the following sections.
4.1.2 Firewall Equivalence, Inclusion, and Redundancy Checking

In this section, we discuss the details of how we check the equivalence and inclusion between two firewalls, and redundancy checking within a firewall. We will first present how to encode a fixed firewall, in which all the rules are given and fixed and then we will present how to use the encoding of a fixed firewall to do firewall equivalence, inclusion, and redundancy checking.

The Encoding of Fixed Firewalls

Figure 4.1 depicts the basic structure of our firewall encoding. As mentioned in section 4.1.1, the firewall is a one-input one-output device. The input is a packet, which is modeled as a set of Boolean variables. We use $b_i$, $i \in [1, N_{pkt}]$ to represent the Boolean variable for the input bit $i$ and $N_{pkt}$ is the total number of input bits needed for the packet. The input packet is fed to the matching table. The encoding of the matching table is exactly the same as the one discussed in Section 2.4 except for the output. Since a firewall only has one output, we can easily use an OR operation to aggregate all the output of rules with a “PERMIT” action.

Assume we use $m_i$ to indicate whether the packet matches this rule and use $p_i$ to indicate if the input has matched some rules with higher priority than rule $i$. For any $i$ that is smaller than $j$, rule $i$ has a higher priority than rule $j$. We use $r_{i,j}$ to
represent the value of the matching field bit for rule $i$ and bit $j$. For example, if rule #10 matches on “010101xx” (“x” is a wildcard expression as discussed in Chapter 2), then $r_{10,1} = 0$, $r_{10,4} = 1$, and $r_{10,7} = x$. We use $k_{i,j}$ to represent the matching for that bit. Then,

$$k_{i,j} = \begin{cases} b_j & \text{if } r_{i,j} \text{ is 1} \\ -b_j & \text{if } r_{i,j} \text{ is 0} \\ \text{TRUE} & \text{if } r_{i,j} \text{ is x} \end{cases} \quad (4.1)$$

$$m_i = \left( \bigwedge_{j=1}^{N_{pkt}} k_{i,j} \right) \land \left( \neg p_i \right) \text{ if } i \geq 1 \quad (4.2)$$

$$p_i = \begin{cases} \text{FALSE} & \text{if } i == 1 \\ m_{i-1} \lor p_{i-1} & \text{if } i \geq 2 \end{cases} \quad (4.3)$$

Then the output of the firewall, $F(B)$ is

$$F(B) = \bigvee_{i \in S_{\text{permit}}} m_i \quad (4.4)$$

where $B = \{b_1, ..., b_{N_{pkt}}\}$ and $S_{\text{permit}}$ is the set of rules with an action of “PERMIT”. If the formula evaluates to 1, the firewall permits the packet; otherwise, it drops the packet. We make the assumption that the default rule is a ‘DROP’ rule, which means that all the packets which do not match any rules specified in the firewall will be dropped by default.

**Firewall Equivalence and Inclusion Checking**

To check if two firewalls are equivalent, we can use an XOR operation to connect the output of the two firewalls and see if it is different. For example, given two firewalls $F_1(B)$ and $F_2(B)$, if $F_1(B) \oplus F_2(B)$ is satisfiable, it means there exists at least one
input packet B, such that the two firewalls differ in their actions; otherwise, there is no such packet that exists.

To check inclusion, we need a formula that can only return false if the stricter firewall returns true but the other firewall returns false. Thus, \( \neg (F_1(B) \rightarrow F_2(B)) \) is used to check if firewall 2 includes firewall 1. If this formula is unsatisfiable, this means that firewall 1 can never permit a packet that is blocked by firewall 2, i.e. firewall 1 permits a subset of packets permitted by firewall 2 and it is stricter than firewall 2. If it is satisfiable, the satisfying assignment to the input packet bits serves as a counterexample for the inclusion property.

**Firewall Rule Redundancy Checking**

To check if a firewall contains redundancies amongst its rules, we need to check whether a matching table created by the removal of a rule (or multiple rules) is equivalent to that of the original unmodified matching table. We add a control bit for each rule so that we can control whether that rule is present in the ruleset or not. We use \( o_i \) to represent the control bit and the formula for \( m_i \) is

\[
m_i = \left( \bigwedge_{j=1}^{N_{input}} k_{i,j} \right) \land (\neg p_i) \land (\neg o_i) \text{ if } i \geq 1
\]

If \( o_i \) is 0, the rule behaves just like a normal rule but if it is 1, that rule will never match any packets and can be regarded as a discarded rule.

Then we can use the array of the control bits to do redundancy removal. We start from the first rule and set its control bit to be 1 and compare the equivalence between the new firewall and the original firewall. If they are equivalent, it means the first rule is redundant and can be removed. Then we keep the control bit for the first rule as 1 and do the same procedure for the second rule and so on until we find all the assignments to the control bits. This results in a minimal set of rules, i.e there is no
subset of the final set of rules that is equivalent to the original firewall. However, this may not be a minimum subset, i.e. subset of the least cardinality, as the rules were removed in a specific order. Generating a minimum subset of rules is essentially a firewall synthesis problem because the rules in the optimal firewalls may be completely different from the rules in the original rule set. Hence, only removing redundant rules is not able to provide the optimality. I will discuss our solution of firewall synthesis problem in the following subsection.

4.1.3 Firewall Synthesis

For the firewall synthesis problem, we use a symbolic firewall with rules represented as symbols instead of being fixed and compare it against the given firewall, which serves as a specification. If we can find an assignment for all symbols such that we can make the two firewalls equivalent, then these symbol values serve as the rules definition of the new firewall.

The Encoding of Symbolic Firewalls

The difference between a fixed firewall and a symbolic firewall is that the symbolic firewall has symbolic rules instead of fixed rules. As shown in Figure 4.2, the matching table has a set of symbolizers to symbolize the matching actions of each rule. Thus,
the symbolic firewall has two sets of inputs: one packet which is the same as the fixed firewall, and one set of input symbols which is used to program the firewall. The output is a single binary bit to indicate the action of the firewall.

The structure of the matching table of a symbolic firewall is the same as that of a fixed firewall but the number of matching rules is parameterized, which means that the number of rules becomes a variable, and we use \( n \) to represent it. The encoding of matching rules is completely changed and as a result the formulas for \( m_i \) and \( p_i \) do not change but the formula for \( k_{i,j} \) becomes

\[
k_{i,j} = (v_{i,j}^1 \land b_j) \lor (\neg v_{i,j}^1 \land \neg b_j) \lor v_{i,j}^2
\]

where \( v_{i,j}^1 \) and \( v_{i,j}^2 \) are two variables to program the bit at rule \( #i \) and bit \( j \). It captures that if \( v_{i,j}^2 \) is 1, \( k_{i,j} \) evaluates to TRUE to capture the ‘x’ case. If \( v_{i,j}^1 \) is 0, \( k_{i,j} \) equals to \( \neg b_j \); otherwise, it is \( b_j \) to capture the ‘1’ case.

The action of each rule is made symbolic using variable \( v_i^3 \). For every rule, the output action is \( v_i^3 \land m_i \). It means it is a PERMIT rule as long as \( v_i^3 \) evaluates to TRUE. Therefore, the output of the symbolic firewall \( F^s(B, n, V) \) is

\[
F^s(B, n, V) = \bigvee_{i \in S_{alt}} v_i^3 \land m_i
\]

where \( V \) is the set of all variables \( v^1 \), \( v^2 \), and \( v^3 \) and \( S_{alt} \) is the set of all rules.

The total number of Boolean variables required to encode a symbolic firewall with \( n \) rules is \( 2 \times N_{input} \times n + 3 \times n + N_{input} \). Since we need 2 variables to represent each matching bit, in total we have \( N_{input} \times n \) such bits. For each rule, we need another variable to encode the action and together with \( m_i \) and \( p_i \), we have an extra 3 variables for each rule.
**Firewall Synthesis**

We encode a symbolic firewall with \( n \) rules and find \( V \) such that the symbolic firewall is equivalent to the given fixed firewall, which serves as the specification. Here we use a *Quantified Boolean Formula (QBF)* formulation to determine the assignments for \( V \).

Given a firewall \( F(B) \) and a symbolic firewall with \( n \) rules installed \( F^*(B, n, V) \), we need to find an assignment for all variables in \( V \) such that for every input packet \( B \), \( \neg(F(B) \oplus F^*(B, n, V)) \) is satisfiable, and there exists an equivalent firewall to the given one with \( n \) rules installed. Expressed as a QBF formula, it becomes

\[
\exists V \forall B : \neg(F(B) \oplus F^*(B, n, V)) \quad (4.8)
\]

If the QBF solver returns satisfiable for this formula, we can build an equivalent firewall with \( n \) rules installed and the assignment to the \( V \) variables that makes the formula true defines the rules; otherwise, there does not exist such firewall. To find a firewall with the smallest number of rules installed, we do a binary search on the number of rules \( n \).

### 4.1.4 Experiments

We used Minisat [18] as our SAT engine to check for equivalence and inclusion between two generated firewalls and we use BDepQBF [36] as our QBF solver for firewall synthesis. These are considered amongst the fastest SAT and QBF solvers respectively based on results of tool competitions [31, 40]. All experiments are run on Gentoo Linux with kernel 3.4.4. We used an AMD Phenom 3.3 GHz processor with 8 gigabytes of RAM for these experiments.

To construct different firewalls to test, we used the ClassBench benchmark generator [54]. The ClassBench program allows the user to input several different parameters
for priority rule generation. For our experiments, we used the firewall parameter file input to simulate real firewall rule sets, available with the downloadable benchmarks. Additionally, we were provided with three input parameters to set, which expanded upon the parameter file: the address scope, the application scope, and the smoothness of the generated rule set. The address scope adjusts the bias of how specific address prefixes are, indicating whether longer or shorter prefixes are desired. The application scope affects the protocol specifications of the rule, and the smoothness value determines the distribution of prefixes across the protocol. For simplicity, we kept all three parameters at their default value: uniform address scope, uniform application scope, and default smoothness as set by the parameter input file.

