Efficient Rules Management Algorithms in Software Defined Networking

Ahmad Abboud

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Efficient Rules Management Algorithms in Software Defined Networking

Algorithmes efficaces de gestion des règles dans les réseaux définis par logiciel

THÈSE

présentée et soutenue publiquement le 09 Décembre 2021

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Ahmad ABBOUD

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Acknowledgments

First and foremost, I am extremely grateful to my supervisors, Dr. Abdelkader Lahmadi, Dr Michaël Rusinowitch and Dr. Adel Bouhoula, for their invaluable advice, continuous support and patience during my PhD study. Their immense knowledge and plentiful experience have encouraged me in all the time of my academic research.

I would like to express gratitude to all my colleagues at Numeryx Technologies and special thanks to Dr. Mondher Ayadi for this beautiful experience and for allowing me to conduct this thesis.

I would like to thank all members of RESIST research team in Inria Nancy for the support you have offered me throughout my journey.

It is my privilege also to thank Dr. Miguel Couceiro, Rémi Garcia, and Dr. Nizar Ben Néji for their kind help and insightful comments on my research.

My appreciation also goes out to all my friends for their unwavering support during the past three years.

And my biggest thanks go to my family, my mother Zeinab, and my sisters Rim, Saba, Layan, and Assan, for their encouragement and support throughout my studies.

For My Father,
May he rest in peace.

Résumé

Au sein des réseaux définis par logiciel (SDN), les exigences de filtrage pour les applications critiques varient souvent en fonction des changements de flux et des politiques de sécurité. SDN résout ce problème avec une abstraction logicielle flexible, permettant la modification et la mise en œuvre simultanées et pratiques d'une politique réseau sur les routeurs.

Avec l'augmentation du nombre de règles de filtrage et la taille des données qui traversent le réseau chaque seconde, il est crucial de minimiser le nombre d'entrées et d'accélérer le processus de recherche. D'autre part, l'accroissement du nombre d'attaques sur Internet s'accompagne d'une augmentation de la taille des listes noires et du nombre de règles des pare-feux. Leur capacité de stockage limitée nécessite une gestion efficace de l'espace.

Dans la première partie de cette thèse, nous proposons une représentation compacte des règles de filtrage tout en préservant leur sémantique. La construction de cette représentation est obtenue par des algorithmes raisonnablement efficaces. Cette technique permet flexibilité et efficacité dans le déploiement des politiques de sécurité puisque les règles engendrées sont plus faciles à gérer.

Des approches complémentaires à la compression de règles consistent à décomposer et répartir les tables de règles, pour implémenter, par exemple, des politiques de contrôle d'accès distribué. Cependant, la plupart d'entre elles nécessitent une réplication importante de règles, voire la modification des en-têtes de paquets. La deuxième partie de cette thèse présente de nouvelles techniques pour décomposer et distribuer des ensembles de règles de filtrage sur une topologie de réseau donnée. Nous introduisons également une stratégie de mise à jour pour gérer les changements de politique et de topologie du réseau. De plus, nous exploitons également la structure de graphe série-parallèle pour résoudre efficacement le problème de placement de règles.

Mots-clés: Minimisation des paquets, Filtrage des paquets, Réseaux définis par logiciel, Placement des règles.

Abstract

In software-defined networks (SDN), the filtering requirements for critical applications often vary according to flow changes and security policies. SDN addresses this issue with a flexible software abstraction, allowing simultaneous and convenient modification and implementation of a network policy on flow-based switches.

With the increase in the number of entries in the ruleset and the size of data that traverses the network each second, it remains crucial to minimize the number of entries and accelerate the lookup process. On the other hand, attacks on Internet have reached a high level. The number keeps increasing, which increases the size of blacklists and the number of rules in firewalls. The limited storage capacity requires efficient management of that space.

In the first part of this thesis, our primary goal is to find a simple representation of filtering rules that enables more compact rule tables and thus is easier to manage while keeping their semantics unchanged. The construction of rules should be obtained with reasonably efficient algorithms too. This new representation can add flexibility and efficiency in deploying security policies since the generated rules are easier to manage.

A complementary approach to rule compression would be to use multiple smaller switch tables to enforce access-control policies in the network. However, most of them have a significant rules replication, or even they modify the packet's header to avoid matching a rule by a packet in the next switch. The second part of this thesis introduces new techniques to decompose and distribute filtering rule sets over a given network topology. We also introduce an update strategy to handle the changes in network policy and topology. In addition, we also exploit the structure of a series-parallel graph to efficiently resolve the rule placement problem for all-sized networks intractable time.

Keywords: Packet minimization, Packet filtering, Software defined networks, Rule placement

Contents

Chapter 1

General Introduction

Contents

1.1 Context

The increasing size and complexity of network topologies make it challenging to manage network devices. In 2021, for example, the number of estimated IoT devices embedded with sensors and software to share data over the Internet is around 13.8 billion worldwide. This number will reach 30.9 billion in just four years from now [1]. This enormous number of connected devices generate over 2.5 quintillion bytes of data every day [2], and by 2025, we create 463 exabytes of data each day [3]. In addition, the type of data transferred over the Internet is significant, and any delay or security breach can cost millions of dollars in loss. TABB Group study shows that if an electronic broker platform has a delay of 5 milliseconds, it could lose at least 4 million dollars per millisecond [4]. In another case, a breach in Yahoo costs the company around $4.48 billion [5], while the loss resulting from another hacking breach is estimated to have cost the same company $444 billion [6]. This shows how severe and costly an attack can be and triggers the scientific community to find new solutions to adapt to the new challenges.

In the last decade, a new networking trend called Software Defined Network (SDN) is developed to enhance the manageability of networks. SDN decouples the control plane and the data plane that communicate with each other by OpenFlow protocol. Thanks to this decoupling network devices are reduced to simple forwarding devices, leading to a faster packet classification. At the same time, dataflow control tasks are assigned to an entity called the controller. This new approach gives the administrator a global view of the network and simplify updating and problem resolution, therefore improving the performance of SDN-enabled devices and controllers.

Moreover, the growth of traffic impacts the size of forwarding tables which are the core OpenFlow-based software-defined networking switches. These tables aim to route any packet to the right destination. Blacklists too, may have to store millions of IP addresses of malicious sources. Some rulesets admit more than 250000 instances corresponding to malicious domains [7] and more than 5000 new malicious IP addresses are discovered daily. On August 8th, 2014, some Internet service providers (ISP) experienced router crashes causing network outages around the globe. This problem arised after the number of entries in a global BGP routing table stretched to more than 512000 entries, leading to a problem in routers with memory up to 512K entries [8]. While 512k seems large, the active BGP entries nowadays are about 900k and keep increasing with time as shown in Fig. 1.1.

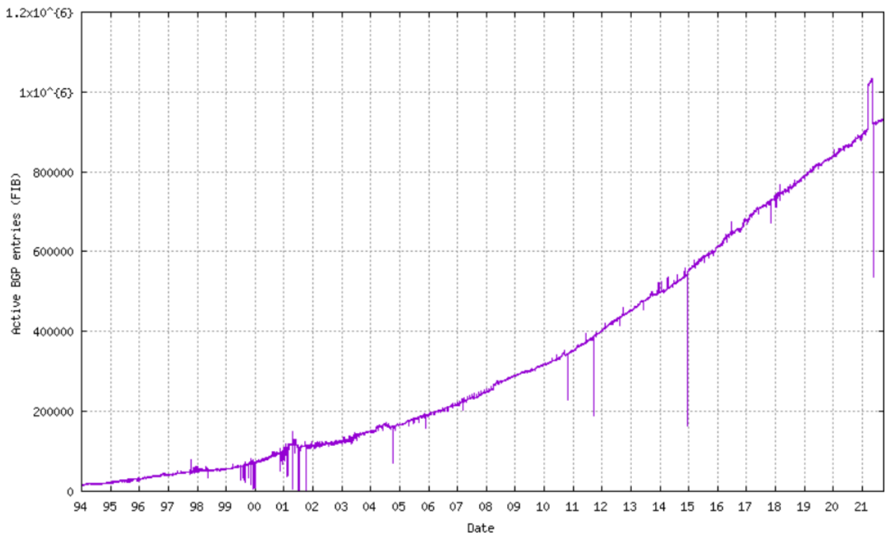


Figure 1.1: Active BGP entries [9].

This growing number of entries in networking devices can create bottlenecks and affect the performance and quality of service (QoS). In addition, with a higher number of entries, we have a higher probability of errors and misconfiguration. According to [10] 20% of all failures in a network can be attributed to poorly planned and configured updates in the network. Managing these large rulesets and data flow is complex. While automatization and artificial intelligence are being deployed in many applications nowadays, many networking systems still rely on the administrator for deployment and configuration of the network, therefore adding more delays and source of errors and interruptions in the network.

Reducing rulesets size is a challenging problem for networking devices and their associated packet classification solutions. Moreover, the placement of any rule entry in the network based on its priority is essential to ensure that every harmless flow can reach the correct destination while malicious ones are blocked. In this thesis, our primary focus will be to address these two problems.

1.2 Problem Statement

With the trend of continue growing of number of rules in ruleset, it becomes a great challenge their management regarding multiple issues like rule shadowing, rule redundancy, inconsistencies, . . . Besides trying to solve these problems directly, one may attempt to mitigate them by relying on rules compression in order to reduce the size of rulesets. CIDR notation is not efficient when dealing with nowadays rulesets especially when the number of exceptions is high. Therefore we need a new notation that can handle a large number of such rules. However, deriving a set of rules semantically equivalent to the initial one while being space saving, is not obvious. Therefore the first objective of this thesis is to minimize rulesets by designing a succinct notation to compress large sets of IP addresses. This notation needs to be generalized to work with any range type, IP, or port range. In addition, we show that the translation between the CIDR representation and the novel one is fast and linear. In some cases, the size of networking switches is too limited to fit all rules even after compression. In some other cases, the type of application requires rules set to be decomposed and distributed over the network instead of storing them in the same location. With the large size of the nowadays networks and their complex topologies, distributing rules is a challenge. For example, packets can traverse multiple paths and need to be rechecked with the same rules in every path. In other scenarios, a switch belonging to several paths needs to maintain specific rules for each one, making it difficult to share the switch capacity between paths.
Most decomposing rules techniques use additional rules in switches to maintain the same semantics. For example, let us consider a switch with all rulesets installed in that switch. We have two scenarios, the packet matches a rule, or no match is found. If we need to decompose the same ruleset on multiple switches, the same packet must match the same rule

in a switch then traverse all following switches without any match in the first scenario. However, in the second case, the packet must always match the default rule in all switches.

Existing decomposition techniques rely mainly on rule duplication and generate new rules to preserve the semantics of the rulesets. However, adding additional rules increases the overhead of switches and affects the performance of the packet filtering and classification process. To overcome this problem, we need to find a way to minimize the number of new rules generated. Moreover, the complexity of network topologies makes the distribution of rules complicated. In particular, intersection nodes, switch capacities, processing power, etc., need to be considered in any distribution strategy. Therefore, the second objective of this thesis is mainly concerned with decomposing and distributing a set of rules in an SDN network while preserving the rulesets' general semantics and respecting each switch's capacity constraint.

1.3 Our Contribution

Packet classification is achievable with software or hardware-based solutions. Devices that use TCAM memories are standard for packet classification due to their faster lookup time. However, these devices have a limited space and require significant amounts of power. On the other hand, software-based applications like a decision tree are easier to implement but slower than TCAMs. Besides, to decrease the classification time, the number of rules in the tables is reduced. This will decrease the power consumption and make searching a data structure more efficient, resulting in a better classification process.
This thesis provides two major contributions in rules management. The first one is an efficient method to reduce the number of entries in tables by using a new range encoding technique extending the standard CIDR notation. The second contribution provides algorithms for distributing rules over an SDN network while reducing the overhead in switches. Finally, we have developed an efficient rules update strategy to handle the continuous changes in network topologies.

1.4 Thesis Organization

The remainder of this thesis is structured as follows. Chapter 2 details the state of the art regarding the rule minimization problem and we discuss the limitations of existing approaches. Moreover, we present the existing research body for the decomposition and distribution of rules in SDN and update strategy approaches to reduce the overall update time and the

overhead in switches.
Chapter 3 introduces a new representation for field ranges called *Double Mask*. This strategy aims to reduce the number of entries in rulesets by targeting a larger number of addresses with a smaller set of rules, thus resolving the rule expansion problem, especially in port ranges. A linear-time algorithm is designed to compute the new notation by transferring the old CIDR notations to new ones or directly generating a list of new entries. Moreover, we implement a solution in SDN by adding support for the representation inside the OpenFlow protocol. This enables sending and receiving messages between the controller and switches that contain rules with *Double Mask*.
In Chapter 4, we introduce our rule decomposition approach by using the Longest Prefix Match (LPM) strategy. Indeed, to reduce the overhead, we tried to minimize the number of forward rules generated by considering the action field of a rule.
In Chapter 5, we develop an algorithm for the decomposition and distribution of rules on series-parallel graphs, and we generalize the distribution to all st-dags graphs. Another proposed approach called Two-Tier, eliminates the need for forward rules, and therefore reduce the overhead of switches. Next, we evaluate our algorithms and compare their results with exiting approaches available in the literature. Finally, we introduce an update strategy that focus only on the part of the network that can be potentially affected by the modification. Our strategy amounts to apply the above decomposition and distribution algorithm with a subset of rules and a (generally smaller) subnetwork. This leads to a faster solution than with the naive approach as shown by our carried simulations.
In Chapter 6 we present the overall conclusions of our contributions. We also discuss some perspectives on how these approaches can be enhanced and extended.

1.5 Publications

The contributions of this thesis have been published in a number of international conferences as detailed below:

- Ahmad Abboud, Rémi Garcia, Abdelkader Lahmadi, Michaël Rusinowitch, and Adel Bouhoula. "Efficient Distribution of Security Policy Filtering Rules in Software Defined Networks." In 2020 IEEE 19th International Symposium on Network Computing and Applications (NCA), pp. 1-10. IEEE, 2020. [11]

- Ahmad Abboud, Rémi Garcia, Abdelkader Lahmadi, Michaël Rusinowitch, and Adel Bouhoula. "R2-D2: Filter Rule set Decomposition and Distribution in Software Defined Networks." In 2020 16th International Conference on Network and Service Management (CNSM), pp. 1-4. IEEE, 2020. [12]

- Ahmad Abboud, Abdelkader Lahmadi, Michaël Rusinowitch, Miguel Couceiro, Adel Bouhoula, and Mondher Avadi. "Double mask: An efficient rule encoding for software defined networking." In 2020 23rd Conference on Innovation in Clouds, Internet and Networks (ICIN), pp. 186-193. IEEE, 2020. [13]

- Ahmad Abboud, Abdelkader Lahmadi, Michaël Rusinowitch, Miguel Couceiro, and Adel Bouhoula. "Poster: Minimizing range rules for packet filtering using a double mask representation." In 2019 IFIP Networking Conference (IFIP Networking), pp. 1-2. IEEE, 2019. [14]

Chapter 2

Related work

Contents

2.1 Introduction

The increasing size of routing tables, blacklists, intrusion detection systems, etc., raises the question of representing a set of addresses by the smallest ruleset. For many decades, the Classless Inter-Domain Routing (CIDR) notation has been used to partially resole this problem by grouping IP addresses from the same network into one representation. However, managing large rulesets in complex topologies is a non-trivial task. We need to consider the limitations in memory capacity and process power as well as the location of each switch in the network.