The matching rules generated by ClassBench have 6 fields, which are source/destination IP addresses, source/destination ports, protocol, and flags. In total, they account for 136 matching bits, i.e, $N_{input}$ is 136.

**Equivalence Checking**

In order to test our SAT based equivalence checker, we wanted to generate two firewalls which were very similar to one another, but with subtle differences. This represents the difficult case for equivalence checking. We first used the ClassBench program to create rulesets of size 50 to 26000. We made a copy of each of these rulesets, but with a small mutation. We tested for three different types of mutations:

1. Flipping a random bit in the firewall.
2. Deleting one random rule in the firewall.
3. Swapping two random rules in the firewall.

For the first mutation, we chose a random bit in the entire ruleset, and depending on its value, flipped it to one of the other two bits. If the random bit was an ‘x’, we
demoted it to either a 0 or a 1, with equal probability. If the random bit was either a zero or a one, we promoted it to an x.

For the second mutation, we uniformly chose one random rule, and removed it from the ruleset. This mutation was chosen to see how well the equivalence checker could pick up a slight difference between two rulesets that are very similar.

For the third and final mutation, we chose two random rules uniformly, and swapped their positions. Since the original ruleset is prioritized, the swapping of two rules could potentially cause a portion of the ruleset to be totally ignored, as a higher prioritized rule is assumed to be more specific than any rule below it in priority.

All experiments are run on Intel Xeon machines with 3.2GHz and ubuntu 12.04. Figure 4.3, 4.4, and 4.5 show the results. As we can see, the execution time increases as the total number of rules grows since the formula size becomes larger. SAT cases (not equivalent) are generally faster than UNSAT cases (equivalent) since for SAT it often only explores part of the search tree till it finds the satisfying assignment. Another thing to note is that all the data points scatter around the plane. This is possibly because the benefit of the heuristics built inside the SAT engine can differ from case to case. Sometimes when we are lucky, it is even faster to check a larger test case. The one that executes the longest is the UNSAT case for mutation 3 with about 26000 rules. It takes about 48 minutes to finish. The SAT instance has a total of 103843 Boolean variables and 5.2 million CNF clauses.

**Inclusion Checking**

To check for inclusion of one firewall ruleset within another ruleset, we again began by developing a set of differently sized benchmarks. We then created a new firewall for each benchmark by removing the lowest priority 10% of the rules. Since it permits fewer packets to go through, our inclusion checking tool can only return UNSAT (i.e.
Figure 4.3: Mutation 1: Flipping a random bit.

Figure 4.4: Mutation 2: Deleting one random rule.
Figure 4.5: Mutation 3: Swapping two random rules.

the inclusion check will pass). Figure 4.6 plots the execution time. Because none of
the firewalls include the other, all cases are UNSAT.

Rule Redundancy Checking

To check for redundancies within a set of firewall rules, we took our original Class-
Bench files and attempted to remove rules one at a time to reduce the size of the
matching table. We attempted to remove the largest cumulative set of rules while
maintaining equivalence to the original ruleset. This process was relatively slow, as
shown in Table 4.1. For a matching table with almost 2000 rules, the process took
about 83 minutes and found approximately 1400 redundancies. This means that were
we to remove all redundancies found, the matching table would behave identically to
the full set, and be much smaller in size. Another interesting thing to notice is that
the percent of redundant rules is very high and the reason is that the default action for
a packet that does not match any rules is “DROP”. Therefore, we can remove all of
the rules that have an action of “DROP” and have no overlap with other “PERMIT” rules.
Firewall Synthesis

The firewall synthesis is prohibitively slow and the speed slows down dramatically as the size of the input packet grows. This is because in our QBF formula we have a universal quantification for the input packet bits. It can manage the firewall size of 20 input packet variables and 3 rules ($N_{\text{input}} = 20$, $n = 3$, and the size of $V$ is 123) in about 10 minutes but it times out at 1 hour for a size of 25 variables. Thus, while the synthesis problem can be encoded as a QBF problem, even simple instances are beyond the reach of the best current QBF solvers.

4.1.5 Related Work

In the past few years, there has been some work in firewall analysis and optimization. [41] proposed a method to minimize the number of rules in Ternary Content-Addressable Memories (TCAM), which are used for packet classification. Similar to our method, they express the output of a TCAM using a Boolean formula. They use Disjunctive Normal Form (DNF) and two-level DNF optimization to minimize the number of rules required. They implement a heuristic solver because the size of the problem is too large to get an exact solution. Some efforts [32, 33, 41] propose to remove redundancies in a firewall based on a decision tree but these techniques are unable to compare firewalls and do not guarantee that they are minimized. Another set of works [10, 8] systematically summarize different kinds of conflicts for both standalone and distributed firewalls. There are also some older works that target optimizing IP routing tables [16, 53]. Since an IP routing table can have multiple routing decisions instead of only two decisions for a firewall, IP routing table optimization is a harder problem than firewall optimization in terms of problem size. However, they use longest-prefix matching and it is difficult for them to adapt to priority-based matching and they also did not propose how to compare the relationship between two routing tables (e.g. subset of routes). In a broader context, many works [28, 51] dis-
cussed about the verification of general network properties, including reachability and forwarding loops. Projects such as [48, 47] have independently discussed how to use Prolog and SMT for firewall equivalence and inclusion checking but their framework cannot handle firewall synthesis problem.

4.1.6 Conclusions and Future Work

In this section, we presented a SAT based method that can check the equivalence and inclusion relationship between two firewalls and how to use SAT to remove redundancy in a firewall. These techniques are shown to scale well for practical sized firewalls using state-of-the-art SAT solvers. We also proposed using a QBF formulation to solve the firewall synthesis problem. Our formulation, even with the fastest QBF solvers, is very slow. Our ultimate goal is to develop a run-time firewall checking tool that can do firewall verification upon firewall configuration changes but this will require significant tool speedups. We are exploring the use of incremental verification techniques for this purpose. Firewalls are a simple case of a network middlebox and our technique can be applied to switch verification and synthesis. Since switches and routers usually have several forwarding ports and routers, in particular, are capable of modifying the packet header, a more complex encoding is needed to model the middleboxes. However, the fundamental principles remain the same.

4.2 Distributed Firewall Management

For large enterprise or datacenter networks, a single firewall may not be powerful enough to handle all the possible attacks. In this subsection, we focus on optimized automatic placement of the access control list (ACL) for distributed firewalls based on a high-level ACL policy description and a given network topology. Each ACL rule specifies a set of packets that are either permitted (PERMIT rule) or dropped (DROP
rule). We consider how to optimally distribute the ACL rules among the switches, with the goal to optimize criteria like the total number of ACL rules installed, while satisfying per-switch capacity constraints for ACL purposes and maintaining all ACL policies.

Instead of using the Boolean model described in Chapter 2, we formulate ACL policies as a prioritized rule list and propose a novel dependency graph based analysis. In this graph, each node represents a rule on a switch and the edges represent dependencies between the rules. These dependencies are based on PERMIT-DROP conflicts in the prioritized list of ACL policy rules. We then use this dependency graph to encode the rule placement problem as an optimization problem or a satisfiability problem, that can be solved by an ILP-solver or an SMT-solver (Satisfiability Modulo Theory) \([15]\), respectively. (The satisfiability problem can also be solved by a Pseudo-Boolean solver \([11]\).) To do so, we generate constraints based on the dependency graph that maintains the ACL policies, while taking the flow routing and switch capacities into account. Our encoding extends naturally to handle additional rule optimizations, e.g. merging rules at switches when possible. It also extends easily to different (and combinations of) objective criteria, e.g. total number of switches, distance from ingress of switches where rules are placed so as to minimize network traffic, weighted placement to favor certain switches, etc.

This work makes the following contributions:

- It provides an ILP-based formulation for the rule-placement problem starting from (i) given ACL policies in the form of a prioritized set of rules and (ii) a given routing. The resulting solution, if one exists, meets the capacity constraints of individual switches and minimizes alternate possible objective functions such as the total number of rules or the estimated traffic by placing DROP rules further upstream.
• It extends this formulation to include the merging of rules common to multiple policies for greater rule minimization.

• It provides an alternative satisfiability-based formulation of the capacity constraint satisfaction problem, including allowing for merging, that can be solved by an SMT or a Pseudo-Boolean solver.

• It provides an experimental evaluation that demonstrates the efficacy of the ILP formulation on practical sized networks.

The subsection is organized as follows. Section 4.2.1 provides the relevant background on the Big Switch Abstraction [24], which discusses how to describe the high-level behavior of a network. Section 4.2.2 formulates the rule placement problem that is the focus of this section. In Section 4.2.3 we discuss our approach to this problem and describe how to use ILP and Satisfiability solvers (SMT and Pseudo-Boolean) for its solution. This is followed by an experimental evaluation of this method in Section 4.2.4. We discuss relevant related work in Section 4.2.5 and we provide concluding remarks in Section 4.2.6.

4.2.1 Big Switch Abstraction

Recently, researchers have introduced the concept of a “Big Switch” abstraction [24] as a network specification mechanism. Since the core function of a network is packet forwarding, we can model the network as a virtual “Big Switch” as long as we have a description of the network’s behavior (Figure 4.7). This behavior can be defined by two kinds of policies:

• a routing policy to define how packets will traverse the network from an ingress to an egress

• an endpoint policy to describe other aspects of the network, such as access control, packet monitoring, accounting, etc.
These two policies may change dynamically as the network operates and therefore, the state of the abstract switch may also change. A key advantage of abstracting the network as one switch is that it helps relieve the burden of configuring the network. Instead of managing the network, switch by switch in the controller, the network administrators only need to specify the high-level behavior for the network while leaving the conversion of the “Big Switch” abstraction to the low-level rule placement to be done by the network operating system or network compiler.

The conversion from a high-level description of the network into a detailed rule placement on each switch is challenging because many constraints have to be satisfied to achieve a correct and efficient network implementation. These challenges are summarized below.

- The results obtained from the rule placement algorithm have to preserve the semantics of the original policies. Since the rules are prioritized, a rule of lower priority exists in the context of rules of higher priority, i.e. a packet header can match a rule only if it does not match any rules of higher priority. Thus, a lower priority rule cannot be placed without considering its relationship to higher priority rules. Thus, to maintain the priority semantics, we have to
create a dependency relationship between rules and account for this during rule placement.