The categorization and grouping of a data flow require to classify each packet in the network. This process is called packet classification, and all packets belonging to the same flow are processed in the same way [15]. This technique enables routing, filtering, monitoring, and many other services on the Internet [16]. Securing a flow of data means testing every packet traversing the network. Unfortunately, this process adds a delay each time a packet arrives, especially if the number of entries is large,and can even block harmless flows if the system is poorly configured. Hardware-based packet classification is the fastest solution for this problem. TCAMs, for example, are becoming the standard in industry due to their high-speed classification capabilities. TCAMs permit a fast parallel search of the memory but admit memory limitations, high power consumption, and costly memory architecture modification. Software solutions, on the other hand, are slower but their implementation, updating, and scaling are much more manageable.

In this thesis we tackle the enhancement of packet classification process by considering the minimization of rulesets and their distribution among network devices with limited memory.

The rest of this chapter proceeds as follows. Section 2.2 introduces multiple hardware and software approaches to handle the packet classification problem. In Section 2.3 we describe two software-based solutions for minimizing the number of entries. Section 2.4 provides an overview of the Software Defined Network. In Section 2.5 we investigate the methods used in the rule placement problem. Finally, in Section 2.6 we analyze and discuss the different proposed approaches for both rule minimization and rule placement problems.

2.2 Packet Classification Approaches

CIDR [17] notation is applied to all network devices and appliances, in blacklists, whitelists, routing table, firewalls, access-control list (ACL), etc. An example of an ACL rule is the following:

$$access-list\ permit\ udp\ \mathrm{192.168.32.0}\ \mathrm{0.0.7.255}\ \ \mathrm{192.168.77.3}\ \mathrm{0.0.0.255}$$

According to this rule, we permit any UDP packet from an IP source 192.168.32.0 with a wild card equal to 0.0.7.255 and a destination IP address equal to 192.168.77.3 with a wild card 0.0.0.255.

With a packet classification process, network nodes must classify individual packets traversing the node by matching such rules stored in the tables of networking devices. Packet classification requires searching the tables and apply the action associated with the highest priority rule which matches the packet. When the number of entries increase the packet classification time also increases. To solve this problem, we need to minimize the number of entries in the rulesets, leading to better performance.

Priority	IP Src	IP dst	Src Port	Src Port	Action
4	192.168.100.0/24	19.153.234.163/32	[1,65534]	[1,65534]	Accept
3	201.162.23.191/32	10.133.202.145/32	[1,65534]	[1,65534]	Deny
2	184.166.27.94/31	98.85.58.204/31	[1,65534]	[1,65534]	Deny
1	192.168.100.2/30	19.153.234.0/24	[1,65534]	[1,65534]	Deny
0	21.79.252.56/30	49.50.220.0/22	[1,65534]	[1,65534]	Accept

Table 2.1: Example of an ACL rule table.

Table 2.1 shows a rule table containing 5 ACL rules. Each rule has a priority between 0 and 4, a source and destination address, and an action field. A packet with an IP source 192.168.100.1 and an IP destination 19.153.234.163, matches two rules, one with priority 4 and the other with priority 1. In this case, the action of rule with the highest priority, accept in this case, is applied.

Packet classification often relies on multiple dimensions which is complex if the number of rules is large and introduces a performance bottleneck. Many attempts to accelerate the classification process have been developed over the years. For example, this process can be improved by decreasing the lookup time for a packet inside a rule table or minimizing the number of entries in a rule table either by re-encoding rules or reducing their size. The type of memory used to store rules can also affect the classification process.

2.2.1 A Hardware Based Solution: TCAM

Ternary Content Addressable Memories (TCAM) are the most popular hardware-based technique in networking devices. TCAMs store rules as a W-bit field (value, bitmask). For example, if W=4, a prefix 01** will be stored as (0100,1100) inside TCAM memory. An input key is compared with all stored patterns simultaneously. A TCAM gives us a result whenever a key matches a pattern or not in one cycle. At the same time, other types of memory like Static Random-Access Memory (SRAM) needs more cycles resulting in slower network switching applications.

Rule	Address	Port
1	10**	a
2	010*	b
3	01**	c
4	1000	d

Table 2.2: Example of a routing table.

In Table 2.2 we have four addresses, each one with a different port. Based on the address, each of the packets will be directed to one of the four ports. In standard cases, and without TCAMs, if a packet has an address of 0101, according to Table 2.2 Rules 2 and 3 will be triggered. To solve this problem, standard memory like SRAM uses Longest Prefix

Match (LPM), where a packet is matched by the most specific entry in the rule table. In this case, Rule 2 is more specific, and the packet will be sent to port b.

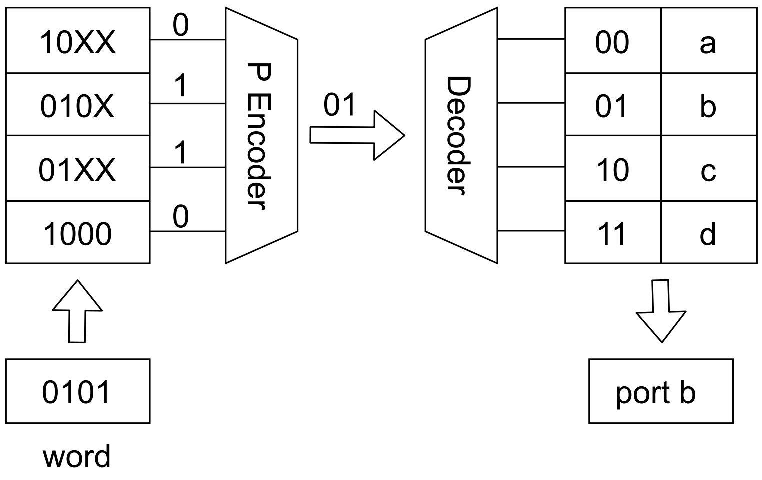


Figure 2.1: TCAM with a priority encoder.

Fig. 2.1 shows a TCAM with the same rules from Table 2.2. In this scenario, we will use a priority encoder instead of LPM. If we need to match the input key 0101, the TCAM tries to find a pattern that matches the key. A vector of N-bit, where N is the number of rows in the TCAM, is generated to indicate which rules match the input key. In case of Fig. 2.1, this vector is equal to $\{0, 1, 1, 0\}$. Then, the vector is sent to a priority encoder which indicates the address of the rule with the highest priority in the memory. The address is then sent to a decoder that matches the address with the list of ports and chooses the right one. In our example, Port b is selected. An encoder can also apply LPM to find the best match.

SRAM and TCAM have the same output, but the set of addresses is divided into multiple parts in SRAM. Each of them takes a cycle to be searched, while in TCAM, we can explore all entries in parallel.

For a faster classification and lookup process in TCAM, we can reduce the number of entries that need to be checked instead of running a parallel search on all entries. This can be done by having multipl blocks inside TCAM, and each one has a set of rules. A key represents each block (i.e., the common prefix of all rules inside this block, for example). Therefore, a packet that matches a key k will be checked against all rules inside the block represented by k.

As mentioned above, TCAMs offer a full parallel search of the memory which makes it very fast but has also some drawbacks. The explosion in the number of TCAMs entries is a well-known problem [18; 19; 20].

According to [21] the maximum number of entries needed to represent an interval of w-bits binary numbers is $2(w-1)$ prefixes for a w-bit phrase. For example, in Table 2.1, the sources and destinations ports [1,65534] need each one 30 prefixes to be represented. This means that we need 30 x 30 = 900 entries to represent all port combinations for a single rule only. With the limitation in space, it becomes worse and limits the scalability of such a technique.

Power consumption is another problem faced by TCAMs. Multiple research attempts have been proposed to solve this problem [22; 23; 24; 25; 26]. According to [16], with the same number of memory access, TCAMs consume 30 times more power than a standard SRAM. In some high-end routers, 30 to 40 percent of all power is consumed by TCAMs [19]. According to [23], a TCAM consumes 100 times more power per bit than a standard SRAM. A power-hungry device increases the heat generated by such a device leading to a degradation in performance and a delay in the classification and lookup process. In [27], the authors show that the energy per access scales linearly with the number of entries in TCAMs.

It is worthy to note that TCAMs memories are expensive. According to [28], TCAMs are 400 times more expensive than standard SRAMs with the same capacity. With the high number of rules that need to be installed in TCAMs, managing that space is a crucial element. The high-cost of memory leads to TCAMs with smaller capacity. For example, an HP ProCurve 5406zl TCAM switch can support up to 1500 OpenFlow rules, and with an average of 10 rules per active host, this switch will support 150 users only [29]. In addition to the space limitation and cost, increasing the space means increasing the number of entries in TCAMs, which reflects on the power consumption and the generated heat, leading to more expensive cooling systems [19].

In addition to the high cost and power-hungry issues, TCAMs are facing their low capacity problem. The largest TCAM chip available has a 72Mb capacity [30], while 1Mb and 2Mb are the most popular ones [31]. According to [32], a 2Mb TCAM can store around 33000 entries of size 60 bit, while the number of entries with 16 bits can reach 125000. This increase in the number of entries can save the cost of buying high-end switches with more significant TCAM memory capacity.

As mentioned before, the worst case for rule expansion is 900 per rule. Since each TCAM entry needs 144 bits to be represented, the worst-case scenario requires 144*900 = 129600 bits or 0.1296Mb in TCAM memory. If all rules have a worst case, the largest TCAM memory can store 555 rules only. While the worst case is unlikely to happen, this is still an alarming issue [31]. This is where range re-encoding and classifier minimization can help to reduce the cost of TCAMs, as shown in the next part.

In addition to memory limitation, TCAMs are not expandable and need to be changed altogether, making it difficult and costly for ISPs to handle the significant growth of TCAMs entries every day.

To overcome TCAMs problems, we need to improve the packet classification process by reducing the ruleset size or implementing a better data structure to store data and speed up the lookup process. With a software solution based on a decision tree, we can find quickly a match between a word and a list of patterns by traversing the tree. However, if the tree is poorly implemented, memory access increases, leading to a longer lookup time.

2.2.2 A Software Based Solution: Decision Tree

Representing rules with a decision tree

Building an efficient decision tree is a key factor in any lookup process. In addition, this structure can be applied to remove redundant rules and detect rule shadowing resulting in a smaller set of rules. Table 2.3 shows a prioritized list of 5 rules on two dimensions, X and Y. In a prioritized list, if a packet matches several rules, the one that comes first in the list has priority, and we apply the corresponding action.

Rules	**X**	**Y**
R_1	[0-3]	[0-3]
R_2	[2-3]	[0-7]
R_3	[0-7]	[4-7]
R_4	[6-7]	[0-1]
R_5	[0-7]	[0-7]

Table 2.3: Example of a prioritized rule list.

Fig. 2.2 shows the 2D representation on two axis X and Y of rules from Table 2.3.

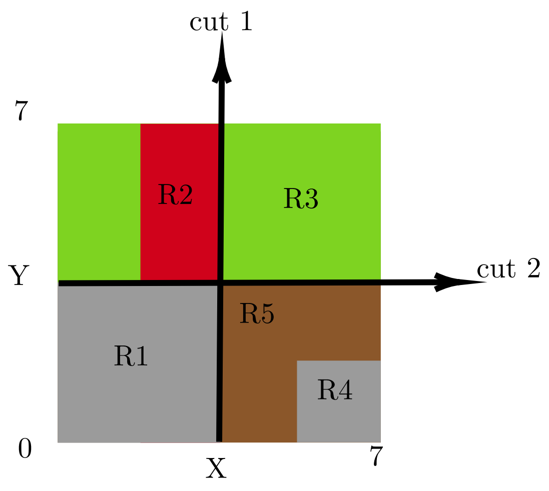


Figure 2.2: A 2D distribution of Table 2.3

Now let us apply Cut 1 and Cut 2 on the 2D representation. The decision tree in Fig. 2.3 shows the different nodes, each belonging to a different part of the network based on the X and Y value. Let $p = (2, 3)$ a packet that needs to be matched. p is in the range ([0-3],[0-3]), by traversing the tree from the root to leaves, we see that Rules R_1, R_2 and R_5 matches p, and since R_1 has the highest priority, the action of R_1 will be applied to p.

The decision tree data structure has to be implemented in the right way for a fast lookup process. For example, if a decision tree has a very long path from the root to a leaf while other paths are short and if most packets need to traverse this long path, the performance of the lookup will be affected. Another problem faced by decision trees is the way a cut is performed. In a set of rules with high overlapping, the duplication in tree nodes will also be increased, affecting the memory occupied by the tree and the lookup time as well.

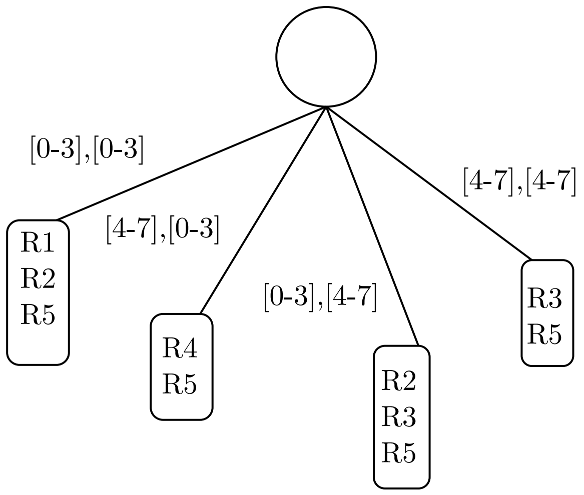


Figure 2.3: Example of a decision-tree of Table 2.3 after Cut 1 and Cut 2.

Decision Tree Based Classifiers

HiCuts [33] is one of the first to apply a decision tree to packet classification. At every packet reception, the tree is traversed to find the leaf that contains a set of candidate rules. Then, this set of rules is be searched to find a match for the received packet. HiCuts works by cutting the two-dimension (rectangle) representation of rules into two equal parts at each time. Each of the parts genrates a node in the decision tree. An extension called HyperCuts [34] allows for cutting the two dimensions (X and Y axis) simultaneously (Fig. 2.3) whereas HiCuts handle only one dimension at a time. HyperCuts minimizes the decision tree depth and speed up the lookup process. HyperSplit [35] achieves better performance by minimizing memory usage and processing time with a different cutting strategy. The authors in [36] propose four heuristics to overcome the extensive memory usage, to reduce the overlapping rules, and to cut the tree into equal dense parts to tackle the variation problem in rule density which occurs with other approaches. CutSplit [37] combines the previous techniques to handle the rule duplication and reduce the memory consumption at the same time. With Multibit Tries [38] two tries representing source and destination are combined in a two-levels tree. Every level is associated with a dimension. This structure can speed up look up time.

While most decision tree techniques described above are developed to accelerate classification by minimizing the number of traversed nodes in a tree, this structure is also helpful to remove redundant rules. In [39], the authors propose an algorithm for reducing the number of entries in TCAM that removes redundant rules based on a tree representation. This structure can be also applied to decompose a set of rules, as we will see in Section 2.5. Although solutions based on a decision tree are easy to implement and modify, and are more scalable, they have some shortcomings. They are slower than hardware solutions and designing an efficient decision tree is hard.

2.3 Minimization of Packet Classification Rules

Solving the rule expansion problem makes the classification process in TCAMs more efficient since this reduction decreases power consumption and overcomes the space limitation. This section will discuss how range encoding and classifier minimization can reduce the packet classification time and simplify rule management.

2.3.1 Range Encoding

An efficient solution to deal with the high number of TCAMs entries is range encoding. It consists in mapping each of the ranges to a short ternary format sequence. Ranges in TCAMs are represented by this sequence without the need to expand them, thus, improving the storage capacity of TCAMs and affecting positively the overall performance.

In [40], Liu and al. propose a technique that requires less TCAM storage space and a deterministic execution time. In their approach, a key is generated for each of the rules and stored alongside the address in TCAM. The key is an n-bit vector $V = \{v_1, v_2..., v_n\}$, where n is the number of distinct ranges. Each of the v_i in this case corresponds to a range with $v_i = 1$ if and only if $v \in R_i$ otherwise v_i will be set to 0. Although this technique can reduce the number of entries in TCAM by relying on ternary format instead of prefixes, it requires extra bits to represent keys in the TCAM memory. An algorithm called SRGE is introduced in [41] to handle the rule expansion problem by encoding ranges with Gray codes. In this binary encoding, two adjacent numbers differ by one bit only. This encoding improves the maximal rule expansion to $2 * w - 4$ prefixes for each range.