- The rule placement is constrained by the rule capacity limitation in each switch. The main component inside an SDN switch (or OpenFlow supported switch) is a Ternary Content Addressable Memory (TCAM) that provides parallel packet matching. However, a TCAM is very expensive in terms of the power consumption and thus is a scarce resource for each switch. This limits the number of rules per switch. The size of the TCAM is usually 1k ~ 2k (1.5k TCAM slots in the 5406zl switch [14]).

- The rule placement algorithm may impact network traffic, and thus we need to ensure that any rule placement suitably trades off optimizing the number of rules with the possible increase in network traffic.

- There may be multiple rule placements that meet the capacity limitation of all switches. Determining an optimal solution for a given objective function involves searching a possibly large number of placement alternatives.

**Distributed Firewalls**

As part of the endpoint policy, the access control list (ACL) is critical to the security of the network. Many cloud computing providers offer a way to have user-specified distributed firewall policies, such as Google Compute Engine’s firewall policy [5]. Google Compute Engine allows users to specify one firewall policy to be associated with each instance created in the cloud and all packets flowing from and to the instance are subject to this firewall policy.

Most firewall policies can be modeled as a list of priority rules. This is compatible with the matching tables for OpenFlow switches. As with OpenFlow’s matching rules, each ACL rule is a tuple of a matching field and a binary decision field which
Table 4.2: Problem Formulation: Notation

<table>
<thead>
<tr>
<th>$N$</th>
<th>The set of switches in the network</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>Switch $i$</td>
</tr>
<tr>
<td>$C_i$</td>
<td>The capacity of switch $i$</td>
</tr>
<tr>
<td>$l_i$</td>
<td>Network entry (ingress and egress) port $i$</td>
</tr>
<tr>
<td>$P_i$</td>
<td>The set of paths originating from the network ingress $i$</td>
</tr>
<tr>
<td>$p_{i,j}$</td>
<td>$p_{i,j} \in P, p_{i,j} = {s_x, s_y\ldots}$ and it represents the set of switches on path $j$</td>
</tr>
<tr>
<td>$S_i$</td>
<td>$S_i = \bigcup_j p_{i,j}$ and it represents the set of switches reachable from $l_i$</td>
</tr>
<tr>
<td>$Q_i$</td>
<td>The policy attached to ingress $i$</td>
</tr>
<tr>
<td>$r_{i,j}$</td>
<td>$r_{i,j}$ is a single rule and $r_{i,j} \in Q_i$</td>
</tr>
<tr>
<td>$m_{i,j}$</td>
<td>The matching field of $r_{i,j}$</td>
</tr>
<tr>
<td>$d_{i,j}$</td>
<td>The decision of the rule</td>
</tr>
<tr>
<td>$t_{i,j}$</td>
<td>The priority of the rule</td>
</tr>
</tbody>
</table>

serves as the action field. The decision field has one bit to specify if this rule drops or permits the packet (this is also supported by OpenFlow).

4.2.2 Problem Definition

In this section, we are mainly concerned with how to place a distributed firewall in the network. There is some flexibility in how rules are placed in switches to accomplish a given firewall policy. Thus, this flexibility can be exploited to try and optimize the number of rules or alternate objectives while meeting the rule capacity of each switch.

The relevant notation used in the problem formulation is listed in Table 4.2.

The network $N$ is composed of a set of switches and each switch is named using its index, thus the $i^{th}$ switch is $s_i$ with capacity $C_i$. Each switch has a set of ports, and switches are connected to each other using these ports. Some of the ports may be the entry/exit points for the network and we use $l_i$ to represent the $i^{th}$ network ingress/egress port.

We assume that the routing policy is generated by some external module. The routing module may run shortest-path routing to generate the paths and it may consider certain load balancing schemes under different traffic patterns. Or it may
simply be a static routing library. The design of this module is beyond the scope of this work. We only require that the routes obtained from the routing policy are provided as an input to this problem as a set of routing paths. If \( l_i \) is an ingress port, then \( P_i \) is the set of paths originating from \( l_i \). Each path in \( P_i \) is an ordered set of switches, \( p_{i,j} \). \( S_i \) is the aggregation of all the switches in all the paths originating from port \( l_i \).

The distributed firewall policy is specified as a set of firewall policies \( \{Q_i\} \), one for each ingress port. This is similar in form to Google Compute Engine’s firewall policy [5] and Amazon’s EC2 security group [4]. Each \( Q_i \) is composed of a set of rules \( r_{i,j} \). Each rule \( r_{i,j} \) is a tuple \((m_{i,j}, d_{i,j}, t_{i,j})\), where

- \( m \) is an array of three-valued elements specifying the matching field \((\{0, 1, *\}) \) where * is a wildcard that matches both 0 and 1
- \( d \) is the action, which can be either DROP or PERMIT
- \( t \) encodes the priority of each rule

Policy rules are strictly prioritized, i.e., \( \forall i \forall j \forall k (j \neq k) : t_{i,j} \neq t_{i,k} \). If \( t_{i,j} < t_{i,k} \), rule \( r_{i,j} \) has a lower priority than \( r_{i,k} \).

The rule placement problem can be defined as follows: Given \( N, P, Q \), generate a mapping from \( r_{i,j} \) to \( S_i \), i.e., assign the rule \( r_{i,j} \) to one or more switches \( S_k, S_l, S_m, \ldots \) reachable from \( l_i \), such that the total number of rules placed in switch \( S_k \) does not exceed its capacity \( C_k \). Note that since a rule may be placed in more than one switch, different placements will lead to a different total number of rules. As we will see, this basic formulation can be augmented with alternative objective functions.

This is illustrated through a small example shown in Figure 4.8. Switches \( S_3, S_5 \) have a network egress port and all packets from the ingress \( l_1 \) are destined to either \( l_2 \) or \( l_3 \). The routing module specifies that packets have to go through \( S_1, S_2, S_3 \) to reach \( l_2 \), and through \( S_1, S_2, S_4, S_5 \) to reach \( l_3 \) (dotted lines in the figure). The policy,
Figure 4.8: Problem Illustration

$Q_1$, attached to the ingress at $l_1$ is shown in the Figure. Then, the rule placement problem is to find a valid mapping of the rules to the switches such that all packets destined to $l_2$ have to match the complete policy $Q_1$ along its path of $s_1, s_2, s_3$ without violating the capacity constraint of any of the switches. Similarly for packets destined to $l_3$. The solution for this example shows that rule $r_{1,3}$ is replicated along both paths when it is placed on both $s_3$ and $s_5$.

Ideally all ACL rules should be placed on the ingress switches. This would lead to the least network traffic. Also, if all rules are not placed on the ingress switch, then this can lead to increased TCAM space usage as the rules are possibly replicated along different paths from the ingress switch, as with $r_{1,3}$ in the example above. Since capacity constraints will likely prohibit all rules from being placed in the ingress switch, and it is difficult to manually explore the different options that satisfy the various capacity and other rule-placement constraints, there is need for an automated solution for the rule placement problem.

Another solution is to put all the firewall policies in the end hosts, meaning running software such as Linux iptables [1] or Open vSwitch [2] on the host machine’s operating system or hypervisor. However, software-based solutions are generally slower...
than TCAM-based hardware solutions ones because it is difficult for software to match thousands of rules simultaneously. Further, in a virtualized cloud environment, occupying significant system resources can degrade the tenant application’s performance.

Real-time or online placement may be needed in response to small policy changes, e.g. in response to security related updates where large latencies may not be acceptable. While it may be possible to make these changes through an ad-hoc method, it is desirable to have an analytical solution framework which allows for an incremental solution that can run in a fraction of a second to a second (depending on the magnitude of the change) and thus be able to provide solutions where a non-analytical method may not be able to. This notion of real-time changes is consistent with that used in other similar contexts, e.g. in incremental data-plane verification in SDNs using real-time Header Space Analysis \[27\] which can take up to seconds for real-time verification for data plane.

4.2.3 Solution Approach

In this work, we transform the rule placement problem into an Integer Linear Programming (ILP) problem, which satisfies the switch capacity, priority, and policy constraints and optimizes some objective function such as minimizing the total number of rules. We also formulate a satisfiability problem which meets the above constraints without the objective optimization. This satisfiability problem can be solved using SMT or Pseudo-Boolean Solvers. A flow chart for our approach is shown in Figure 4.9.

We start with an optional stage of removing the redundant rules in each policy by using techniques proposed in the previous section or [32, 33]. Then, we build the rule dependency graph (explained in Section 4.2.3) and find mergeable rules (Section 4.2.3). These steps are independent and can be done in any order. The next two steps handle the ILP formulation and solution, and finally, we add ingress tags to each rule (Section 4.2.3).
The basic variable of the ILP formulation is the binary variable, $v_{i,j,k}$, which indicates if rule $r_{i,j}$ is placed on $s_k$, where $s_k \in S_i$. $v_{i,j,k} = 1$ if and only if $r_{i,j}$ is placed on $s_k$. A rule may be placed on more than one switch, e.g. it may be placed on two different switches along two different paths in $P_i$. Note that we do not construct new rules or modify rules (besides merging as discussed below). Next, we discuss the constraints on the $v$ variables and the objective function in the ILP formulation.

**ILP Formulation**

Besides being binary (0, 1), the $v$ variables are constrained in other ways. The Rule Dependency Constraints ensure that the semantics of the prioritization in the ACL rules is obeyed during placement. The Path Dependency Constraints ensure that every DROP rule in a policy has to be placed somewhere along every path from an
ingress. The Switch Capacity Constraints, constrain the number of rules in each switch based on its capacity.

**Rule Dependency Constraint** The deployed distributed firewalls have to drop exactly the packets as specified in the given policy, i.e., the deployed policy should not drop a packet which is specified as PERMIT and it should drop every packet specified as DROP. Since the firewall has a binary action, the DROP rules are complementary to the PERMIT rules and thus, we only consider DROP rules. (An alternative formulation that considers only PERMIT rules would be similar.) Thus, the problem reduces to how to place the DROP rules on every path originating from an ingress.