From a topological view of the TCAM re-encoding process, the authors in [42] propose a technique to optimize packet classifiers with domain compression to eliminate as many redundant rules as possible, and prefix alignment to reduce range expansion.

A TCAM-based algorithm is presented in [43] called parallel packet classification (P^2C). In this approach, each header field is re-encoded using fewer bits, and then all fields are concatenated to form a TCAM entry. Although we will end up with the same number of rules, the total bit size is smaller. This method is based on a primitive-range hierarchy, where different ranges are mapped to a different layer. Disjoint rangers are put in the same layers and represented by code-words. An approach that relies on inclusion tree (i-tree) called prefix inclusion coding (PIC) is developed in [44] to overcome some drawbacks of the P^2C approach, especially w.r.t. the update cost where all ranges in a layer will be affected by an insertion while in PIC an insertion will affect a small part of ranges in that layer. DRES [45] is a bit-map-based scheme based on a greedy algorithm for range encoding in TCAM co-processors. In DRES, ranges are encoded by the algorithm from [43]. Ranges with the highest prefix expansion will be assigned extra bits. In DRES, decoding is done inside TCAMs, meaning that a modification is necessary and existing network devices cannot be employed directly. To handle the range expansion, [46] proposes a Non-Adjacent Form or NAF. In this approach, a range is re-encoded with a set of signed prefixes. While the encoding reduces ranges, signed representations complicate the matching implementation and performance and add overheads.

The approach *Flow Table Reduction Scheme* has been introduced in [47] to minimize the number of flow entries in SDN. This paper focuses on reducing the number of entries by using a new representation for IP ranges since reducing the number of entries can improve the power consumption of TCAM while respecting the capacity constraint.

The DNF (disjunctive normal form) has also been applied to compute the minimal Boolean expression for a range in linear time [21] and to prove the $2w - 4$ upper bound. Table 2.4 shows a summary of some range encoding techniques already presented.

2.3.2 Classifier Minimization

Another technique to reduce the number of entries before storing them into TCAMs, relies on transforming a given classifier into a semantically

Reference	Approach	Goal	Limitations
[40]	Store a key for each rule in TCAM	Reduce memory usage	Require extra bits
[41]	Gray Coding for ranges	Reduce rule expansion	Require more entry in some cases
[42]	Rule compression / redundancy removal	Reduce classification time	Not optimal / rule can be more compressed
[43]	Rule header re-encoding	Reduce memory usage	Costly update / Hardware modification
[44]	Rule header re-encoding	Reduce memory usage/ Better update cost	Hardware modification
[45]	Bit-map based scheme	Reduce memory usage	Need extra bits / Hardware modification
[46]	Non-Adjacent form	Reduce memory usage	Additional overhead for classification and lookup
[47]	New IP range representation	Reduce memory usage	Costly update

Table 2.4: Summary of range encoding techniques.

equivalent one with fewer TCAMs entries. This technique is called classifier minimization.

A framework presented in [48] is aimed to reduce the size of rulesets in a classifier by running multiple techniques like trimming, merging, expanding, or adding new rules. This approach does not introduce any hardware modification. In [49], an extended version of a tree representation of a classifier called decision diagram allows one to remove redundancy. This technique decreases the number of entries needed in TCAMs.

Alternatively, [16] proposes a greedy algorithm to decompose a multi-dimensional rules list into several one-dimensional rules, and then tries to solve each one apart and finally combines all solutions into a smaller and equivalent classifier.

Most classifier compression techniques rely on a prefix to minimize the number of rulesets. In [30], a non-prefix approach takes advantage of the fact that TCAMs can handle ternary format where the don't care bit '*' can appear at any position. In this case, all TCAMs entries with the same action and with only one bit difference in the same position can be merged. This approach runs in a polynomial-time without any regard to the number of TCAM entries.

In [50], McGeer and Praveen Yalagandula propose an algorithm based on two-level logic optimization to reduce the size of the classifier at first before converting rules into TCAM entries. Unfortunately, although this method can achieve an excellent compression ratio, it is time-consuming.

Compression techniques are also applied to reduce the size of the rulesets. For example, in [51] the authors try to compress routing tables with aggregation on the source field, the destination field, or with a default rule. This technique reduces the number of rules needed for the routing application and has a limited impact on loss rates. In [52], a new compression scheme was presented to minimize the number of policies in a firewall by removing redundant and shadowed rules. Finally, in [53], the authors present a new aggressive reduction algorithm by merging rules relying on a two-dimensional representation. Other techniques besides compression such as shadowing removal, generate a set of rules semantically equivalent to the original one. Authors in [54] propose an approach to detect anomalies in firewall rules like generalization and shadowing and correct these anomalies to reduce the number of rules.

While classifier minimization can reduce the number of entries, the original classifier cannot be computed from the new one resulting in a loss of information that could be helpful if new rules are added or removed.

2.4 Software Defined Networking

The number of users and connected devices are in continuous increase which led to a high demand for new data management and network architecture design that can handle the large amount of data transferred and providing a support of simplified software management. Software Defined Networking (SDN) [55] and its associated OpenFlow protocol have emerged to provide programmability to networks and simply network management.

2.4.1 SDN Architecture

SDN decouples the control plane from the data plane as shown in Fig. 2.4. Switches and routers in SDN perform forwarding packets only. However, the controller in the control plane controls how and where each packet will be handled. This level of separation between the control and data plane builds a more flexible environment. In addition, the controller has an overview of all the network. The controller oversees switches in the data plane via a well-known protocol called OpenFlow [56]. Switches running OpenFlow have one or more tables that can store rules called flow tables. OpenFlow switches can have TCAM or SRAM memory. When a packet arrives at a switch for the first time, a *packet_in* message is sent to the controller if no match is found. Then, the controller sends back a rule to be installed in the switch's flow table with the proper action that needs to be taken for every packet with the same header field as the first received packet. Since switches in SDN apply the matching process based on rules sent from the controller, they can behave like a router, a firewall, a switch, etc.

The controller in SDN called NOS is a software-based platform that runs on a server. The network is programmable through a software application running on top of NOS [57]. In this thesis we consider that any

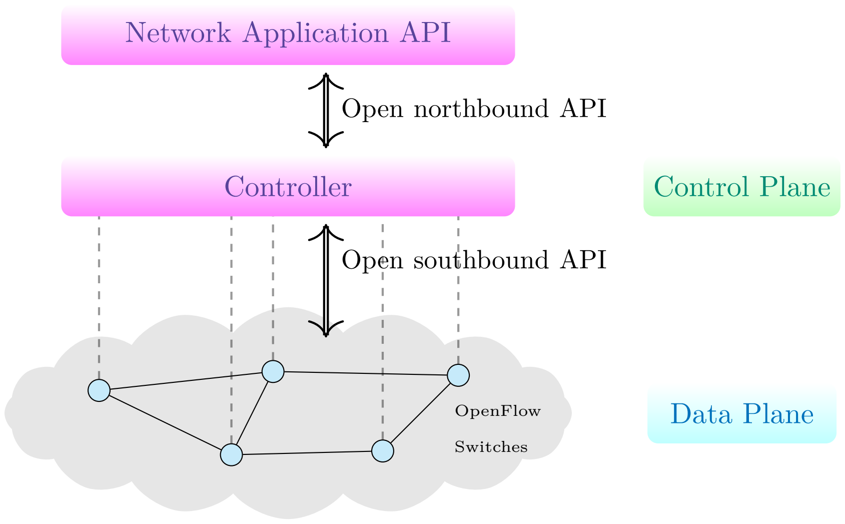


Figure 2.4: Overview of SDN architecture.

packet classification approach needs to run on the controller before sending rules to the switches. For example, if a new classifier is computed in the controller based on an old one but with the same semantics, new rules of the new classifier will be sent to the switches. Since the number of rules in the controller is smaller after applying different techniques, then the number of rules in switches will also be smaller. If switches in SDN use TCAM memory, then the classification process becomes faster. The communication between controller and switches uses the OpenFlow protocol implemented in both the data and control plane. In the next part, we will describe this protocol and its operations.

2.4.2 OpenFlow Protocol

OpenFlow is the most widely used communication protocol in southbound API. Started from an academic project in campus networks [56], this protocol quickly spread between giant companies like Facebook, Google, Microsoft, etc. In addition, OpenFlow-enabled switches have been produced by many vendors, including HP, NetGear, and IBM [58]. While SDN gives an abstraction of the entire network, OpenFlow provides an abstraction for components in the network. Therefore it is important to decouple the two from each other. Even though OpenFlow protocol and SDN can complete each other, they are not tied together. For example, SDN can use other communication protocols for the southbound API like POF [59], OpFlex [60] OpenState [61], Open vSwitch [62], PAD [63].

One key element of OpenFlow is the centralization of the control plane. One controller can be connected to multiple switches, and any decision can be based on the global view of the network instead of having limited knowledge of the network. This centralization can be helpful in case of network failure since, in traditional network architecture, a new path needs to be computed at each switch. However, in SDN, the controller can locally compute a new route and send new rules for each affected switch.

OpenFlow can also be used to analyze traffic in real-time. Each rule in the routing table has a counter that indicates how many times this rule has been matched. This information can be sent to the controller, where it can be analyzed. To detect a denial of service attack (DDoS), authors in [64] propose a new method that uses self-organizing maps to classify network traffic as malicious or not. In [65] a new method for source address validation mechanism is used with OpenFlow where a new rule received by the controller for the first time will be analyzed based on the destination source. OpenFlow is also used in all sort of network application from traffic engineering [66; 67; 68; 69], mobility and wireless [70; 71; 72; 73], monitoring [74; 75; 76], data centers [77; 78; 79], security [80; 81; 82] and many other.

Packets Matching

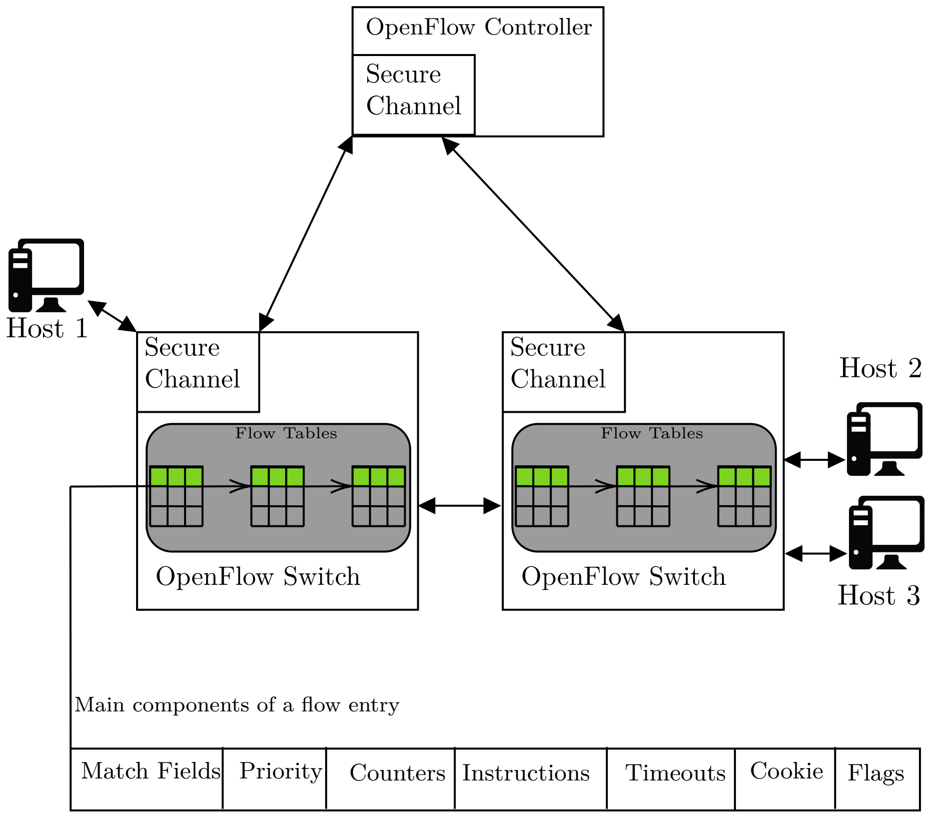


Figure 2.5: Overview of OpenFlow-based Networking.

Fig. 2.5 shows the main components of an OpenFlow architecture. Each OpenFlow-enabled switch has one or multiple flow tables that store rules received from the controller. Each rule in the flow table has a matching field. Incoming packets are compared with the match field. If a packet p matches a rule, then the action associated with that rule is applied. If no match is found, a message will be sent to the controller, which sends back a rule to handle all packets like p in the future without consulting the controller. Counters in flow entry are used to keep statistics about incoming packets. A packet can have multiple matches inside the flow table. Each rule is associated with a priority field that defined which from the matched rules will be considered.

In OpenFlow, when a new packet p arrives, the switch tries to match p with all rules inside flow tables. Let us call the time needed for this process t_1. Next, the switch sends a part of p or the whole packet to the controller via a *packet_in* message if no match is found. When a new message is received, the controller checks his tables for the proper rule r. Let call the time needed for this process t_2. Finally, the controller sends r back to the switch with a *packet_out* message. The total time of this whole process is equal to $t_{tottal} = t_1 + t_2 + t_{packetIn} + t_{packetOut}$. As seen, t_{tottal} can be affected by many variables. First, if the number of entries in flow tables in the switch is high, t_1 will increase. Second, if the number of rules in the controller is high, the matching process inside the controller will need more time to find the right rule, and t_2 will increase. The third factor is the time needed to transfer a packet between the switch and the controller. If the number of packets transferred between all switches and the controller is high, this could create a bottleneck in the controller side and add more delay affecting $t_{packetIn}$. Moreover, fourth, if the switch spends the majority of time sending *packet_in* messages to the controller and trying to match an incoming packet with a large number of entries, the time to process a *packet_out* message will increase. Reducing t_{tottal} means reducing the number of entries in switches and controllers and reducing the number of packets transferred between the two.

Rules Management

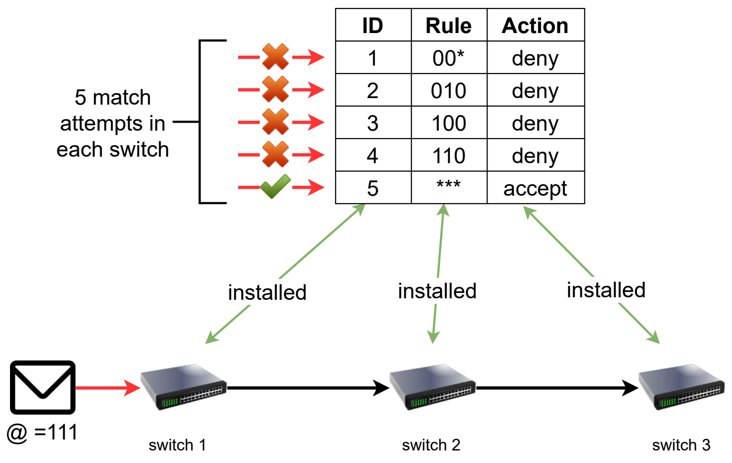
Although, in SDN, the network is seen as one entity, the data plane typically consists of multiple switches. For an operator, the network is seen as one big switch. While managing a network seems simple in this case, we need to resolve how the low-level policies are selected and where they will be installed.

Every OpenFlow switch has one or more flow tables to store rules sent from the controller. Switches need to match every incoming packet against all flow tables entries. This classification and lookup process can turn to be a bottleneck for the network and can add delays. Therefore, some OpenFlow-enabled switches store entries in TCAMs. However, as mentioned before, TCAMs are costly and limited in capacity.

Rules in switches are not the same, and their placement is based on the type of application. For example, rules are stored in the ingress switch with some firewall applications to filter all incoming packets. While in other applications, rules for a given flow need to be installed in all switches for that flow. These rules placement and allocation problems are hard to solve, especially with the sizable network seen nowadays. Therefore, besides knowing the proper place of rules, we need to minimize the total number of rules decomposed over the network to accelerate the classification process and overcome some of OpenFlow-enabled TCAM switches problems like power consumption and space limitation. In addition, switches need to consult the controller for each new flow. The time needed for a switch to check the entire flow tables, send a message to the controller, receive the new rule and store it could be large. By making the lookup time faster, the total time needed to install a rule for a new flow will decrease.

Efforts to reduce the classification process, lookup time and the size of the flow table mentioned in subsection 2.2.2 can be applied to OpenFlow switches. On the controller side, the software-based solution can be implemented to reduce the time needed by the controller to take a decision regarding a new flow.

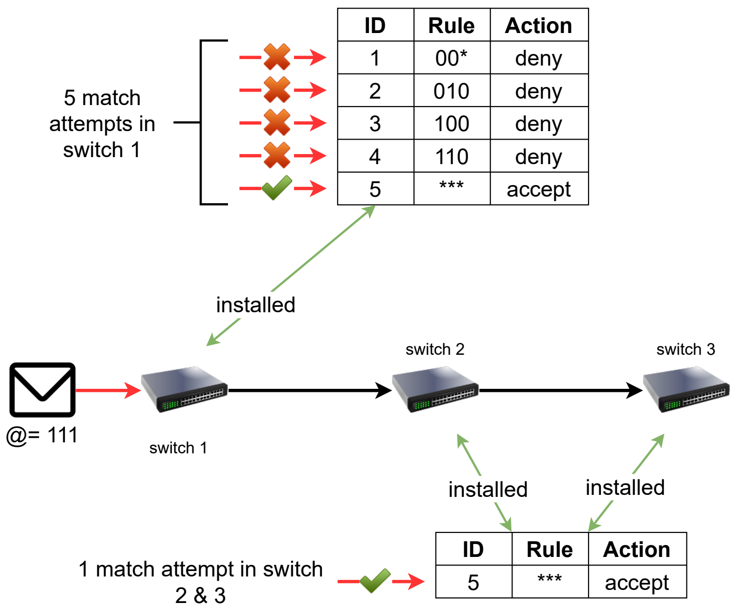
Compression techniques cannot always be practical since usually the nature of rules and the overlapping can block any attempts to reduce the size of a ruleset. Decomposing a set of rules over multiple switches can be helpful in this case. In addition to this, since switches use TCAMs memory, distributing rules over multiple switches can enhance the performance of the overall classification process by minimizing the time in each switch.



ID	Rule	Action
1	00*	deny
2	010	deny
3	100	deny
4	110	deny
5	***	accept

Figure 2.6: Installing rules into a flow table without compression (Example 1).

Fig. 2.6 shows an illustrative example of a flow table with five rules. Now let us install the same table in three switches in series. A packet p with address 111 will only match rule number 5. To traverse the three switches, p needs to be checked with five rules at each switch, a total of 15 times. However, we can see that any packet that matches the first four rules will not be forwarded to the next switch, so rules from 1 to 4 in Switches 2 and 3 will never be used. The right way to distribute rules, in this case, is shown in Fig. 2.7 where all five rules are installed in the first switch and only the rule number 5 in Switch 2 and 3. Now, if packet p needs to traverse the network, only 7 (5+1+1) attempts to match the rule are made.



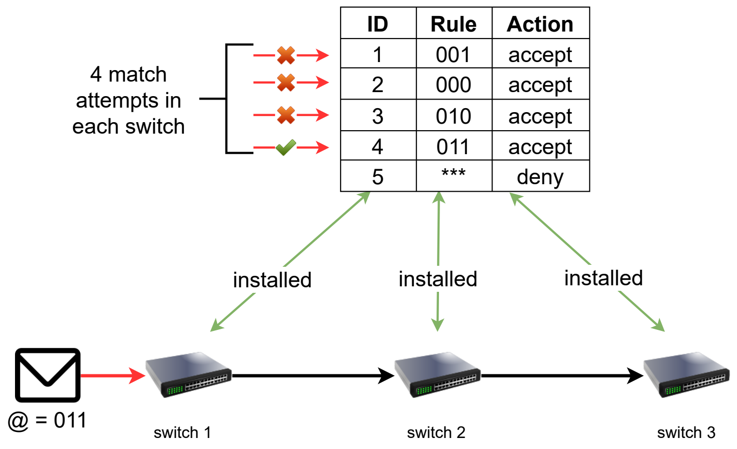
ID	Rule	Action
1	00*	deny
2	010	deny
3	100	deny
4	110	deny
5	***	accept

ID	Rule	Action
5	***	accept

Figure 2.7: Installing rules in flow table with compression (Example 1).

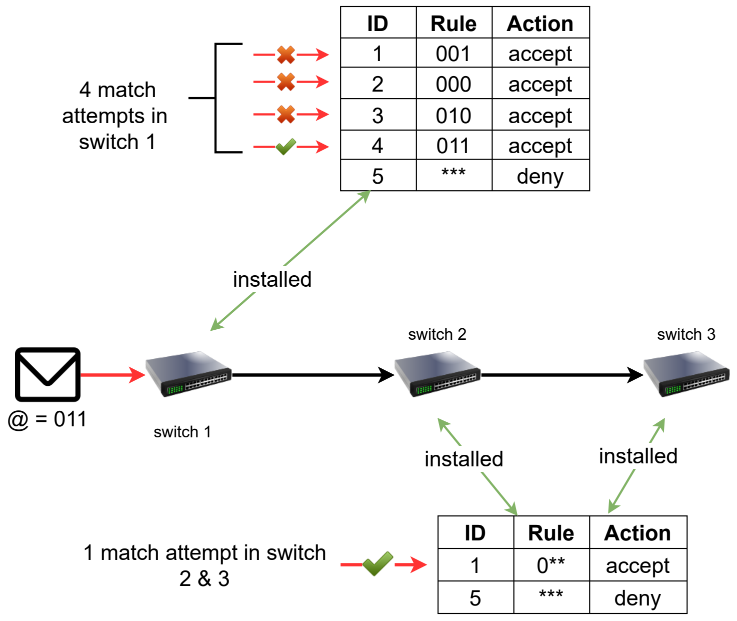
Fig. 2.8 shows another example of a flow table with five rules. If we have the same three switches in series and the same flow table in each switch, a packet p with address 011 will be checked with rule number 4, so four attempts in each switch are needed. The total will be equal to 12 attempts for a packet p to traverse the three switches. Now a better way to distribute rules is shown in Fig. 2.9. In this case, one rule with address $0 * *$ can replace the first four rules in flow tables of Switches 2 and 3. Tables in Switches 2 and 3 will have two rules only instead of 5, and a packet p with address 011 will be checked six times only.

The two examples above show the importance of compression and distribution techniques in making the classification and lookup process faster by minimizing the number of entries in the flow table and decreasing the power consumption and the time needed for a flow of packets to traverse



ID	Rule	Action
1	001	accept
2	000	accept
3	010	accept
4	011	accept
5	***	deny

Figure 2.8: Installing rules in flow table without compression (Example 2).



ID	Rule	Action
1	001	accept
2	000	accept
3	010	accept
4	011	accept
5	***	deny

ID	Rule	Action
1	0**	accept
5	***	deny

Figure 2.9: Installing rules in flow table with compression (Example 2).

a network.

Switch memory size can be a problem in some cases where the ruleset size needs more space than the one available in the switch. For example, if Switch 1,2 of the example in Fig. 2.6 have a capacity of 2, and the switch number 3 have a capacity of 3, the only way the initial set of rules in the table can be installed is if Rules 1,5 are installed in Switch 1, then Rule 2,5 are installed in switch two and finally Rule 3,4 and 5 in Switch 3. The capacity problem is not only specific to the switches in SDN, but the controller also needs to have a smaller ruleset to respond quickly. Authors in [83] for example, try to solve this problem by using multiple controllers. Another proposition that improves the performance and the distribution of the controller can be found in [84].

Messages between switches and controllers can also increase delays. When the number of rules is smaller, the number of messages between the data and controller plane will be smaller. According to [29], a setup of N-Switch path with bi-directional flow generates 2N flow-entry and 2N+4 extra packets. This number of packets can add latency on the controller side. In simple network architecture with a small number of rules, the controller installs all rules in all switches to avoid any *packet_in* messages to minimize the delay and enhance the performance of the packet classification process. This approach can only be applied in a small network with a basic application, while others and because of the space limitations and the complexity of the network, rules need to be decomposed and distributed while less used rules can be stored in the controller.

2.5 Rules Placement

To avoid expensive hardware with large memory, we can decompose rulessets into smaller subsets, each one being assigned to a switch, with less capacity requirements. Beside sparing from high price of TCAMs memory, decomposition accelerates the classification process by allowing to perform matching on smaller set of rules. Without decomposition, some rules will never be matched since they are shadowed by rules with higher priority, for example, or because flows from the same addresses will be blocked in previous switches, and will never reach the switches where they are stored. The decomposition problem is not trivial, since rules usually intersect on multiple dimensions, and each application needs a different decomposition approach. In access-control policies, we use IP source and destination, port source and destination, in addition to the protocol to match a packet. However, in routing applications, only the IP destination is required. While some works aim to reduce the total number of rules generated by the decomposition, others try to minimize energy consumption or maximize traffic satisfaction [85].

2.5.1 Rulesets Decomposition and Placement

To decompose and install rules in a network, the authors in [86] propose a an approach named One Big Switch (OBS), for rules with a two-dimensional endpoint policy. Each section of the two-dimensional representation corresponds to a node in the tree (a cover). With each cut, two nodes are created, each one with its own rules. OBS tries to solve the problem separately for each path.

OBS generates forward rules to avoid the processing of packets already matched by rules in previous switches. The duplication of rules with lower priority and high overlapping with other rules can increase the overhead since, according to their approach, they must be installed in multiple switches. The number of forwarding rules generated can affect the overhead of switches if the set of rules is poorly chosen.

Rule	**Src**	**Dst**	**Action**
R1	00	1*	accpet
R2	11	**	accept
R3	**	00	deny
R4	0*	11	deny
R5	**	**	deny

Table 2.5: Prioritized ruleset.

Let us consider, as an example, the ruleset shown in Table 2.5. This table shows prioritized rulesets composed of 5 rules with different source and destination addresses as well as different actions. This table can be represented by a rectangle as shown in Fig. 2.10.

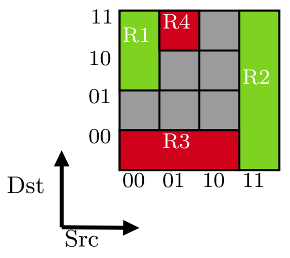


Figure 2.10: A 2D distribution of Table 2.5

The OBS algorithm tries to compute a tree by cutting the rectangle into multiple parts based on several factors like the number of duplicated rules. For example, let us consider a path of two switches in serie. Each one has a capacity of 4. Since the number of rules in Table 2.5 is greater than the capacity of each switch, we need to decompose it into two parts. In Fig. 2.11 the rectangle is divided into two parts. Rules in each part will be installed in a separate switch. As mentioned before, OBS generates forward rules to define all rules installed in previous switches so a packet can be matched only once. In this example, a forward rule will be generated to represent the chosen rules of the blue rectangle of Fig. 2.11-(b). Rules in every switch are depicted in Table 2.6. In Table 2.6-(a), four rules are installed. In Table (b), forward rules that represent all four rules installed in Table 2.6-(a) are generated and added to the remaining rules. A forward rule must be installed at the top of the table. If a packet $p = \{00, 11\}$

matches R1, p will be accepted in Switch 1 and forwarded to Switch 2. Moreover, if the forward rule is added last in Switch 2, p will match $R3$ and be deleted.

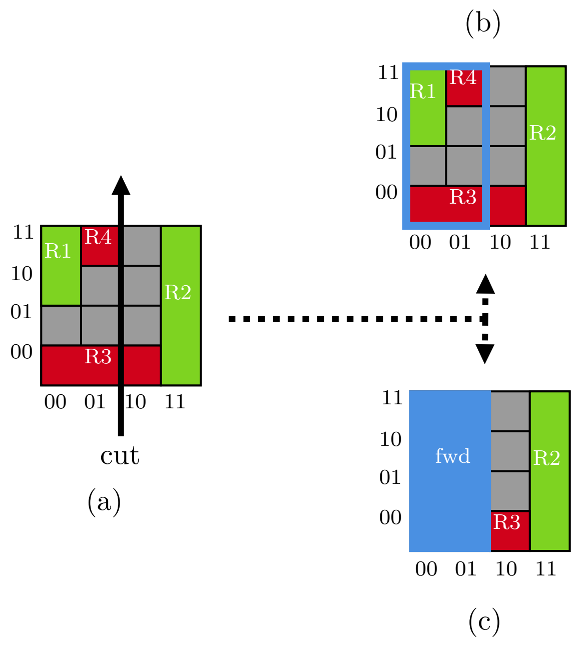


Figure 2.11: Decomposing the 2D representation of Table 2.5

Rule	**Src**	**Dst**	**Action**
R1	00	1*	accpet
R3	**	00	deny
R4	0*	11	deny
R5	**	**	deny

(a) Rules in switch 1

Rule	**Src**	**Dst**	**Action**
fwd	1*	**	forward
R2	11	**	accept
R3	**	00	deny
R5	**	**	deny

(b) Rules in switch 2

Table 2.6: Rules decomposition example for two switches in series.

An optimization for choosing the best coverage in two-dimensions representation is proposed in [87] to improve the storage and the performance of the decomposition of rules. More optimizations have been reported in [37; 88], in order to find the best cut while minimizing the size of the table, the rule duplication, and the pre-processing time for rule updates. Another two-dimension cutting technique like those mentioned in subsection 2.2.2 can also be used to choose the best set of rules in each decomposition.

The decomposition process aims to maximize the number of rules chosen and minimize the number of forwarding rules and the number of rules duplication. Finding the best cover proves to be hard for OBS. Some rule-set has more than 155% overhead caused by the overlapping rules, which increases the number of rule duplication and forward rule generated. For the distribution part, each path in OBS is considered as a set of switches in series. The memory of a switch is divided between all paths traversing the switch.

Palette [89] is another approach for rule placement and decomposition providing two methods for transforming an original rule table into multiple subtables. The first method is called Pivot Bit Decomposition (PBD), where a table is divided into two subtables by changing a wildcard bit "*" to 0 or 1 in each table. The second approach, called Cut-Based Decomposition (CDB), is based on a directed dependency graph. Nodes represent rules while edges represent a dependence between two nodes, in this case, a key that matches both rules. The idea here is to cut the dependency graph into smaller graphs by removing the dependency between nodes. This approach can create problems when the dependency between rules is high. A set of rules with a large number of wildcards generates a denser graph, making the decomposition harder. Palette also proposes a suboptimal greedy algorithm to distribute rules over a network. In this approach, rules are distributed on all shortest paths from the ingress to the egress node. Since some paths can be longer than others, some of the switches in longer paths can stay empty while first switches do all the work.

A near-zero decomposition overhead technique has been proposed in [90]. This technique is based on adding an extra bit to the packet header. When a packet has matched a rule, this bit ensures that the packet will not be processed again. While costing only a bit per rule and in the packet, this stateful technique requires non-standard modifications on the packet structure and leads to security issues in case an attacker can manipulate this additional bit to bypass the filtering mechanism.

The rule distribution problem has been studied in [91] where a predefined set of rules can be shared by multiple paths across the network plus an immediate failure-recovery to a backup path and without any policy violation. In this approach, since rules in a switch belong to all paths, choosing another path in the case where the main one fail will not affect the total behavior of the network since packets can use other paths.