To place DROP rule $r_{i,j}$ in a switch, we need to understand which other rules are affected by such a placement.

- Rules with matching fields disjoint with $r_{i,j}$ are not affected because they work on a different set of packets.

- Other DROP rules, even with an overlapping, i.e. non-disjoint, matching field are not affected because they have the same action as $r_{i,j}$. Placement of $r_{i,j}$ on a switch does not pose any constraints on where other DROP rules shall be located because it does not matter where to drop the packet as long as it is dropped.

- PERMIT rules with higher priority and an overlapping matching field are placed in the same switch as we place $r_{i,j}$. If a PERMIT rule intersects with $r_{i,j}$, then it potentially permits some packet that may also match the lower priority DROP rule $r_{i,j}$. Thus, this PERMIT rule must be placed as a higher priority rule on the same switch as $r_{i,j}$ to ensure that the permitted packets continue to be permitted and are not dropped by $r_{i,j}$.
We use a third subscript on variables $m_{i,j,k}, d_{i,j,k}, t_{i,j,k}$ to indicate which switch we actually place $r_{i,j}$. This rule dependency constraint is encoded as follows:

$$\forall i \forall k \forall u, w ( d_{i,u,k} = \text{PERMIT}, d_{i,w,k} = \text{DROP},$$

$$m_{i,u,k} \cap m_{i,w,k} \neq \emptyset, t_{i,u,k} > t_{i,w,k} ) : v_{i,u,k} \geq v_{i,w,k} \quad (4.9)$$

Since $v$ is binary, this formula ensures that if DROP rule $r_{i,w}$ is placed on switch $s_k$, PERMIT rule $r_{i,u}$ has to be placed on that switch as well.

**Path Dependency Constraint** Every DROP rule has to be placed somewhere along each path from the ingress. (We defer the discussion on redundant rules to Section 4.2.3 and assume that there are no redundant rules here.) This constraint is expressed as follows:

$$\forall r_{i,j} ( d_{i,j} = \text{DROP} ) : ( \sum_{s_i \in S_i} v_{i,j,l} ) \geq 1 \quad (4.10)$$

**Switch Capacity Constraint** Further, the number of rules placed on a switch must not exceed the number of rules the switch can hold. This constraint is specified as:

$$\forall k : ( \sum_{i,j} v_{i,j,k} ) \leq C_k \quad (4.11)$$

**Objective Function** In addition to satisfying constraints, an ILP formulation allows for optimizing an objective function. For example, we can minimize the total number of rules in the network:

$$\sum_{i,j,k} v_{i,j,k}$$

This maximizes the available rule space for the future addition of rules. Alternatively, we can optimize the location of the DROP rules such that they are placed further upstream along the path to minimize the traffic induced by the rule placement.
Then, the objective function is to minimize:

$$\sum_{i,j,k} (v_{i,j,k} \times \text{loc}(s_k, P_i))$$

where the function $\text{loc}(s_k, P_i)$ calculates the distance (number of hops) between an ingress port and the switch $s_k$ and it can be determined in the compile time.

**Identifying Ingress Policy using Tags** Since switches contain rules for different ingress port policies, $Q_i$, we need to have a mechanism to identify which policy a rule applies to. One solution is to add an ingress-policy-specific tagging field, such as VLAN tag. Whenever a packet enters the network, the VLAN field is used to indicate the ingress port where it enters the network. This tag then has to be included as a part of the matching field. The priority of rules deployed in each switch has to respect the original priority in the policy. However, the relative orders of the rules from different ingress policies does not matter because the tagging field results in a non-overlapping rule space.

**Rule Merging**

Networks often have a network-wide blacklisting policy, which specifies that no packet from any port shall go to certain places, or packets originating from certain IPs are potentially dangerous and need to be dropped. To take advantage of the fact that some rules are global to every ingress policy, we can merge certain rules across policies to further reduce the rule space required. The merging rules are merged if they are identical, i.e., the rules have the same matching field and the same action but belong to different policies. When rules are merged, the resulting tagging field is the union of the policies of the merged rules.

We now present the encoding for merging identical rules. We use the binary variable $v_{i,j}^m$ to represent a possible merged rule $i$ at switch $s_j$. The superscript
“m” means “mergeable”. This rule depends on a set of other rules from the ingress policies, $R_{m,i,j} = \{v_{a,x,j}, v_{b,y,j}, \ldots\}$. Note that we have represented the rules here by variables indicating their presence or absence. $R_{m,i,j}$ captures the set of rules from all the paths traversing switch $s_j$ that are identical except for the policy they apply to. $v_{i,j}^m$ is 1 if and only if $\forall v \in R_{i,j}^m : v = 1$. This constraint can be expressed as:

$$v_{i,j}^m \geq \left( \sum_{v \in R_{i,j}^m} v \right) - (M - 1)$$  \hspace{1cm} (4.12)

$$v_{i,j}^m \leq \frac{1}{M} \sum_{v \in R_{i,j}^m} v$$  \hspace{1cm} (4.13)

where $M = |R_{i,j}^m|$. From equation 4.12 if all of $v \in R^m$ are equal to 1, $v_{i,j}^m \geq (M) - (M - 1) = 1$, i.e., it is equal to 1. If there exists $v \in R_{i,j}^m$ such that $v \neq 1$, Equation 4.13 specifies that $v_{i,j}^m \leq \frac{v}{M} < 1$, i.e., $v_{i,j}^m = 0$.

There is a corresponding change in the rule capacity constraint and objective function. The set of rule placement variables now includes $v_{i,j}^m$, and each $v \in R_{i,j}^m$ is replaced by $v - v_{i,j}^m$. Thus, if $v_{i,j}^m$ is 0, then each $v \in R_{i,j}^m$ is reflected in the capacity constraint and the objective function as in the non-merged case. If $v_{i,j}^m$ is 1, then each $v \in R_{i,j}^m$ does not contribute to the capacity constraint and the objective function, and all $v \in R_{i,j}^m$ are effectively replaced by a single $v_{i,j}^m$.

There is one subtle issue that needs to be addressed during rule merging that has to do with rule dependencies arising from rule priorities. It is possible that there exists a circular dependency between the mergeable rules as illustrated by the following example. The core of this example is two rules, with rule 1 having higher priority than rule 2 in some policies and a lower priority than rule 2 in others. Further, there are three paths A, B, and C traversing a switch. All three paths have these two rules in their ingress port policies, one of which is a PERMIT rule $r_1$ with matching field of src:10.0.0.0/16 and dst:11.0.0.0/8. The other rule is a DROP rule $r_2$ with matching
field of src:10.0.0.0/8 and dst:11.0.0.0/16. The dependency relationship for all three paths is shown in Figure 4.10. If we decide to share the three rules, there will be a circular dependency since we have to put $r_1$ before $r_2$ for path A and B but the order is reversed for path C; otherwise, the semantics of the policies will not be obeyed. Hence, the dependency relation cannot be fulfilled if we do not break this circular dependency.

This circular dependency can be easily broken by adding dummy rules to the policy. In the previous example, we can add a dummy rule $r'_2$ to the policy of path C, which is exactly the same as $r_2$ but has a lower priority than $r_1$. Then, we mark $r_2$ as not mergeable and we merge rule $r'_2$. This technique does not change the semantics of the policies because $r'_2$ is dominated by $r_2$ and can never be matched.

**Path-Sliced Policy Rules**

Often, the routing library not only specifies all the packet routes, but also the flows or packets traversing each route. It is possible that the set of packets following a route only matches a portion of the rules of the ingress policy. Instead of placing all rules of the policy onto the route, we only need to place the rules that are overlapping with the packets and discard the non-overlapping rules. Figure 4.11 shows an example. Assume the ingress policy has three rules and there are two routes originating from...
the ingress. Packets for one route are destined to IP prefix 10.0.1.0/24 and packets for the other route are destined to prefix 10.0.2.0/24. Since packets for the dark route only match the first and the third rules, we only need to place these two rules onto the route, instead of placing all three of them. Similarly, only the second and the third rules need to be placed for the light route. This path-specific slicing of policy rules is managed by limiting the path dependency constraint (Equation 4.10) to rules that need to be placed for the path.

**Satisfiability Encoding**

If we are interested in a solution that satisfies the three sets of constraints and do not need to optimize any objective function, then this problem can be encoded as a satisfiability problem that can be solved using an SMT or Pseudo-Boolean Solver. In this formulation, $v_{i,j,k}$ is a Boolean variable or a binary integer variable depending on the constraint. This is a slight abuse of notation that simplifies the formulation.
Equation 4.9 is expressed as:

\[
\forall i \forall k \forall u, w \left( d_{i,u,k} = \text{PERMIT}, d_{i,w,k} = \text{DROP}, m_{i,u,k} \cap m_{i,w,k} \neq \emptyset, t_{i,u,k} > t_{i,w,k} \right) : \bigwedge (v_{i,w,k} \rightarrow v_{i,u,k})
\] (4.14)

Equation 4.10 is expressed as:

\[
\forall r_{i,j} (d_{i,j} = \text{DROP}) : \bigvee_{s \in S_i} v_{i,j,l}
\] (4.15)

Equation 4.11 stays the same with \( v_{i,j,k} \) being a binary integer variable. The equations 4.12 and 4.13 for merging rules reduce to the following constraint:

\[
v_{m}^{i,j} = \bigwedge_{v \in R_{i,j}} v
\] (4.16)

which specifies that \( v^{m} \) is 1 if and only if all \( v \) are 1. All constraints need to be satisfied to achieve a satisfiable solution.

**Incremental Deployment**

The network is a dynamic system and the packet routes may change from time to time, requiring our placement to quickly re-adapt to the new routing policies. The network changes include the following [24]:

1. **Ingress Policy Change:** Rules inside a policy can be modified or removed and new rules can be added to each policy.

2. **Ingress Policy Installation:** When new switches join the network, new policies may need to be placed in the network.

3. **Routing Policy Change:** New routes may be added or old routes deleted from the network.
The network update can also be at different scales:

1. **Small Scale:** Only a small part of the network is changed such as a new route is added or only a few rules for a single policy are changed.

2. **Medium Scale:** A portion of routes are updated or multiple policies need to be modified. This can happen, for example, when tenants leave or join a datacenter.

3. **Large Scale:** The majority of the network needs to be updated. This can happen in planned network upgrade or planned maintenance.

As seen in the experimental evaluation, running the ILP solver can take from a fraction of a second to a hundred seconds or so. This may be impractical for small and medium scale updates, which may require shorter latencies, e.g. for security updates needing rapid deployment. This prompts a look at alternative solutions for these cases.

Similar to the techniques used by [24], if the update scale is small, it may be possible to find a solution by modifying the existing solution using some heuristic. For example, if a new rule is added to the policy, we can try to place the rules as close to the ingress as possible. Such a simple heuristic may be enough to obtain a satisfying solution. While it may not guarantee a globally optimal one, it is a good trade-off between performance and optimality.

For medium scale updates, there may be an advantage in constructing a sub-problem that is smaller than the original problem by limiting the constraints and the objective function to the policies, paths and switches impacted by the change. Consider the case where an old route is removed and a new route added. We remove the rules corresponding to the old route and add variables corresponding to the new route. All other rule placements are fixed as in the original solution. This incremental version is then solved. This version is restrictive as it does not allow any rules
not relating to these paths to change. Thus, it is possible it may have no solution that satisfies all constraints, even though solving the problem from scratch may have resulted in a feasible solution. The hope is that if the changes are small, the solver may be able to modify the original solution to meet all constraints. Similarly, even if all constraints are met, the solution may be sub-optimal. Since in the context of the rule placement problem, the constraints are more critical and optimality is an optional desired goal, this may be less of an issue.

4.2.4 Experimental Results

We tested the effectiveness of our approach through a set of experiments using an ILP-based rule-placement tool we developed based on the flow in Figure 4.9 (The experimental evaluation of the satisfiability based formulation in Section 4.2.3 is the subject of future work.) The primary goal of these experiments was to study the scalability of the proposed technique. As there is no other approach that accomplishes rule placement with global network-level optimization across multiple paths and multiple policies, it was not possible to do a direct comparison with an alternative method.

We used CPLEX as the underlying ILP solver [3]. The benchmarks used were synthetically generated. This is standard in the networking community where proprietary network data is hard to obtain. Further, the synthetic benchmarks permit for constructing a family of benchmarks that can be used in scalability studies. The underlying network topology used was a Fat-Tree graph [7]. The policy for each network ingress was generated by ClassBench [54], a synthetic firewall benchmark generator. A randomly generated shortest-path routing was used as the routing policy. The objective function was to minimize the total number of rules inside the network. As mentioned earlier, this maximizes the slack available for adding rules in the future.
The computing equipment used in these experiments had an Intel Xeon processor (3.2 GHz) running Linux (kernel version 3.2.0).

We ran five experiments as follows to demonstrate the scalability of the approach.

• **Experiment 1:** For the first set of experiments, we fixed the topology, the routing paths, and the switch capacity while increasing the number of rules \( n \) at each ingress (Figure 4.12, Figure 4.13 and Figure 4.14). For each of these cases, the numbers of paths \( p \) in the network is 1024, and the number of rules \( n \) ranges from 20 to 110. This is based on practical sized policies presented in the literature [22]. The \( y \)-axis plots the run-time (log) for the rule placement procedure. As the policies are randomly generated, there is a variation in runtimes depending on the specific instance generated. To account for this, we ran 5 instances for each point along the \( x \)-axis (# rules in the policy) and indicate the average run-time with variation bars along the \( y \)-axis. Note that for a Fat-Tree topology, the number of switches is equal to \( 5k^2/4 \) and the number of hosts is \( k^3/4 \), where \( k \) is the number of ports per switch [7]. The three figures correspond to different network sizes specified in terms of \( k \). For each \( k \), we consider two switch capacities, \( C \), 200 and 1000. Again, this represents the possible range of interest in practical switches, e.g., there are 1.5\( k \) TCAM slots in the 5406zl switch [14]. Only a fraction of these will be available for ACL rules, with the rest used for routing.

We make the following observations. (i) For a given network size \( (k) \), the runtime is higher for the smaller switch capacity \( (C = 200) \) as the instance is more tightly constrained. (ii) The runtime generally increases with the number of rules except when the instance is over-constrained (no solution exists) with a large number of rules. The cases \( (k = 8, C = 200, r = 90, 100, 110) \), \( (k = 16, C = 200, r = 110) \), and \( (k = 32, C = 200, r = 110, \) except one data point) are infeasible and all others return the optimal solution. For the case with
k = 8, C = 200, r = 110, one data point took 26 minutes to return the optimal value and all the rest return infeasible within 20 seconds. The execution time for the infeasible cases actually decreases and there is a sudden drop when r transitions from 100 to 110 in all three figures. At this transition the problem instances become over-constrained, and the solver discovers that and returns quickly. Similarly, if the problem is under-constrained, as when the capacity of the switches is large (such as C = 1000), and it is easy to find a solution, the execution time is quite short.

• **Experiment 2:** For the second set of experiments, we fixed the topology and the total number of rules while increasing the number of paths from 256 to 2048
with a step of 256 (Figure 4.15). For this set of experiments, we used a network with $k = 8$, $r = 100$, and $C = 200, 500$. With $C = 200$, the solver returns infeasible when $p > 512$. The run times indicate both easy and difficult to prove infeasible cases. With $C = 500$, all points are feasible and the execution time is flat indicating that the number of paths is not as significant as the number of rules or the number of switches if the switches are not capacity constrained.

- **Experiment 3**: In this experiment we studied rule merging for a network with $k = 8$, $p = 1024$. The number of rules that are not mergeable is 20 while the number of mergeable rules increases from 1 to 10. The capacity of the switches grows from 65 to 75. The results are shown in Table 4.3. The columns “65-MR”, “70-MR”, and “75-MR” represent the cases when rule merging was enabled for the three different capacities and the columns without the “-MR” suffix indicate the cases when rule merging was not enabled. In each column, the left sub-column is the total number of rules installed in the network and the right sub-column reports the percentage overhead due to the rules not being able to fit on the ingress switch and the consequent duplication of rules over paths. “Inf” (infeasible) indicates that the solution was infeasible due to a capacity constraint. Let $A$ be the total number of rules on all policies. If these
rules could all fit in the ingress switches, then $A$ would be the total number of rules in the network. However, when this is not possible, as rules are spread across paths, they may need to be duplicated along different paths. Let the total number of resulting rules placed in the network be $B$ (reported in the left-subcolumn). $B$ is larger than $A$ due to rule duplication. The overhead of this duplication is given by $(B - A)/A$ which is reported as a percentage in the right-subcolumn of the table.

We make the following observations. (i) Rule merging results in several infeasible cases becoming feasible as the merged rules can meet the capacity constraints. (ii) Overall, the overhead of rule duplication is reduced by an average of 15% by rule merging. (iii) In some cases, the overhead may become negative. This reflects the savings due to merging across different policies.

- **Experiment 4:** In this experiment we study the effect of changing switch capacity. We increase the switch capacity from 50 to 1000 for a fixed network with $k = 16$, $r = 100$, and $p = 1024$ (Figure 4.16). CPLEX returns infeasible quickly for all data of $C = 50, 100$. We make the following observations: As the capacity becomes larger, the execution time rises and then dramatically decreases. The data points in the tail have a lower execution time and a very small variance. This indicates that the under-constrained and over-constrained cases are relatively easier to solve.

- **Experiment 5:** In this experiment, we studied the feasibility of adapting the solution incrementally for new policy installation. Since rule deletion is relatively easy and rule modification can be modeled as a combination of rule deletion and installation, the execution time for rule installation is of more interest to us. We first ran the algorithm using a network with $k = 16, r = 100, p = 1024, C = 500$ and determined the spare rule capacity for each switch.
Table 4.3: Capacity vs. Overhead in Rule Merging

<table>
<thead>
<tr>
<th># MR Rules</th>
<th>Capacity (with and without merging)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>65</td>
</tr>
<tr>
<td>1</td>
<td>3.8k 41%</td>
</tr>
<tr>
<td>2</td>
<td>Inf</td>
</tr>
<tr>
<td>3</td>
<td>Inf</td>
</tr>
<tr>
<td>4</td>
<td>Inf</td>
</tr>
<tr>
<td>5</td>
<td>Inf</td>
</tr>
<tr>
<td>6</td>
<td>Inf</td>
</tr>
<tr>
<td>7</td>
<td>Inf</td>
</tr>
<tr>
<td>8</td>
<td>Inf</td>
</tr>
<tr>
<td>9</td>
<td>Inf</td>
</tr>
<tr>
<td>10</td>
<td>Inf</td>
</tr>
</tbody>
</table>

Then we use this as the new capacity specification for incremental rule placement. For the experiment we considered adding 64, 128, and 256 new policies with each policy having 100 rules and a single path to place the rules for each policy. Even though it is a single path for the policy, the rule dependency and capacity constraints still need to be met. All three cases finished within 1.2 seconds. The cases with 64 and 128 returned a feasible solution, and 256 returned infeasible. For the second part of the experiment, we still use the spare rule capacity for each switch but instead of adding more policies, we modify the original policies by forcing them to be placed in fewer or more paths and this captures a routing path change. We modified 1, 16, and 32 policies and they finish in 126, 217, and 442 ms respectively. This demonstrates that small policy changes can be handled relatively quickly compared to a complete solution starting from scratch.

The ILP instances generated by these experiments could all be handled by the CPLEX solver. The total number of variables is proportional to the total number of rules and switches. The number of constraints is proportional to the number of paths, switches, and correlated with the number of rules (dependency constraints). For the case with $k = 8, r = 100, p = 1024$, we have about 290K variables and 520K
constraints. It takes about 2 minutes to return infeasible for $C = 200$ and 8 seconds to return optimal for $C = 1000$. For the case with $k = 32, r = 100, p = 1024$, we have about 500K variables and 940K constraints. It takes about half an hour for $C = 200$ and about 12 seconds for $C = 1000$. In both cases we can obtain the optimal solution. Thus, the run time is acceptable for the scale of problems of interest. Further, this long computation time is only for the initial configuration and we use the incremental deployment for all the later network changes. This only takes a fraction of a second and hence can be used in online or real-time deployment.