2.5.2 Rules Caching and Swapping

Each application in OpenFlow needs a different rule placement solution. Caching rules is one of the solutions, where a chosen set of rules is cached in switches while others stay in the controller. The way rules are chosen to be cached can affect the performance of the classification process. Rules belonging to a critical flow must be present in switches while others can stay in the controller.

In [28], the authors propose an efficient way to support the abstraction of a switch that relies on caching rules in a software switch while more essential rules will be installed in hardware switches. The algorithm constructs a dependency tree, where essential rules are sent to TCAMs switch, while the rest is sent to a software switch. If no match is found in the hardware switch, the software switch will be checked. If no match is found either, the controller is contacted. This technique avoids rule compression or re-encoding to preserve rule priorities and counters needed for any monitoring application.

CRAFT [92] is another caching technique that can achieve a higher cache hit ratio, meaning that the criteria of rule importance that chosen rules are based on is better than others like [28]. In this approach, each sub-table has a weight based on the hit ratio of his rules. The hit ratio is the number of times a rule has been matched with a packet. If the hit ratio of a rule is high, the rule will have more importance. Sub-table with the highest hit ratio will be cashed.

Huang *et* al. in [93] propose a rule partition and allocation algorithm to distribute rules across network. Rules with the same policy or have a dependency will be put in the same sub-table. Like OBS, the decomposition is based on a flow path. Meaning that on each switch, a space will be allocated to each flow.

The approach in [94] aims to overcome the space limitation of TCAMs switch by swapping rules between the flow table of the TCAM and a less powerful but more significant memory in the controller called Memory Management System (MMS). Rules with smaller hit ratio will be moved to MMS while other will be installed in the TCAMs flow table. Other solutions like [95] use a caching mechanism on flow-driven rule achieving a high cache hit ratio.

Angelos *et* al. [96] propose two types of tables: a cheap software table with large capacity or an expensive hardware table with less capacity. The goal here is to increase the utilization of the software-based table without affecting the network's overall performance in terms of packet loss or delays. For a more scalable system, DIFANE [97] proposes keeping all traffic in the data plane by directing traffic through an intermediate switch called the authority switch, as seen in Fig. 2.12. The controller installs rules in the authority switches while those play the role of a controller, thus sending rules to the remaining switches of the network. Although DIFANE can be scalable, rules duplication on the authority switches and the controller can be costly, especially with a large ruleset.

2.5.3 Path Based Rules Placement

Distributing forward rules differs from access control rules since rules are tied to flows, and each flow has a path. The best path for a flow is chosen based on the application like traffic engineering, load balancing, routing,

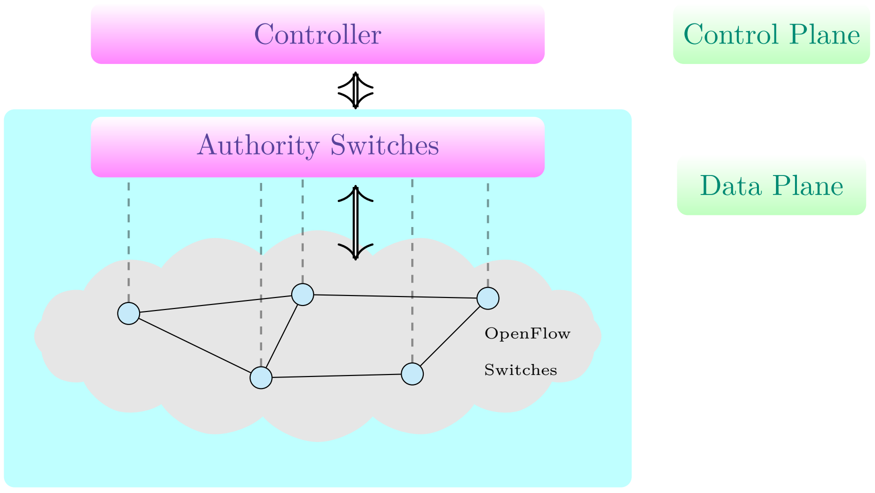


Figure 2.12: DIFANE architecture.

etc. For example, the shortest path is always the best in routing applications, but this solution does not always work due to capacity constraints. To deal with this problem, we can relax the constraint of the shortest path by choosing another with more switches but with higher capacity. Path heuristic is used in [98] to find the best path while respecting as many policies as possible within the resource limits of the network. The flow of data can be enhanced by mobilizing several paths for it. In this case, copying the same rules on each path will lead to high TCAM power consumption and poor space management. Moreover, the fact that paths have different lengths and resources makes it impossible sometimes to deploy the same set of rules over two paths. Intersection nodes between the same flow can be helpful since rules do not have to be duplicated, but at the same time, we need to ensure that rules before the intersection node are the same between all paths, same for rules in following switches. In [99], paths for the same flow are chosen to satisfy the requirement and, at the same time, maximizing the number of nodes. The placement of rules and paths can also be predicted for mobile users with SDN-enabled access networks. Due to users' fast movement and mobility in such a network, predicting the next step is essential to avoid interruption in connectivity and quality of service (QoS). MoRule [100] proposes an efficient rule management scheme to optimize the rule placement problem for mobile users and minimize rule space occupation at switches.

OPTree [101] is another proposed data structure that considers the position of a device for placing rules in a network. The approach checks if a rule is covered by another one or if it can be merged. The approach also builds a relationship between devices based on whether devices are in series or parallel.

2.5.4 Rules Update

Network topologies change needs to be handled to ensure the security and stability of the network. As mentioned before, according to [10] 20% of all failures in a network can be attributed to a poorly planned and configured update. This interruption plus the delay of an update is costly. A study shows that Amazon will lose 1% of the sales amount for every 100ms of latency [102]. If an update fails, the delays will increase, resulting in more losses. In addition, When adding an entry in a TCAM memory, all the addresses of existing entries with lower priority must be moved. This process makes the TCAM update slower. According to [103], n flow entries need n/2 movements to insert a new flow in TCAM.

The delay is not the only problem, since an incorrect update can cause forwarding loops where packets will be stuck in the network without reaching the destination. Forwarding black hole is another problem where during an update, an incoming packet cannot match any entries in the flow table or if an update matches an old entry that has not been updated yet by the controller because of some delays. A solution named "add before remove" is presented in [104] to try solving the forwarding black hole problem. In this solution, new rules with high priority are added to the tables before removing old rules. The space limit in switches can cause problems in this approach when no space is available for new rules to be installed and forces to remove the old ones before adding any new rule leading to connection problems since no old or new rules are present in the switch. A generalised and more enhanced approach of "add before remove" is introduced in [105]. In this approach, switches receive a new version of rules but still process the packet classification and lookup according to the old ones. When all switches receive the new rules, all new packets will be checked according to the new set, and rules from the old version will be deleted. When a packet enters the network, the controller defines which version the packet will be processed. This will avoid the problem of matching a packet according to two different versions in the network.

Another problem is link congestion, where during an update, a link becomes congested by flows after redirecting some of them to a link already used by other flows. In [106] a solution for link congestion is presented called SWAN. In this approach, each link has a capacity that will only be used in an update. Authors show that if the left capacity is equal to s, a congestion-free update sequence needs no more than $1/s - 1$ steps. In [107], the authors propose another congestion-free approach for a consistent update. The new generic algorithm trades update time for rule-space overhead. The updated policy is spitted into multiple slices then applying one at a time. Liu *et* al. [108] propose ZUpdate to perform a congestion-free network updates in data centers.

Network policy violation is another form of problems in network updates. A packet can be blocked or allowed by mistake due to a bad configured update in middleboxes. To overcome this problem, we add Tags to each packet. Next, the mapping between the tag and the IP of the packet is sent to the controller [109]. Switches forward packets according to packets tags. This ensures a consistent network policy for packets, especially if middleboxes modify their header or content.

[110] propose a resource constrain splitting algorithm to deal with the dynamic nature of networks by updating only the set of rules affected by an update and not the entire topology since generating a solution with integer linear programming (ILP) takes long time, especially in the case of high rule overlapping. A firewall is divided into multiple sets based on the IP source of rules, for example. If a newly added rule overlaps with a subset of rules, only the dependent subproblems will be solved with the ILP. Dependency graphs (DAG) are used in [111] to minimize the update latency by avoiding unnecessary entry moves. With DAG, if a new rule r is inserted, only rules on which r depends will be moved, reducing the overall update time. A greedy algorithm that aims to reduce the time complexity of an update and uses the dependency relationship between rules to avoid any unnecessary rule modification is presented in [103]. According to the authors, the algorithm is more than 100 times faster than RuleTris [111] by using a faster TCAM update scheduler.

2.6 Summary

Packet classification is a fundamental task for several network devices. With the increasing number of users and applications, entries in tables and rules sets will only increase. Many existing work have tried to resolve this problem by software-based approaches: range encoding, classifier minimization, removing redundancy, etc. A first category of approaches try to deal with the classification problem by minimizing the lookup process time. This can be accomplished by developing data structures similar to decision trees, to ease rule lookup. A second category of approaches rely on removing redundant and shadowing rules. While these approaches can reduce the number of entries, the final solution is not optimal, and other compression techniques should be applied to reduce the ruleset. Other approaches applied a hardware-based solution by modifying the TCAM memory to store rules with a specific classifier matching only a part of them which reduces the packet classification time. However, modifying a TCAM architecture is costly and creates compatibility issues between TCAMs devices from different manufacturers.

In this thesis, we focus on one aspect of the packet classification problem, rule minimization. To reduce the number of entries in rulesets, we will introduce a new representation for the CIDR notation called *Double*

Mask. This notation permit to accept and deny a set of IP address at once. To transform a set of rules in CIDR notation to a set of *Double Masks* we have developed a polynomial algorithm that we call Generate-DMasks. Matching a packet with a CIDR notation is different from the matching process of *Double Mask*. Therefore we have developed a matching algorithm for our new notation. We have implemented our solution in a real testbed with a Zodiac-Fx switch and a Ryu controller to test our new approach.

With the limitation in memory space in switches nowadays, compression techniques cannot be sufficient and have to be complemented by ruleset decomposition and distribution. The approaches discussed in this chapter generate forward rules and duplicate others, increasing by that the overhead. Other approaches try to reduce the overhead by modifying packets leading to security issues. Table 2.7 shows a summary of rule placement techniques used to divide rules over switches in an SDN network. Any rule decomposition's primary goal is to minimize the overhead inside switches with limited capacity and the number of *packet_in* messages between switches and controllers to achieve a better QoS. In this thesis, we combine multiple techniques and solutions and develop a general approach for any rules while respecting switch limitations in terms of capacity.

Since most of the works on decomposition in the literature deal with a set of prioritized rulesets, in our work, we first introduce a decomposition algorithm for one-dimension rules and Longest Prefix Matching. Although this algorithm generates forward rules, we tried to reduce their overhead by considering the action field of rules. A rule with a deny action will not have a forward rule since a packet that matches this rule will be deleted and does not traverse to the following switches.

For decomposing a set of rules on multiple dimensions, we introduce an approach that works with priority lists, and does not require any forward rules at all or any modification to the hardware or to packets but at the cost of one additional switch. We design this switch as a forward switch or FoS. When a packet is matched, the switch sends it to FoS, which forwards it to the destination. Thus minimizing the time needed for the packet to traverse the network and avoiding any additional rules in switches. After the decomposition of rulesets, we need to distribute them over memory limited switches. In our approach, we consider the position of each switch in the network. We start by developing an algorithm for series-parallel graphs. Our algorithm works with any decomposition algorithm like the one in [86]. We then generalize the algorithm to work with any st-dags.

Updates, on the other hand, are frequent in any network application. Contrary to some approaches mentioned before, switches in our technique depend on each other. The distribution algorithm that we have designed can be applied with any decomposition algorithm as the one in [86], which

Reference	Approach	Discussions	Packet modification	Forward rules	Rule duplication
[28]	Rule caching, Dependency graph	Used for access control rules	no	no	no
[86]	2D Representation	For forwarding rules	no	yes	yes
[89]	Pivot Bit Decomposition, Cut-Based Decomposition, Dependency graph	For access control list	no	N/A	no
[90]	Add extra bits for each packet	Reduce classification time	yes	no	no
[91]	Two type of rules (shared/non shared)	Multi path routing, Failure-recovery	no	no	no
[92]	Rule caching based on hit ratio, Minimize overlapping rules	Solve the problem of long chains of overlapped rules	no	no	no
[93]	Rule caching	Flow rules	no	no	no
[94]	Rule swapping mechanism between two type of memory	Optimizes the usage of network devices memory	no	no	no
[96]	Two type of rule table (hardware and software)	Very high computational complexity between software and hardware switches	no	no	no
[97]	Add an intermediate switch between network switches and controller	Faster response time	no	no	yes
[101]	Take the position relationship between neighbor devices into consideration	For access control list	no	no	no

Table 2.7: Summary of rule placement techniques.

means that rules in switches, especially forward rules, depend on rules installed in previous switches. In this case, any modification will affect all the following switches but not the previous ones. A naive solution will be to run an update on the whole topology with the new rulesets. However, in our approach, we consider only the affected part of the topology and apply the same decomposition and distribution algorithm mentioned before to this subtopology.

Chapter 3

Filtering Rules Compression with Double Masks

Contents

3.1 Introduction

Multiple network appliances and applications including firewalls, intrusion detection systems, routers, and load balancers rely on a filtering process using sets of rules to decide whether to accept or deny an incoming packet. Effective filtering is essential to handle the rapidly increasing and dynamic nature of network traffic where more and more nodes are connected, due to the emergence of 5G networks and the increasing number of sources of attacks. With many hosts, it remains crucial to minimize the number of entries in routing tables and accelerate the lookup process. On the other hand, attacks on Internet keep increasing according to [112], which increases the size of blacklists and the number of rules in firewalls. The limited storage capacity of switches [52] requires efficient space management. To face the large number of hosts and routing tables, [17] developed Classless Inter-Domain Routing (CIDR) to replace the classful network architecture. However, using this notation to represent routing table rules that contain ranges can lead to multiple entries, and thus there is a need for a better notation along with an efficient algorithm to reduce the number of entries and therefore the classification and lookup time, and memory usage [46]. Since TCAM is the standard for rules storage and matching in packet classification for Openflow switches, multiple attempts to solve their problems was considered in [23; 40; 113; 114; 115]. Range expansion is one of the most important problems faced by TCAMs. This problem occurs when the port range of rules have exceptions. In the worst-case scenario, a rule needs more than 900 entries to be represented, as seen in Table 2.1.

In this Chapter, we design a simple representation of filtering rules that enables more compact rule tables, easier to manage while keeping their semantics unchanged. The construction of rules is obtained with reasonably efficient algorithms too. This representation applies to IP addresses and port ranges to mitigate the range expansion problem. For that, we express packet filter fields with so-called *Double Mask* [116], where a first mask is used as an inclusion prefix and the second as an exclusion one. This representation adds flexibility and efficiency in deploying security policies since the generated rules are easier to manage. The *Double Mask* representation makes configurations simpler since we can accept and exclude IPs within the same rule. A *Double Mask* rule can be viewed as an extension of a standard prefix rule with exceptions. It is often more intuitive than the alternative representations and therefore prevent errors in network management operations. Our work is software-based, and relies only on accept rules, unlike [46; 117; 118]. Our notation can reduce dramatically the number of entries in routing tables. In comparison, representing a w-bit range may need $2w - 2$ prefixes [119]. For example $[1, 14]$ needs 6 entries but with the *Double Mask* notation two entries are sufficient. This new notation has the same upper bound of $2w - 4$ presented in other papers [21; 41], but in some cases, the number can be reduced as shown before in our experimental results.

This chapter is organized as follows. In Section 3.2 we introduce our new representation. Section 3.3 describes our algorithm for computing a *Double Mask*. Then, in Section 3.4 we evaluate the performance of our linear algorithm. Next, in Section 3.5 we develop a real testbed and evaluate our matching algorithm's performance. In Section 3.6 we discuss how we can extend our representation to achieve a higher compression ratio. Finally, Section 3.7 summarizes and concludes this chapter.

3.2 Double Mask Technique

3.2.1 Preliminaries

Before introducing the *Double Mask* representation, we define the notation used throughout the thesis, that is summarized in Table 3.1.

ip	IP address
w	number of bits representing an IP address
P	prefix covering an ip
$bin_v(a)$	binary representation of integer a using v bits
$val_v(a)$	integer value of bitstring a with length v
range $[a, b]$	set of IP addresses with value between a and b
t_w	perfect binary tree of height w
ε	empty bitstring
p	bitstring (or path in t_w)
$\|p\|$	length of p
$t(p)$	perfect binary subtree of t_w with root p
$l(p)$	set of leaves of $t(p)$
DM_p	set of *Double Masks* covering $l(p)$

Table 3.1: Employed notation.

Prefix and *Simple Mask*

A prefix P is a word of length w on alphabet $\{0, 1, *\}$ where all $*$'s occur at the end of the word: $P = p_{k-1}...p_0*^i$ where $k + i = w$. To avoid confusion with the usual notion of word prefix, we will also sometimes call P a *Simple Mask*. An address $ip \in \{0, 1\}^w$ is covered by *Simple Mask* P if $ip_i = p_i \;\; for\ i \in [k, k - i + 1]$. The subword $p = p_{k-1}...p_0$ is called the path of P for reasons to be explained below.

Range

A range is denoted by an integer interval $[a, b]$ (where $0 \leq a \leq b \leq 2^w - 1$). A range represents the set of IP addresses ip, with integer value $val_w(ip)$ between a and b.

Perfect Range

$[a, b]$ is a perfect range if there is $r \in \{0,1\}^{w-k}$ such that $bin_w(a) = r.0^k$ and $bin_w(b) = r.1^k$.

Perfect Binary Tree

The IP addresses of length w are in bijection with the leaves of a perfect binary tree t_w of height w. More generally, we can define the following bijection $t()$ on the set of bitstrings of length $\leq w$ with the perfect binary subtrees of t_w : $t(\varepsilon) = t_w$ where ε is the empty bitstring, and given bitstring v we define $t(v0)$ (resp. $t(v1)$) to be the left (resp. right) subtree of $t(v)$. In particular, if prefix P has a path p of length k the IP addresses covered by P are exactly the leaves of the perfect subtree $t(p)$. This set of addresses is a perfect range. Every perfect range is also the set of leaves of a perfect subtree.

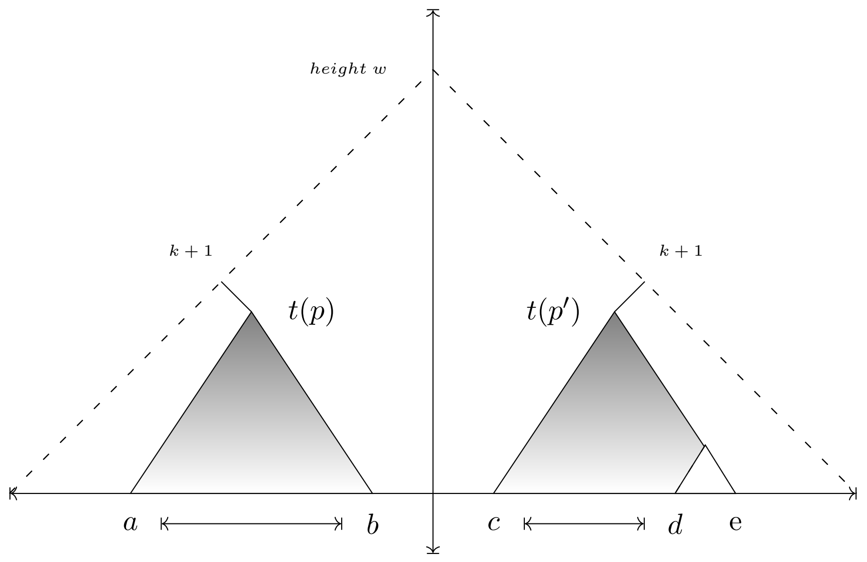


Figure 3.1: Illustration of a perfect Binary Tree.

In Fig. 3.1, $[a, b]$ is a perfect range as it is the set of leaves of the perfect binary tree $t(p)$ of height k. However, $[c, d]$ is not a perfect range as it is not the set of leaves of a perfect binary tree.

3.2.2 Double Mask Representation

Now we will define the *Double Mask* representation for range fields, in particular for IP address fields. Note that the representation can be applied to other range fields such as ports.

We assume in the following that IP addresses are binary words of length w, i.e., IP addresses are elements of $\{0,1\}^w$ indexed from 1 to w. A *Double Mask* representation has three components and is denoted $netpref/mask1/mask2$. The first component $netpref \in \{0,1\}^w$ is a network prefix. The second and third components are integers $mask1, mask2 \in [0, w]$. Component $mask1$ defines all accepted IPs, and component $mask2$ defines all excluded IPs from the list of accepted ones. In case where $mask2$ is equal to 0, the *Double Mask* will be equivalent to a *Simple Mask*.

Definition 1. *An IP address ip is in the set defined by* $netpref/mask1/mask2$ *if* $ip_i = netpref_i$ *for* $i \in [1, \ldots, mask1]$ *and there exists* $j \in [mask1 + 1, mask2]$ *such that* $ip_j \neq netpref_j$. *In that case we say that ip is covered by* $netpref/mask1/mask2$.

Let us consider the following example of a *Double Mask* representation:

192.168.100.96/26/2

This representation means that any selected (or filtered) address must have its 26 first bits equal to the 26 first bits of 192.168.100.96 (that is equal to 11000000.10101000.01100100.01), and at least one of the two following Bits 27,28 should not be equal to the corresponding bit of 192.168.100.96. In other words, either Bit 27 is not 1 or Bit 28 is not 0. As we will see, this new representation can reduce the number of filtering rules dramatically. It is also possible to represent more explicitly the *Double Mask* as a word where the forbidden combination of bits is overlined, the leftmost part specifies the fixed bits and the rightmost part the free bits (that are allowed to take any value). The two possible notations of a *Double Mask* are given below:

$$N1: \quad a_{k-1}...a_0a_{j-1}...a_00_{i-1}...0_0/k/j$$

$$N2: \quad a_{k-1}...a_0\overline{a_{j-1}...a_0}0_{i-1}...0_0$$

where $(i + j + k = w)$, and if $j = 0$ the *Double Mask* is equivalent to a *Simple Mask* (or a TCAM entry) $a_{k-1}...a_0*^{w-k}$.

When designing filtering rules, it is proper to represent the excluded addresses rather than the accepted ones when there are many more excluded addresses than accepted ones. In this case, using a *Double Mask* representation has a better effect since by reducing the number of filtering rules, we reduce the computation time, memory, and power usage.

The examples below illustrate the benefits of using *Double Masks* over *Simple Masks*.

Example 1. *Range [1,14] needs a set of 6 standard prefixes to be represented. However this range can be represented using only two Double Masks prefixes as shown below :*

$$\begin{array}{ccc} \textit{range} & \textit{Simple Masks} & \textit{Double Masks} \\ [1,14] = & \begin{cases} 0001 \\ 001* \\ 01** \\ 10** \\ 110* \\ 1110 \end{cases} & \begin{cases} \overline{0000} \\ \overline{1111} \end{cases} \end{array}$$

Example 2. *Range $[1, 15]$ is of form $[1, 2^4 - 1]$ and needs 4 simple masks $\{0001, 001*, 01**, 1***\}$ but only one Double Mask: $\overline{0000}$.*
More generally, a range $[1, 2^w - 1]$ can be represented by a unique Double Mask $\overline{0}^w$ However, it cannot be represented by less than w Simple Masks. Let us demonstrate this by contradiction. Let us assume that $[1, 2^w - 1]$ can be represented by strictly less than w Simple Masks. Then at least two different addresses $2^i - 1, 2^j - 1 (j > i)$ are covered by the same mask. The mask has to be a common prefix of their binary representations: therefore it has to be a prefix of 0^{w-j}. However, in that case, the mask would also cover 0^w, which is a contradiction.

3.3 Double Mask Computation Algorithms

3.3.1 Naive Algorithm

We now present an algorithm to generate a set of *Double Masks* that covers a range $[a, b]$, i.e., selects exactly the addresses in this range. The algorithm proceeds recursively on the binary tree $t_w = t(\varepsilon)$ that stores all IP addresses of size w. Note that each node of $t(\varepsilon)$ can be located uniquely by a path (bitstring) p from the root to this node: the root is located by ε; the left and right children of the node located by p are located by $p0$ and $p1$, respectively. We will identify a node with the path that locates it. A path can also be viewed as a prefix where the $*$'s are omitted. The leaves of $t(\varepsilon)$ are the IP addresses. We denote the set of leaves of subtree $t(p)$ by $l(p)$. Algorithm 1 computes in a bottom-up way a set of *Double Masks* covering $l(p)$. Moreover we denote these partial results by by DM_p. We denote by $\bar{i}$ the complement of boolean i, i.e., $\bar{0} = 1$, $\bar{1} = 0$. To process a node p in $t(\varepsilon)$ we have to consider several cases according to the left and right children of p, as described below and as illustrated in Fig. 3.2.

Case 0: if p is a leaf and $l(p) \subseteq [a, b]$, then
$DM_p = \{p/|p|/0\}$, else $\emptyset$

Case 1: if $l(p0)$ and $l(p1)$ are both subsets of $[a, b]$ then
$DM_p = \{p0^{w-|p|}/|p|/0\}$

Case 2: if there is a unique $i \in \{0, 1\}$ such that $l(pi)$ is a subset of $[a, b]$ then

Case 2.1: if $DM_{p\bar{i}} = \{p\bar{i}dq/|p|+2/0\}$ $(d \in \{0,1\})$ then
$DM_p = \{p\bar{i}dq/|p|/2\}$

Case 2.2: if $DM_{p\bar{i}} = \{p\bar{i}q/|p|+1/m\}$ (where $m > 0$) then
$DM_p = \{p\bar{i}q/|p|/m+1\}$

Case 3: Otherwise $DM_p = DM_{p0} \cup DM_{p1}$

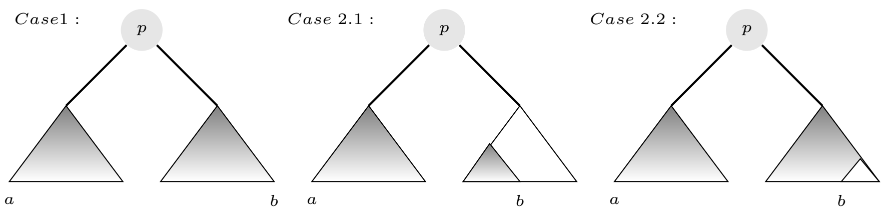


Figure 3.2: Typical examples for *Cases* 1, 2.1 *and* 2.2

Given a range $[a,b]$, the set of *Double Masks* DM_ε returned by Algorithm 1 covers $[a,b]$. The proof of correctness is to demonstrate by induction on $w-|p|$ that DM_p spans $l(p) \cap [a,b]$. For the base case $|p| = w$ and $l(p)$ is an IP address: then $DM_p = \{p/0/0\}$. For the induction step we have to prove by cases that if DP_{pi} spans $l(pi) \cap [a,b]$ and $DP_{p\bar{i}}$ spans $l(p\bar{i}) \cap [a,b]$ then DM_p spans $l(p) \cap [a,b]$. We then conclude that DM_ε spans $l(\varepsilon) \cap [a,b] = [a,b]$. Fig. 3.3 gives an illustrative example of the algorithm execution.

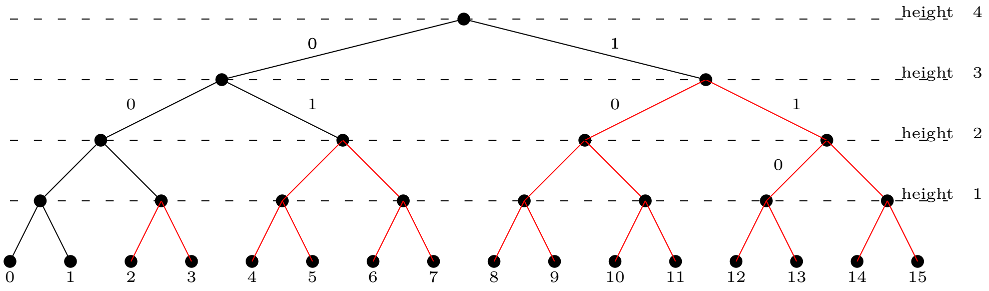


Figure 3.3: Illustration of the execution of Algorithm 1

Let $[a,b] = [2,15]$. The algorithm start from the button. $[2,2]$ is a leaf and $\in [a,b]$. According to Case 0, $DM_{0010} = \{0010/1/0\}$. Same for each leaf in $[a,b]$. At Height 1, if $l(p0), l(p1) \subseteq [a,b]$, the algorithm return $DM_p = \{p0^{w-|p|}/|p|/0\}$. For $[2,3]$, $DM_{001} = \{0010/3/0\}$ according to Case 1 since $l(p0) = 2$ and $l(p1) = 3$. At Height 2, according to Case 3, $DM_{00} = DM_{000} \cup DM_{001}$, but $DM_{000} = \emptyset$ since $0,1 \notin [a,b]$, so $DM_{00} = \{0010/3/0\}$. For $[4,7], [8,11]$ *and* $[12,15]$, $l(p0)$, $l(p1) \subseteq$

Algorithm 1 DM-Naive(a,b)

```
Input: a,b
Output: set of Double Masks representing [a,b]
return $DM_\varepsilon$ where:
if $p$ is a leaf then
  if $p \notin [a,b]$ then
    return $DM_p = \emptyset$
  else

    return $DM_p = \{p/|p|/0\}$                              ▷Case 0
  end if
end if
if $l(p0), l(p1) \subseteq [a,b]$ then

  return $DM_p = \{p0^{w-|p|}/|p|/0\}$                        ▷Case 1
else
  if $l(pi) \subseteq [a,b]$ then
    if $DM_{p\bar{i}} = \{p\bar{i}dq/|p|+2/0\}$ $(d \in \{0,1\})$ then

      return $DM_p = \{p\bar{i}\bar{d}q/|p|/2\}$             ▷Case 2.1
    else
      if $DM_{p\bar{i}} = \{p\bar{i}q/|p|+1/m\}$  $(m > 0)$ then

        return $DM_p = \{p\bar{i}q/|p|/m+1\}$                ▷Case 2.2
      end if
    end if
  end if
end if

return $DM_p = DM_{p0} \cup DM_{p1}$                           ▷Case 3
```

$[a,b]$. For example, in $[4,7]$, $l(010)$, $l(011) \subseteq [2,15]$, the algorithm return $DM_{01} = \{p0^{w-|p|}/|p|/0\} = \{0100/2/0\}$ according to Case 1. At Height 3, for $[2,7]$, $l(00) \notin [2,15]$ but $l(01) \in [2,15]$ and $DM_{00} = \{p\bar{i}dq/|p|+2/0\} = \{0010/3/0\}$. DM_0 will be equal to $\{0000/1/2\}$ according to Case 2.1. For $[8,15]$ the algorithm return $DM_1 = \{1000/1/0\}$ since $l(10)$, $l(11) \subseteq [2,15]$. At height 4, $l(0) \notin [2,15]$, but $DM_0 = \{0000/1/2\}$, according to Case 2.2 the algorithm return $DM = \{0000/0/3\}$. By performing a case analysis, we can show the following results:

Proposition 1. *Let $v \geq 2$ and $0 \leq a,b \leq 2^v - 1$. Any range of type $[a, 2^v - 1]$ or $[0,b]$ can be represented by at most $v-1$ masks.*

Proof: By symmetry we only consider $[a, 2^v - 1]$. We perform an induction on v. If $bin_v(a) = 0^k 1s$ with $k \leq v - 2$. By induction hypothesis applied to $v - k - 1$ $[val_v(s), 2^{v-k-1} - 1]$ can be represented by $v - k - 1$ masks in $t(0^k 1)$. These masks can be extended to masks in v bits by adding $0^k 1$ to the left of the network prefix and adjusting the components of the mask. The complement $[2^{v-k-1}, 2^v - 1]$ can be represented by the k prefixes $01{*}^{v-2}, 0^2 1{*}^{v-3}, \ldots, 0^{k-1} 1{*}^{v-k}$. Overall we get $(v - k - 1) + k = v - 1$ masks. If $a = 0^{v-1}1$ then $[a, 2^v - 1]$ is represented by a *Double Mask* excluding 0. If $a = 0^v$ then $[a, 2^v - 1]$ is represented by a *Simple Mask* associated to prefix ${*}^v$. Hence the proposition holds. $\square$

Using Proposition 1, we will show now that, for $w > 2$, any range can be covered with at most $2w - 4$ *Double Masks.*

Proposition 2. *Let $w > 2$. Every range $[a, b] \subseteq [0, 2^w - 1]$ can be represented by at most $2w - 4$ masks.*

Proof: Let $[a, b]$ be a range of addresses of length w. It is well known, according to [120], that $n \leq 2w - 2$ *Simple Masks* are sufficient to represent a range $[a, b] \subseteq [0, 2^w - 1]$.