Further, the technique implicitly shares rules across paths by placing them at switches common to many paths whenever permitted by switch capacities. This is in contrast to other techniques which place all rules in all paths and thus end up placing $p \times r$ rules in the network [24]. These techniques have a significantly higher rule placement overhead compared to our modest overhead seen in the results of Table 4.3. For example, the total number of rules deployed for the largest overhead case ($p=1024$ and $r=25$ without rule merging enabled at row 5, column 75) is 4650, which is only 18% of $p \times r = 25k$. Also, our approach does not preclude the greedy solution, which places all the rules in the ingress switches as long as this solution is satisfiable and optimal in terms of the objective function.

In summary, the experiments demonstrate (i) scalability, as the ILP based solution can be applied to practical sized networks with acceptable run time (ii) good quality of results, as measured by the low overhead of rule duplication resulting from spreading the rules across paths.

4.2.5 Related Work & Discussion

The works that are closest to our approach are [24, 25] and [46]. Palette [25] tries to distribute a global policy to every route in the network by partitioning the whole table into small pieces and placing each piece in one switch. Their formulation of the
problem is much simpler than ours because they are limited to the same policy for every path in the network, which may not be the case in multi-tenant datacenters. Kang et al. [24] formulated the placement problem as distributing a global policy, specified as a union of one policy per route, to the network. However, their work did not leverage the fact that many rules can be merged together if they share the same origin. Further, they did not make use of the possibility of sharing the network-wide blacklisting rules. vCRIB [46] tried to balance the rule space between the physical networks and the hypervisors in the servers. One assumption they made is that they are allowed to change the packet route to achieve a feasible solution. This assumption
may not be valid because routing is based on multiple considerations, and thus may
not be permitted to change due to ACL rule placement.

SIMPLE [50] presents an SDN-based policy enforcement layer by steering traffic
between different middleboxes. However, SIMPLE can only deal with middlebox level
granularity and not rule level as in our approach. Further, it does not optimize rules
across multiple paths per policy or across policies. MERLIN [52] proposes a regular
expression based policy language and it uses a constraint solver to generate valid
policy deployment solutions based on the policy language. Although MERLIN can
optimize for bandwidth and link capacities, it does not handle switch rule capacities.

In contrast to these approaches, our solution offers the following advantages:

1. We maximally use rule sharing over multiple routes for a single policy, and over
   multiples policies.

2. Our approach can handle multiple constraints and optimize many aspects of the
   network, such as the total number of rules, in a single mathematical optimization
   framework.

3. Our approach has no false negatives, i.e., we will find a solution if one exists.

4. Our approach is flexible and can be extended to other goals, e.g. routing and
   load balancing (in the future) and other optimization criteria (e.g. distance
   from ingress, weighted placement, slack in table capacity, etc.).

5. Our solution is a good match for a multi-tenant datacenter environment. In
   a datacenter, every tenant has the view that they own their private virtual
   network but they share the underlying physical network. Therefore, in a dat-
   acenter, we have to overlay the policies of multiple virtual networks onto the
   shared physical network. Every tenant has a disjoint set of virtual ingress ports
   that map to the same set of physical ingress ports. This can be easily handled
   in our formulation. It is not clear how other approaches handle virtualization.
Also orthogonal, are other efforts that focus on easier network management through language design. The Frenetic project \cite{20} proposed a declarative network language to specify high-level primitives for the network. Pyretic \cite{45} uses the parallel and sequential composition operators to enable modular programming. Maple \cite{57} provides a centralized algorithmic policy for network programmers.

4.2.6 Concluding Remarks

In this section we formulate the ACL rule placement problem as ILP and satisfiability problems. We identify the various rule placement constraints, such as switch rule capacity, and show suitable encodings for each case. These formulations capture the sharing of rules across different paths for a given policy, and across different policies. Further, we show how the ILP formulation can be used to optimize different objective functions such as the total rule size.

We also study various aspects of the applicability of this technique to practical sized networks through an experimental evaluation. In particular, we consider scalability in the face of increasing number of rules, paths and different capacity constraints. Overall, in practice this technique is applicable to real-sized networks, e.g. it was successful in placing the rules for a network with 320 switches and a total of 70k rules in 25 seconds of computation time.

Future work in this direction should explore experimental evaluation of the Satisfiability formulation using SMT and Pseudo-Boolean solvers. In addition, more complex rule placement constraints may be explored, e.g. if the network wants to monitor certain packets, we do not want to let firewall rules block the packets before they reach the monitoring rules; otherwise, the monitoring session may produce inaccurate results.
4.3 Chapter Summary

In this chapter, we discussed many automatic firewall management problems for both single and distributed firewalls. In the first half of the chapter, we outlined a general single-firewall framework that can check firewall equivalence, firewall relationship such as inclusion and reduce firewall redundancies. The tool is capable of solving cases with thousands of firewall rules within seconds. We also proposed a solution for firewall synthesis problem, which can generate optimal firewalls based on a firewall description. However, modern QBF solvers are not fast enough to handle such large sized instance. In the second half of the chapter, we discussed how to transform a high-level distributed firewall description into a detailed firewall implementaion. Our tool is capable of handling a network with hundreds of switches within tens of minutes.
Chapter 5

In Band Network Update

In the previous chapters, we have covered how to diagnose data-plane errors and automatic management for firewalls. However, in order to apply bug fixes or send firewall rule changes to each firewall, the network administrators have to modify the data plane state. Moreover, network operators may want to adapt their routing policy because of new traffic characteristics (e.g., upon a surge in popularity of a hosted content). For example, they may want to migrate the routing protocol from IS-IS to OSPF. They may also want to re-configure their traffic engineering policy to optimize the network bandwidth. Alternatively, newer security requirements may force them to redirect all ingress traffic to middleboxes such as firewalls or intrusion detection agents. Due to the frequency of these events—more than one per day in large networks [56]—it is crucial for routing policies or firewall policies to be migrated (i) without bringing down the entire network and (ii) without impacting service availability.

Recently, there has been a lot of work on network updates in the context of both traditional networks (e.g., [56, 55]) and SDN (e.g., [43, 51, 26, 42]). However, the control packets either go through a fast and dedicated out-of-band (OOB) network, such as another VLAN based network, or go through the network the controller man-
Performing updates through an OOB network is easy because bi-directional connectivity between the controller (or the network management system) and the forwarding equipment is guaranteed at all times. However, this requires the administrator to manage an additional network. Therefore, there is significant incentive to be able to perform updates in-band. While performing OOB updates correctly is known to be hard [55], performing in-band updates correctly is even more challenging for at least two reasons. First, in-band updates require maintaining network-wide consistency in addition to bi-directional communication between the controller and any forwarding equipment at all times. Second, as control messages are sent on the production network directly, in-band updates also need to take into consideration in-path middleboxes that could drop them.

In this chapter, we present a general in-band reconfiguration framework that addresses both concerns. We make the following contributions:

1. We introduce and formally define the in-band network update problem. Our approach applies to both traditional and SDN networks, but focuses on the latter.

2. We show how we can compute a valid update ordering of the switches that can successfully migrate the network; or determine that no such solution exists. Our solution is based on the Boolean model outlined in chapter 2.

3. We show how to maintain full bi-directional connectivity between the controller and the forwarding devices.

4. We show how we minimize the time it takes to perform the update by simultaneously updating multiple devices whenever possible.

This chapter is organized as follows. Section 5.1 demonstrates the challenges of in-band updates with a simple example. Section 5.2 defines and formulates the in-band update problem. Section 5.3 describes our reconfiguration approach. Section 5.4
demonstrates the scalability and effectiveness of the approach via experimental results. Finally, Section 5.5 provides a comparison with related work and Section 5.6 some concluding remarks.

### 5.1 Motivating Example

In this section, we walk through a small example to demonstrate why in-band network update is difficult. Consider the network shown in Figure 5.1 composed of a management unit, which we call a controller, and three forwarding devices (i.e., SDN switches or IP routers). Assume this is a simple network and no hosts are attached to any ports. The traffic in the network is only destined to the switches or the controller. The controller is in charge of setting up the forwarding entries in each device. It also
initializes and manages the entire update process. Initially, the controller reaches switch B or C in one-hop, via A. Likewise, packets between B and C go through A.

Now assume the network operator adds a firewall, D, configured to act as a Deep Packet Inspection (DPI) device, between B and C (Figure 5.2) and she wants to force all network traffic to traverse the firewall. To make the forwarding action as simple as possible, the operator configures the firewall to forward all packets that arrived at port 1 to port 2 and vice versa. Another solution would be to configure the firewall to forward whatever traffic it receives to the same port it came from. This however would make forwarding decision at B and C more difficult as they would have to consider the ingress port of the packets in addition to the destination to avoid forwarding loops. After the update, device A forwards packets destined to C to port 2 and packets destined to B to port 3. B will have one entry in the routing table to route packets to C to port 2 and similarly for device C. The question is how to update the network from the initial to the final routing state, using only the in-band network. The routing state includes not only the forwarding states for normal traffic but also the control packets.

A strawman approach is to update the devices according to their relative distance to the controller. For instance, update the devices that are directly connected to the controller first, then the ones located one-hop away, etc. In this case, the controller would first update the forwarding table on A. Doing so, A will start forwarding packets destined to C to port 2 and packets destined to B to port 3. However, B and C have not been updated yet, and when B (resp. C) receives a packet destined to C (resp. B), it will send it back to A. Hence, packets will get stuck in a forwarding loop. This loop is permanent and renders B and C completely isolated from the controller. Restoring controller-to-device connectivity requires the network operators to manually reconfigure the forwarding tables on B and C (using a physical interface such as a serial port). Another solution would be to update devices located further first, and
work backwards towards the controller. Again, this strategy does not guarantee that
the controller will be able to reach all equipment. As an illustration, consider the
opposite migration in which the network is updated from the final state (Figure 5.2)
to the initial one (Figure 5.1). If B is updated first, C is no longer reachable from the
controller until A is updated. However, if we update A before B and C, the controller
can reach all devices during the entire updating process.