Now let us prove by induction that a range $[a, b]$ of w bits can be represented with $\leq 2w - 4$ masks.
For $w = 3$, two masks are sufficient. For $w = 4$, four masks are sufficient to represent any range $[a, b]$.
Assume the proposition holds for $w - 1$ bits. We perform a case analysis and assume that one case is applied only if the previous ones are not applicable:

- $[a, b] \subseteq l(0)$ or $[a, b] \subseteq l(1)$ then by induction hypothesis the proposition holds.

- $[a, b] \subseteq l(01) \cup l(10)$ then applying Proposition 1 to $[a, b] \cap l(0)$ with $v = w - 2$ we obtain that $[a, b] \cap l(0)$ is covered by $v - 3$ masks. These masks can be extended to masks in w bits. In the same way $[a, b] \cap l(0)$ is covered by $w - 3$ masks. Therefore $[a, b]$ can be represented with $2w - 6$ masks.

- $[a, b] \subseteq l(01) \cup l(10) \cup l(11)$ we apply the previous item reasoning to show that $[a, b] \cap (l(01) \cup l(11))$ is represented by $2w - 6$ masks. Since one mask is sufficient for perfect range $[a, b] \cap l(10)$, we obtain overall $2w - 5$ masks.

- $[a, b] \subseteq l(00) \cup l(01) \cup l(10)$: we reason as in the previous case.

- $[a, b] \subseteq l(00) \cup l(01) \cup l(10) \cup l(11)$: we need $2w - 6$ masks for $[a, b] \cap (l(00) \cup l(11))$, one mask for each of $[a, b] \cap l(01)$ and $[a, b] \cap l(10)$ since they are perfect ranges. Hence overall $2w - 6 + 2 = 2w - 4$ masks are sufficient.

Therefore the total number of masks needed to represent $[a, b]$ is $2w-4$. $\square$

The following proposition shows that the $2w - 4$ bound is tight, i.e., some ranges cannot be represented by less than $2w - 4$ *Double Masks.*

Proposition 3. *Let* $w > 3$. *The range* $[3, 2^w - 4]$ *cannot be represented by less than* $2w - 4$ *Double Masks.*

Proof: First note that no mask can cover a set of addresses with non empty intersection with both $l(0)$ and $l(1)$. Therefore we have to add the minimal number of masks for covering $[3, 2^{w-1} - 1]$ and the minimal number of masks for covering $[2^{w-1}, 2^w - 4]$. Address 3 cannot be covered by a mask $s/p/k$ with $p < w - 1$: otherwise, if $k > 0$ only a unique perfect subrange $l(s|p+k)$ would be excluded, but $[0, 2]$ is composed of two perfect subranges, contradiction ; if $k = 0$ then $l(s|p)$ would contains address 2, contradiction. Hence address 3 can be covered only by $0^{w-2}11/w/0$ or $0^{w-1}10/w-1/1$. In the same way, no address between 3 and $2^{w-1} - 1$ can be covered by a *Double Mask.* By reasoning as in Example 2 we can also show that two addresses of type $2^{w'-1} - 1$ with $3 < 2^{w'-1} - 1 \leq 2^{w-1} - 1$ cannot be covered by the same *Simple Mask.* As a consequent the minimal number of masks needed to cover $[3, 2^{w-1} - 1]$ is $w - 2$. By symmetry this is also true for $[2^{w-1}, 2^w - 4]$. The total number of masks is therefore $2w - 4$. $\square$

From Proposition 2 and Proposition 3, we can easily see that the $2w-4$ bound is tight.

3.3.2 Linear Time Algorithm

This section introduces a more efficient algorithm, named *DoubleMasks*, to compute a set of masks covering any range $[a, b]$. The new designed algorithm is linear in k where k is the number of bits to represent an IP address. Given two binary strings u, v we write $u \prec v$ (resp. $u \preceq v$) when u is a strict prefix (resp. prefix) of v. We denote by $prec(p)$ the longest proper suffix of p. Recall that $u < v$ indicates that the natural number denoted by u is smaller than the one denoted by v. We assume that $bin_w(a) = ca', bin_w(b) = cb'$ where c is the longest common prefix of $bin_w(a)$ and $bin_w(b)$.

DM-Naive (Algorithm 1) processes all nodes on paths from the root to leaves with value in $[a, b]$. Hence the number of processed nodes can be exponential in w. Unlike DM-Naive, *DoubleMasks* (Algo. 3) only processes nodes p in paths leading to leaves with value a or b, i.e., *DoubleMasks* examines only two branches in the tree $t(\varepsilon)$. Fig. 3.4 depicts the idea behind the algorithm. *DoubleMasks* works in two phases. The algorithm computes first for each node p a set of masks for $l(p) \cap [a, b]$, in a bottom up way and starting from the two nodes $bin_w(a)$ and $bin_w(b)$. Then, when reaching node c, the set of masks computed at the siblings of c (i.e., $c0$ and $c1$) are combined and the algorithm stops. This strategy is justified by the following Fact 1:

Fact 1. *Let c be the longest common prefix of $bin_w(a)$ and $bin_w(b)$. Interval $[a, b]$ is the disjoint union of $[a, val_w(c01^{w-|c|-1})]$ and $[val_w(c10^{w-|c|-1}), b]$.*

Now we introduce *ComputeMasks*, a procedure that computes the *Double Mask DM* representation of each subinterval in Fact 1.
ComputeMasks has a parameter x that will be successively substituted by a and b in the main algorithm *DoubleMasks*. The Boolean parameter β is chosen such that $c\beta$ is a prefix of x. If $x < val_w(c\beta\bar{\beta}^{w-|c|-1})$ (resp. $x > val_w(c\beta\bar{\beta}^{w-|c|-1})$), the algorithm computes a *DM* representation of range $[x, val_w(c\beta\bar{\beta}^{w-|c|-1})]$ (resp. $[val_w(c\beta\bar{\beta}^{w-|c|-1}), x]$).
ComputeMasks relies on the following case analysis:

Case 1: $c \prec p\beta \preceq x$:

- **Case 1.1:** if $DM_{p\beta} = \{p\beta\bar{\beta}s/|p|+2/0\}$, then $DM_p = \{p\beta\beta s/|p|/2\}$ (DM generated)
- **Case 1.2:** if $DM_{p\beta} = \{p\beta s/|p|+1/k\}$, then $DM_p = \{p\beta s/|p|/k+1\}$, since $l(p\bar{\beta}) \subseteq [a, b]$ (DM extended)
- **Case 1.3:** if $DM_{p\beta} = \{p\beta s/|p|+1/0\}$, then $DM_p = \{p\beta s/|p|/0\}$, since $l(p\bar{\beta}) \subseteq [a, b]$ and $DM_{p\beta}$ is a *Simple Mask* (SM Extended)
- **Case 1.4:** otherwise $DM_p = DM_{p\beta} \cup \{p\bar{\beta}s/|p|+1/0\}$, since $l(p\bar{\beta}) \subseteq [a, b]$ (SM added)

Case 2: if $c \prec p\bar{\beta} \preceq x$, then $DM_p = DM_{p\bar{\beta}}$ (masks maintained)

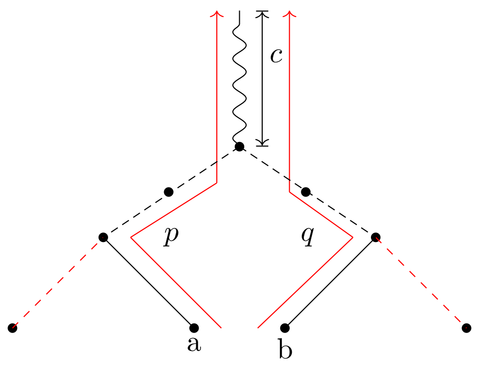


Figure 3.4: Illustration of *DoubleMasks* strategy.

Now we detail the auxiliary procedure *ComputeMasks* and the main procedure *DoubleMasks*.

ComputeMasks (Algo. 2)

This algorithm takes a binary number x such that $c\beta \prec x$ and returns a set of masks representing the interval between x and $val_v(c\bar{\beta}^{w-|c|})$. First we add the *Simple Mask* corresponding to x (Line 3). Then, we proceed on all prefixes of x from the longest one (Lines 4-20). For each prefix, the algorithm checks the type of the previously computed mask. If this mask contains a $mask1$ of length $|p| + 2$ (corresponding to a perfect tree of height $|p| + 2$) a new *Double Mask* is generated (Lines 7-8). If a *Double Mask* is present, this *Double Mask* will be extended (Lines 9-10). If the mask computed before has a $mask1$ of length $|p| + 1$ the same mask will be extended (Lines 11-12). If neither of the previous cases holds, the algorithm adds a new mask to the set of masks computed before (Lines 13-14).

The proof of correctness of $ComputeMasks$ is by induction on $w - |p|$ and checks whether DM_p covers $l(p) \cap [a, b]$, as for $DM - Naive$. Therefore the result of $ComputeMasks(a, c, 0)$ (resp. $ComputeMasks(b, c, 0)$) is a DM representation of $l(c0) \cap [a, b]$ (resp. $l(c1) \cap [a, b]$).

Algorithm 2 ComputeMasks(x,c,β)

```
1:  Input: $x, c, \beta$ such that $c\beta \prec x$
2:  Output: set of masks $DM_{c\beta}$
3:  $p \leftarrow prec(x); DM_x \leftarrow \{x/w/0\}$                    ▷processing path $x$
4:  while $c \prec p$ do
5:    if $case$ 1 then
6:      switch $(DM_{p\beta})$
7:      case 1.1 $= \{p\beta\bar{\beta}s/|p| + 2/0\}$:
8:        $DM_p \leftarrow \{p\beta\beta s/|p|/2\}$                    ▷new DM generated
9:      case 1.2 $= \{p\beta s/|p| + 1/k\}$:
10:       $DM_p \leftarrow \{p\beta s/|p|/k + 1\}$                     ▷DM extended
11:     case 1.3 $= \{p\beta s/|p| + 1/0\}$:
12:       $DM_p \leftarrow \{p\beta s/|p|/0\}$                         ▷SM extended
13:     default:
14:       $DM_p \leftarrow DM_{p\beta} \cup \{p\bar{\beta}\beta^{|w|-|p|-1}/|p| + 1/0\}$▷SM added - Case 1.4
15:     end switch
16:   else
17:     $case$ 2
18:   end if
19:   $p \leftarrow prec(p)$                               ▷process parent node on the path
20: end while
21: return $DM_p$
```

DoubleMasks (Algo. 3)

This algorithm takes as input an interval $[a, b]$ and computes a set of masks representing it.

Algorithm 3 DoubleMasks(a,b)

```
Input: $a, b$
Output: set of masks representing $[a, b]$
$c \leftarrow$ longest common prefix of $a$ and $b$
$p \leftarrow c0, q \leftarrow c1$          ▷final result will be computed from siblings of $c$
$DM_p \leftarrow ComputeMasks(a, c, 0)$          ▷refer to Algo. 2
$DM_q \leftarrow ComputeMasks(b, c, 1)$          ▷refer to Algo. 2
if $DM_p = \{pr/|p|/0\}$ then
    if $DM_q = \{qs/|q|/0\}$ then
        return $\{cr/|c|/0\}$          ▷new SM generated
    else if $DM_q = \{c\{1\}^{w-|c|}/|q|/|s|\}$ then
        return $\{c1s/|q| - 1/|s| + 1\}$          ▷DM extended
    else if $DM_q = \{qs/|q| + 1/0\}$ then
        return $\{c11t/|c|/2\}$          ▷new DM generated
    else
        return $DM_p \cup DM_q$
    end if
else if $DM_q = \{qs/|q|/0\}$ then
    if $DM_p = \{c\{0\}^{w-|c|}/|p|/|s|\}$ then
        return $\{c0s/|p| - 1/|s| + 1\}$          ▷DM extended
    else if $DM_p = \{ps/|p| + 1/0\}$ then
        return $\{c00t/|c|/2\}$          ▷new DM generated
    else
        return $DM_p \cup DM_q$
    end if
else
    return $DM_p \cup DM_q$
end if
```

First, the algorithm computes the common prefix c of a and b (Line 3). Then, according to Fact 1, interval $[a, b]$ can be divided into $[a, val_w(c01^{w-|c|-1})]$ and $[val_w(c10^{w-|c|-1}), b]$. $ComputeMasks$ is called for each subinterval (Lines 5-6). The final result depends on the sets of masks DM_p and DM_q generated by Algo. 2. If $l(c0), l(c1) \subseteq [a, b]$ a new *Simple Mask* is generated (Lines 7-9). If a $DoubleMask$ is generated for $t(c1)$ (resp. $t(c0)$) , the $DoubleMask$ will be extended (Lines 10-11) (resp. Lines 18-19). If $l(c0)$ (resp. $l(c1)$) is covered by a *Simple Mask* of length $|p|$ (resp. $|q|$) and $l(c1)$ (resp. $l(c0)$) is covered by a mask of length $|q| + 1$ (resp. $|p| + 1$), then a new $DoubleMask$ will be generated (Lines 12-13) (resp. Lines 20-21). If not, the algorithm returns the union of the two parts.

DoubleMasks computes a DM representation of $l(c) \cap [a, b]$ from DM representations of $l(c0) \cap [a, b]$ and $l(c1) \cap [a, b]$ obtained by calling *ComputeMasks*. We can stop when reaching c in the main "while" loop of *ComputeMasks* and return the result DM_c since we can see easily that $l(c) \cap [a, b] = l(\varepsilon) \cap [a, b] = [a, b]$.

3.4 Evaluation by Simulation

We evaluate the performance of the Algorithm *Double Mask* (DM), and we compare it with an algorithm that only generates *Simple Masks* (SM). This algorithm is obtained by a simple modification of *Double Mask*. We conducted experiments using two types of data sets. The first dataset is a real-world IP ruleset downloaded from the repository [121]. The second ruleset is a list of synthetically generated IP addresses.

3.4.1 Simulation Setup

The real-world ruleset contains more than 133 million IP addresses. Therefore, we first transform this set of IPs into ranges. Then we compare the effects of a *Double Mask* representation w.r.t. a *Simple Mask* representation in reducing the size of our ruleset. To generate *Double Masks* we rely on Algo. 3 and to generate *Simple Masks* we rely on a simple modification of the same algorithm.