As we can see from this example, an ill-designed updating sequence could lead
to permanent unreachability problems. Moreover, in this example the firewall only
inspects the packets (for simplicity). In real networks, the presence of firewalls would
complicate the update order even further as they could block packets in the middle
of the migration.

### 5.2 Problem Definition

In this chapter, we define the network as a controller, a set of switches, and their
connecting links. The controller is the management unit in traditional networks that
initiates the update or the centralized controller in SDN. Similarly, both forwarding
devices and access control devices are called switches (denoted as $S_i$ with $i$ being its
unique ID) and they are connected by links. The routing behavior of each switch
is completely determined by its policy. For forwarding devices, such policy can be
extracted from the *Forwarding Information Base* in routers or *IP Routing Table* in
Linux kernel-based routing devices. For firewalls, such policy can be the ACL table.
The policy is denoted as $Q^1_i$ or $Q^2_i$. The superscript is used to indicate the version
of the policy. We assume a switch-level atomic update from $Q^1_i$ to $Q^2_i$, as supported
by modern routers [19]. The policy can be a priority based routing table as in SDN
or an IP prefix based routing table as in traditional networks as long as the policy
can determine a specific routing decision based on the incoming packets. Since the
update can be implemented round by round, we use $P_i$ to represent in which round switch $S_i$ shall be updated. For example, if $P_i = 3$, the controller should update $S_i$ in the third round. If two devices have the same value of their update indices, they are updated in the same round. A round need not be updated atomically, i.e., the switches in a round can be updated independently.

The in-band network update problem is defined as follows:

**Problem 1.** *Given two routing configurations of the network $\{Q^1_i\}$ and $\{Q^2_i\}$, find a partial order “$<$” of the switch update indices such that $\forall S_i, S_i$ is reachable from the controller using the in-band network if $\forall S_j : P_j < P_i$ have been updated from $Q^1_j$ to $Q^2_j$; or prove that no such partial order exists.*

Optionally, we can define a stronger network update problem by enforcing a two-way connectivity, such as “$\forall S_i, S_i$ is reachable from the controller for both directions during the update.” This will be useful for TCP messages.

The reason why it is a partial order is that it is possible to have two switches which do not share any switches in their update routes, meaning that their update order does not affect the reachability to each other. Therefore, we can cluster such updates together to achieve simultaneous updates when two switches have no relative ordering constraints. The specific actions needed to update from $Q^1_i$ to $Q^2_i$ can be flexible and dependent on the scenario.

### 5.3 Solution

In this section, we discuss how to solve the in-band update problem using an ILP formulation. First, we explain the reason why we formulate the problem as a constraint satisfaction formulation and more specifically an ILP problem. After that, we will provide an overview of our approach and the encoding details.
5.3.1 Intuition

Our work is based on the following observation: in order to update switch $S_i$, the controller has to establish a route destined to the switch. This route may have a set of switches $X$ that have been updated to $Q^2$ with the rest of switches $Y$ not updated. In order for this route to be successfully established, we have to update $X$ before updating $S_i$ and update $Y$ after we update $S_i$, i.e.,

$$\forall s_j \in X : P_j < P_i \text{ and } \forall s_k \in Y : P_i < P_k$$ \hspace{1cm} (5.1)

This constraint is the foundation of our work to find a valid ordering among all the switches and thus our work is based on the satisfaction of a set of order constraints. These order constraints can easily fit into a constraint satisfaction based formulation. As the $P_i$ are integer variables, and thus it is natural to consider formulating the constraint satisfaction problem as an ILP problem. Further, since $X$ and $Y$ are not known beforehand, our method has to consider all possible cases for $X$ and $Y$ and we use a symbolic encoding for the network to accomplish this. If there are firewalls in the network, our approach has to either bypass the firewalls, i.e. there is no firewall on the route, or make sure that firewalls do not block the packets. We now show how the symbolic encoding of the network and the order constraints can be modeled as linear constraints in an ILP formulation.

5.3.2 Overview

As mentioned in the previous subsection, we need to determine if we can successfully establish a valid route from the controller to the switches. The valid route finding problem is very similar to the reachability checking described in Section 3.1.2. For each $S_i$ that needs to be updated, we build one Boolean model of the entire network, which is used to find the valid route from the controller to that switch. For example,
if there are 3 switches to be updated, we conjunct 3 encodings of the entire network model together. The variables in one encoding are disjoint from the variables in the other except the update index variables $P_i$, which bridge all encodings together by forcing the global ordering constraint described in Equation 5.1.

The symbolic encoding has two parts, one for network encoding and the other for constraint encoding. The network encoding or the Boolean network model, represented as $T_i$, specifies the forwarding function of the network. The majority of the networking is the same for the one outlined in Chapter 2. The difference will be discussed in the next subsection. We can treat the controller as a special switch or we can abstract the controller as the set of ports it uses to connect to the network. The constraint encoding described in subsection 5.3.3 captures all the necessary constraints to generate the correct solution.

5.3.3 Encoding

Network Encoding

As discussed above, if we want to update $N^{\text{update}}$ switches, we need to create $N^{\text{update}}$ separate Boolean models for each update switch. Therefore, we need to add another index to all the variables to reflect this. For example, $l_{i,j,k}^{\text{in}}$ represents the variable for...
link $k$ of switch $S_j$ and for the model to find a valid update route for switch $S_i$ as shown in Figure 5.3.

In addition, this problem requires multiple versions of the matching table, we need a symbolic variable to choose which version of the matching table is in effect. Assume we are building the model to update switch $S_i$. Then, we use $v_{i,j}^{\text{version}}$ to represent the symbolic version of the matching table. We use $v_{i,j}^{\text{version}} = 0$ to encode the matching table uses the old version and $v_{i,j}^{\text{version}} = 1$ for the new version. As shown in Figure 5.3, this can be achieved by a multiplexer or mux. The functionality is $S^1_{i,j} = s_{i,j} \land \neg v_{i,j}^{\text{version}}$ and $S^2_{i,j} = s_{i,j} \land v_{i,j}^{\text{version}}$.

The outputs of the two version of matching tables are aggregated together. Since at least one of $s^1_{i,j}$ and $s^2_{i,j}$ is 0, the outputs of the matching table with $s$ variable to be 0 will be all zeros (remember that $s = 0$ will cause all outputs to be 0). The aggregation module at each output is just an OR operation.

**Constraints**

1. First, we have to ensure that exactly one egress port of the controller has to be on the route because we want the route to have only one source port. This can be expressed as $\sum_k l^{\text{out}}_{i,\text{controller},k} = 1$. An example is shown in Figure 5.4. In
this example, assume we try to find a valid route for updating switch C. The constraint is $l_{sw, out}^{C, controller, 1} = 1$.

2. Further, the controller’s ingress link shall not be on the path; otherwise, there will be a loop in the route with the controller in it. We use $\sum_k l_{in}^{controller, k} = 0$ to express that. In the previous example, the constraint is $l_{sw, in}^{C, controller, 1} = 0$.

3. At least one ingress port of the destination switch has to be on the route and this guarantees a route with valid destination: $\sum_k l_{in}^{i, i, k} > 0$. This constraint guarantees that if there are firewalls on the route, they do not block the control packets. In the example in Figure 5.4, the constraint is $l_{sw, in}^{C, C, 1} + l_{sw, in}^{C, C, 2} > 0$.

These set of constraints are also similar to the reachability checking constraints described in Section 3.1.2.

**Update Order Constraint**

Equation 5.1 has to be captured in the constraint formula and it can be easily expressed using $v_{version}$ and $s_{i,j}$. If $v_{version} = 0$ and $s_{i,j} = 1$, $S_j$ is on the route to update $S_i$ and it uses $Q_j^1$ as the forwarding policy. This means that $S_j$ is updated later than $S_i$ and thus $P_i < P_j$. Similarly, if both $v_{version}$ and $s_{i,j}$ are 1, $S_j$ is on the route and it uses $Q_j^2$ as the forwarding policy and thus $P_i > P_j$. Overall, the formula to capture the update order constraint is $(s_{i,j} \land \neg v_{version} \Rightarrow (P_i < P_j)) \land (s_{i,j} \land v_{version} \Rightarrow (P_i > P_j))$.

### 5.3.4 Optimization

Using the constraints expressed in the network encoding and the constraint encoding, we are able to find a sequence of $P_i$ values if one exists. *These values provide a valid ordering of the switch update.* As mentioned in Section 5.2, since the update order is a partial order, there may exist some update order indices that have the same value, meaning we could update those switches simultaneously. Usually, it is preferable to
update the network as fast as possible i.e., using as few update rounds as possible. However, the solution returned by the ILP solver does not reflect this. One solution is to let the ILP solver minimize $\max\{P_i\}$. We set an upper bound for each $P_i$ to $P_{upper}$ ($\forall i : P_i < P_{upper}$) and let the solver minimize the upper bound $P_{upper}$. Depending on how $P_{upper}$ is constrained, it may be possible that the solver does not return any results in a reasonable amount of time. This will likely happen when $P_{upper}$ is not too tight or too loose a constraint. The former may be easy to determine as being infeasible and the latter may be easy to satisfy. To deal with this, we can set a time limit for each run and we can regard time-outs as “Infeasible” cases. Thus, we get a result that may not be the minimum number of rounds, but possibly very close to it. We refer to this solution as the “minimal” solution within the fixed execution time budget to distinguish it from the minimum one.

5.3.5 Two-Way Communication

In the formulation mentioned in the previous subsections, we only considered the communication from the controller to the switch. In many cases, after the switch updates to its new routing policy, it may send reply messages back to the controller to mark the end of the update; otherwise, the controller will not know when the update is completed. In order to include such “two-way” communication, we can add another version of the symbolic encoding for each update switch and this encoding tries to find a valid route from the update switches to the controller. Almost everything is the same as the previous network encoding except that we need to reverse the constraint on the ingress and egress links of the controller and the update switch. We specify that at least one of the egress links of the update switch and exactly one of the controller’s ingress links are on the route.
5.4 Experimental Results

We implemented our ILP based approach using IBM CPLEX as the underlying ILP solver with at most 7 threads enabled. We ran a variety of experiments to test the effectiveness of our approach. We tested our tool on a set of synthetic benchmarks by using the Waxman topology as the network topology and we randomly connect the controller to two switches. We used the shortest path algorithm to generate different versions of the routing policies. The machines used in these experiments have Intel Xeon 3.20 GHz processors running Linux with kernel version 3.2.0. Our tool successfully finds valid orderings for all the experiments we ran.

We increased the size of the networks from 10 switches to 100 switches with an increment of 10 switches with about 50% of the switches needing to be updated. We ran the experiments twice without optimizing for $P^{upper}$. One enables the two-way communication from the update switch to the controller and the other without it. The execution time is shown in Figure 5.5 and the total number of indices required for such an update to finish is shown in Figure 5.6. The total number of indices represents the number of rounds needed for the update. As we can see from Figure 5.5, the execution time increases as the total number of switches grows. This is expected as the size of
Figure 5.6: # Switches vs. # Indices

Figure 5.7: Percent of Updated Switches vs. Time

Figure 5.8: Percent of Updated Switches vs. # Indices
the formula is linear in the size of the network and the total number of switches that need to be updated. The requirement of two-way communication also increases the execution time because the search space is more constrained. On average, the two-way communication increases the execution time by about 2.8 times. In Figure 5.6, “Opt” presents the minimal \( P_{\text{upper}} \) achieved by setting the time limit to be 200s. The cases without the “Opt” flag represents the total number of update rounds returned. Since we only constrain the relative relationship between \( P_i \), there are no constraints on the specific values of \( P_i \). In this case, we count how many unique \( P_i \) there are. As we can see, the total number of update rounds increases with the size of the network and the optimization on \( P_{\text{upper}} \) is effective in reducing the total update rounds. On average, it reduces the number of update rounds by 62% (compare the 1st bar with the second and the third with the fourth in Figure 5.6).

In the second set of experiments, we increase the percentage of the switches in the network needing to be updated on two networks, one with 50 switches and the other with 80 switches. The results are shown in Figures 5.7 and 5.8. Both execution time and the total number of update indices increase with the percentage of the updated switches. We set the time limit for the execution to be 200s. The optimization on \( P_{\text{upper}} \) saves an average of 57% of update rounds for the network with 50 switches and 65% for the case with 80 switches.

5.5 Related Work & Discussion

[55] proposes a seamless network update strategy using a different ILP based formulation, which enumerates all the forwarding loops in the network and breaks loops by forcing certain update orders. However, they can only handle forwarding loop based reachability problems and cannot handle packet blocking or parallel updates. Recently, there have been many other works on updating the network using SDN.
proposed per-packet consistency which requires that every packet in the network is processed by either the pre-update configuration or the post-update configuration, and never a mixture of the two. Based on per-packet consistency, they proposed a two-phase update that includes a version number in each rule and packet. Whenever an update is scheduled, they first install rules with a newer version number in the middle of the network and then update the new rules at the border of the network. Similarly, [42] proposes an OpenFlow safe update protocol to reduce the rule space requirement of two-phase update by temporarily forwarding packets to the controller in the middle of the update. [26] improves two-phase commit by increasing the number of rounds to complete an entire update. They partition the network into different slices and each slice observes similar network behavior. They update the network one slice at a time. zUpdate [35] is used to update the network without transient violation of the bandwidth requirements on each link. This approach computes how to update one network traffic pattern to another pattern using a sequence of updates. Unlike our approach, these works do not target in-band updates and cannot deal with in-band middleboxes. Our framework is also able to leverage partial ordering constraint to update the network in parallel whenever possible. Meta-Management System [38] uses packet broadcasting to update different network components. This method is valid in traditional computing environment but not in embedded computing. If each switch is a battery-constrained device, broadcasting may consume too much power. In comparison, our method does not waste much traffic in addition to the update itself.

5.6 Chapter Summary

In this chapter, we proposed a general network routing policy update mechanism using in-band networks. In-band networks do not require dedicated links from the
controller to the forwarding devices. Our ILP-based symbolic representation of the network model can model the network routing behavior and can handle both normal forwarding devices and access control devices. By constraining the symbolic model, we can find an update order that guarantees valid routes from the controller to all the switches that need to be updated. Based on the route found, we can generate a correct ordering of the switches in which an update can be successfully implemented.

We demonstrated that our approach is scalable for real-world networks as it can find a correct ordering within 10 seconds for networks with less than 70 switches (Figure 5.5). Moreover, our approach can minimize the update rounds by enabling simultaneous updates for switches and it can reduce the total number of the update rounds by around 60%. One of the future directions is to handle networks with additional kinds of middleboxes such as Network Address Translators. It is also interesting to extend our framework to maintain not only the connectivity from the controller to the updated switches, but also from end-points to end-points to improve service availability.
Chapter 6

Conclusion

6.1 Thesis Contributions

This thesis proposes a novel Boolean model for network data plane. This Boolean modeling technique couples a packet’s temporal information and spatial information together to reduce the size and complexity of the network models. This approach can support many packet processing actions, including forwarding to incoming ports, broadcasting, packet encapsulation/decapsulation, and any arbitrary propositional logic modification on the packet header. The network model is also capable of handling a variety of network management tasks. It can check many properties, which require only one packet trace to prove the property is violated. We call those properties single-packet-trace properties. The model can check many single-packet-trace properties, including forwarding loop checking, reachability checking, slice isolation, and even complete network equivalence checking. We present a comparison between our method and Header Space Analysis on Fat-Tree topology for reachability checking and forwarding loop freedom checking. Our tool is consistently 2 to 3 orders of magnitude faster than Header Space Analysis.
In addition to the data plane property checking, our model can solve many firewall management problems. We show how to use our Boolean model for single firewall comparison to check the equivalence between two firewalls. Based on the equivalence checking, we can detect which rules are redundant and can optimize the matching table of firewalls. Built upon the Boolean model, our Quantified Boolean Formula based solution can synthesize optimal firewalls although current QBF solvers are not powerful enough for decent firewall size. For distributed firewall management, we can convert a high-level firewall behavior description to the detailed firewall implementation and we formulate the ACL placement problem as an ILP problem. Our formulation can support several rule placement constraints, such as switch rule capacity. Through experimental evaluation, we demonstrated that our tool can handle real-world sized problem.

In the end, based on the Boolean network model, we also proposed an in-band network update mechanism that can apply network changes or network migration without using dedicated links/networks or additional routing rules for control packets. The difficulty of the problem is to figure out a correct update ordering of all the switches such that the early updates on the routing tables do not lead to unreacheability from the controller to the switches that have not been updated. Therefore, in order for an update packet to reach a destination, we have to maintain a valid reachability from the controller to the destination switch when it is the time to update the destination switch. This problem can be modeled as a combination of multiple small path finding or reachability checking problems and we can enforce a global switch update order when we encode for the reachability problem. Our approach is scalable for real-world networks. It can find a valid update order for a network with 70 switches within 10 seconds.
6.2 Future Research Possibilities

This thesis mainly focuses on how to build a Boolean model to solve networking problems. This work can be extended in many ways. One direction is how to reformulate the problems so that we can speed up the solution time. Another direction is to explore the opportunity to extend the Boolean model so that it can handle other types of problems. Next, we explore possible extensions in each of these categories.

6.2.1 Performance

Although our Boolean model can handle real-world sized problems, it is far away real-time analysis. The speed of the tool is important to network management because slow management can affect the performance and security of the entire network. For example, a faster data plane debugger can find data plane bugs quicker and thus, it reduces the duration of the bugs. There are many ways to speed up our Boolean model based framework.

- Packet Space Partition: The packet space and the total number of packet locations are huge. One way of speeding up solving a problem with such a large space is to divide and conquer. We can partition a large problem into several smaller individual problems so that we can solve the smaller problems in parallel. For example, if a network does not modify the packet header in any way, we can partition the entire packet space into smaller subspace and we can build a Boolean model for each subspace. For example, we can partition the entire packet space into two spaces: one with the first packet bit being 1 and the other with the first bit being 0. Then, we can create two separate models for each case. Any rules overlapping with the first case belong to the model created for the first case and vice versa. Since the new problems may be much smaller,
we can solve them faster and in parallel and hence, we can speed up the entire process.

- Network Location Partition: another way to partition the problem is to divide the entire network topology into smaller sub-graphs. For each sub-graph, we can use a Big Switch Abstraction described in Section 4.2 to replace the set of switches in the sub-graph to construct a new abstracted topology. Then, we can transform the problem on the original topology into the problem on a set of sub-graphs and the abstracted topology.

- Incremental Solving: a network is a dynamic system and the network state only changes a small part at a time. If a network state changes, it is time-consuming to re-run the management tool on the entire network state. Another solution is to run the tool only on the state that gets modified so that we can manage the network incrementally. Chapter 4 briefly mentions how to incrementally deploy firewall placement based on incremental changes but the solution is not general enough to handle other network problems. Many network state changes are matching rule deletion, addition, and modification, which are reflected as Boolean formula changes and modification. There are many incremental Boolean Satisfiability solvers that support a faster runtime on the incremental changes of the formula [17]. It would be interesting to see how this can speed up the problem on the problem whose state can dynamically but incrementally change.

### 6.2.2 Other Applications

Another direction to extend the work is to make the Boolean model even more powerful. One aspect that is missing in this work is handling quantitative network resources, such as network bandwidth. These are important to networks as they affect
the performance of the network. We can extend the Boolean model to support those quantitative features. For example, we can use integer variables to represent the bandwidth allocated for certain links or switches and use additional ILP constraints to encode bandwidth requirements, such as to guarantee that the total bandwidth allocated to certain packets does not exceed certain amount.

Further, we have proposed the solution of synthesizing optimal firewalls. It would be interesting to study whether we can solve the synthesis problem for the entire network. For example, given a high-level behavior description of the network, we generate an optimal routing rule and firewall rule placement to satisfy the high-level description.
Bibliography