The two algorithms were coded in Java language and the experiments are carried on a desktop computer with Intel Core i7-7700 3.6-GHz CPU, 32 GB of RAM, and running Windows 10 operating system.

We define the following metric for analysing the performance of the two algorithms:

$$\text{Average Compression Ratio} = 1 - \frac{M}{n * S}$$

where

M is the number of masks generated in all iterations,

S is the number of IPs in the ruleset,

n denotes the number of iterations.

To compute the average compression ratio, the number of iterations is set to 20. We use this metric to show that our algorithm can generate a more compact list of rules in comparison with *Simple Mask* algorithm. We also compute the total number of masks generated with each algorithm.

3.4.2 Real-world IP Ruleset

The ruleset IPs are aggregated into approximately 11K ranges. To have much larger ranges, the two algorithms will take as input all the ranges located between the set of ranges computed previously. The two programs take as input each range and compute a set of masks covering this range.

Fig. 3.5 compares the number of masks generated by the two algorithms. By using *Double Masks* representation, we can reduce the number of masks by more than 18%. The total number of generated masks using *Simple Mask* is 18118. Using *Double Mask*, the number is reduced to 14919 masks. In total, 15.4% of all generated masks are *Double Masks* (i.e. 2301 DM). As the number of ranges increases, we observe that Algo. 3 generates fewer masks than *Simple Mask* algorithm.

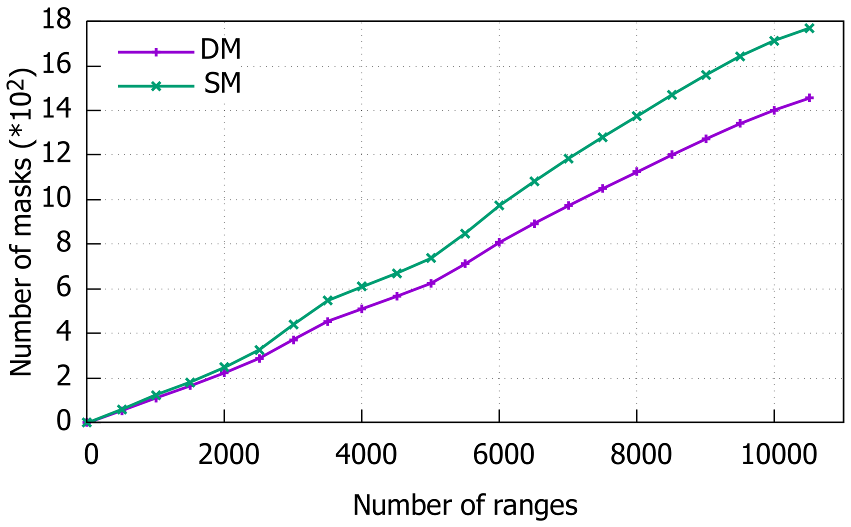


Figure 3.5: Number of masks generated respectively by *Double Mask* (DM) and *Simple Mask* (SM) algorithms using the real-world IP ruleset.

3.4.3 Synthetically Generated Rulesets

In the second experiment, we evaluated more than 6000 ranges computed from more than 1.5 million IPs obtained synthetically. Fig. 3.6 shows the difference between the total number of masks computed respectively by the two algorithms. In this scenario, we observe a significant difference between simple and *Double Mask* techniques. The total number of generated *Simple Masks* is 29958. Using *Double Mask* algorithm, we are able to reduce this number by 74% (i.e., 7872 masks). The synthetic ruleset used in Fig. 3.6 contains a higher number of ranges of the form $[1, 2^w - 1]$ which explains the difference between the obtained number of double and

Simple Masks.

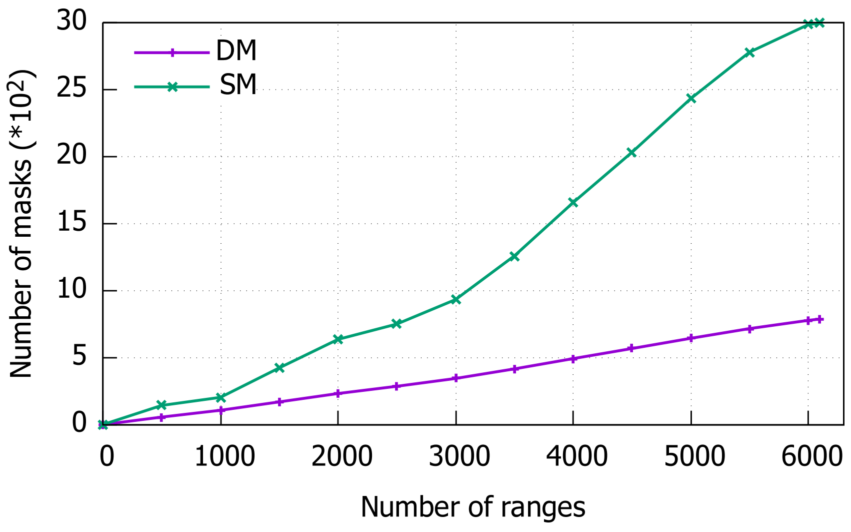


Figure 3.6: Number of masks generated respectively by *Double Mask* (DM) and *Simple Mask* (SM) algorithms using the synthetic ruleset.

Fig. 3.7 shows the average compression ratio of the two algorithms while increasing the number of IPs. We observe, that *Double Mask* algorithm performs better than *Simple Mask* with a difference of at least 10%.

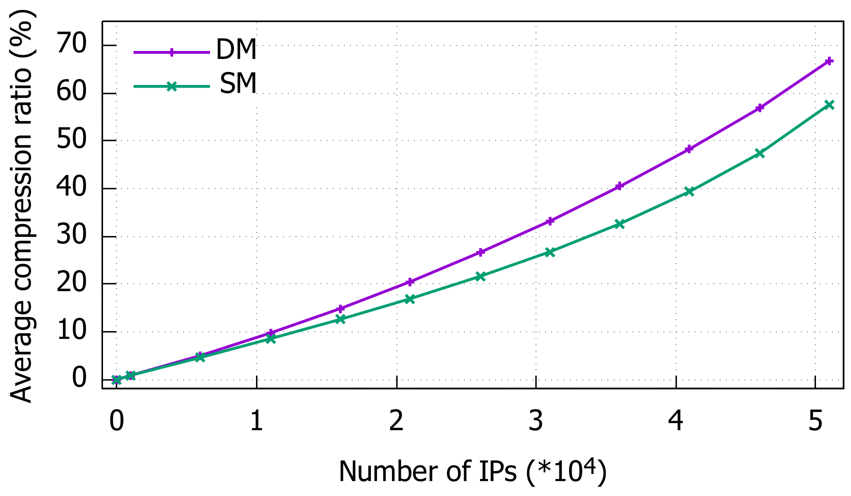


Figure 3.7: Compression ratio of *Double Mask* (DM) and *Simple Mask* (SM) using a synthetic ruleset of range fields of length 16bits.

Fig. 3.8 shows the difference in compression ratio between *Double Mask* and *Simple Mask* while modifying the length of IPs field. Furthermore, we observe that *Double Mask* algorithm always performs better than *Simple Mask* for each length value.

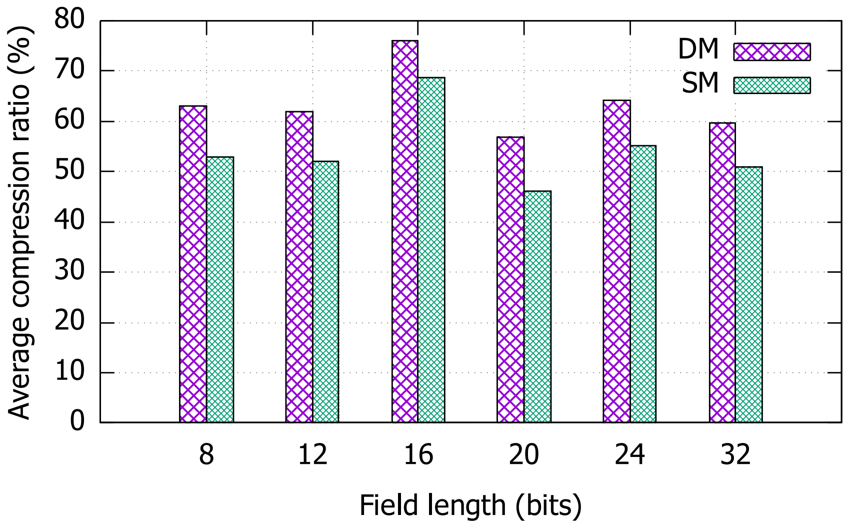


Figure 3.8: Comparison of compressions ratio between *Double Mask* (DM) and *Simple Mask* (SM) algorithms while varying the length of a field.

The compression ratio depends on each ruleset and on the nature of IPs ranges. We use two types of rulesets to demonstrate that this technique can reduce the number of rules by 79% and more in some cases and by 18% or less in others, depending on the nature of IP ranges. Since *Double Mask* algorithm generates a *Simple Mask* when no *Double Mask* can be generated, the total number of masks will be at most equal to the number of *Simple Masks* computed by *Simple Mask*. This is why, according to our empirical simulations, *Simple Mask* cannot generate a smaller set of masks than *Double Mask*.

3.5 Experimental Evaluation

We implement the *Double Masks* representation and its respective matching algorithm using an OpenFlow switch and an SDN controller. We evaluate and compare the performance of our matching algorithm with a *Simple Masks* representation which is considered as a baseline.

3.5.1 Setup and Parameters

In our experimental setup, we use the physical SDN testbed shown in Fig. 3.9.

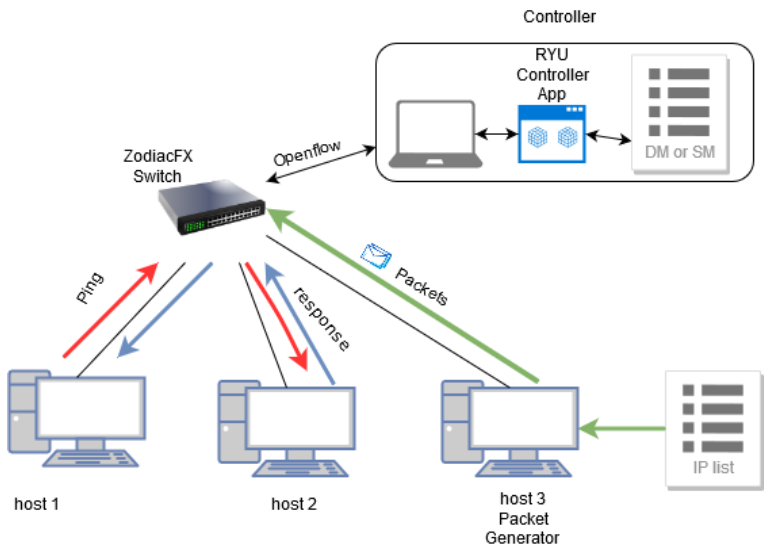


Figure 3.9: The experimental physical SDN testbed.

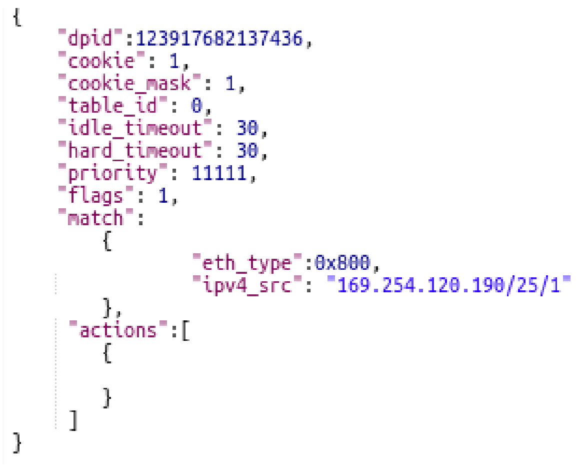
The testbed contains a Zodiac FX switch [122] connected to a RYU controller [123] and three hosts. The code in the controller and the switch has been modified with *Double Masks* representation for IP matching fields. A matching function is added in both of them to match a received packet with the set of rules in the routing table of the switch or with a list of rules in the controller. In this scenario, a host machine (Host 3) connected to the switch takes a list of IP addresses as input and sends packets to those destinations. Another host (Host 1) tries to send a ping message to Host 2. The ping message will match all rules in the switch's routing table before forwarding the packet to Host 2. If no match is found, the message will be sent to the controller, who matches the message with the list of rules then send back the action needed for the specific packet to the switch. The round trip time (RTT) from host 1 to Host 2 is recorded.

3.5.2 Implementation and Integration

Integration in the Controller

The code of the RYU controller has been modified to integrate the *Double Masks* representation in the OpenFlow protocol match fields. The controller takes a set of rules (simple or *Double Masks*) computed from a blacklist using our transformation algorithm. The controller matches the header of the OpenFlow *PACKET_IN* messages with the set of rules to block or not the traffic to specific destinations.

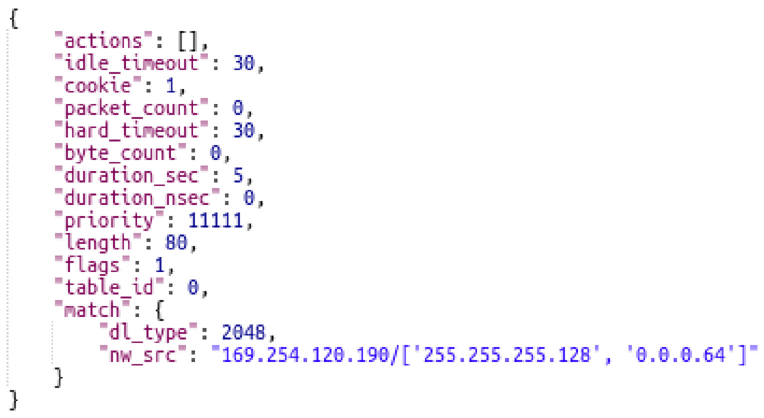
As depicted in Figure 3.10, when a rule is being sent to the controller containing a *Double Mask* represented field (i.e. 169.254.120.190/25/1, action field empty = [Drop]), the controller add it to its flow table as shown in Figure 3.11. This message also contains the switch id, the timeout, and the priority or the rule. When a packet matches multiple rules, only the one with the highest priority will be applied.



```
{
    "dpid":123917682137436,
    "cookie": 1,
    "cookie_mask": 1,
    "table_id": 0,
    "idle_timeout": 30,
    "hard_timeout": 30,
    "priority": 11111,
    "flags": 1,
    "match":
        {
                "eth_type":0x800,
                "ipv4_src": "169.254.120.190/25/1"
        },
     "actions":[
        {

        }
     ]
}
```

Figure 3.10: Illustration of OpenFlow rule sent to the controller through the REST API.



```
{
    "actions": [],
    "idle_timeout": 30,
    "cookie": 1,
    "packet_count": 0,
    "hard_timeout": 30,
    "byte_count": 0,
    "duration_sec": 5,
    "duration_nsec": 0,
    "priority": 11111,
    "length": 80,
    "flags": 1,
    "table_id": 0,
    "match": {
        "dl_type": 2048,
        "nw_src": "169.254.120.190/['255.255.255.128', '0.0.0.64']"
    }
}
```

Figure 3.11: The rule with *Double Mask* is inserted in the controller flow table.

Integration in Zodiac FX Switch

In our setup, we use the OpenFlow-enabled Zodiac FX switch shown in Fig. 3.12 that provides an inexpensive alternative to experiment SDN networks in hardware. The OpenFlow implementation of the Zodiac FX switch has been modified to integrate the processing of rules by using the *Double Masks* representation. This code has also been modified to apply a matching between the IP source of a packet and all the rules in the switch.

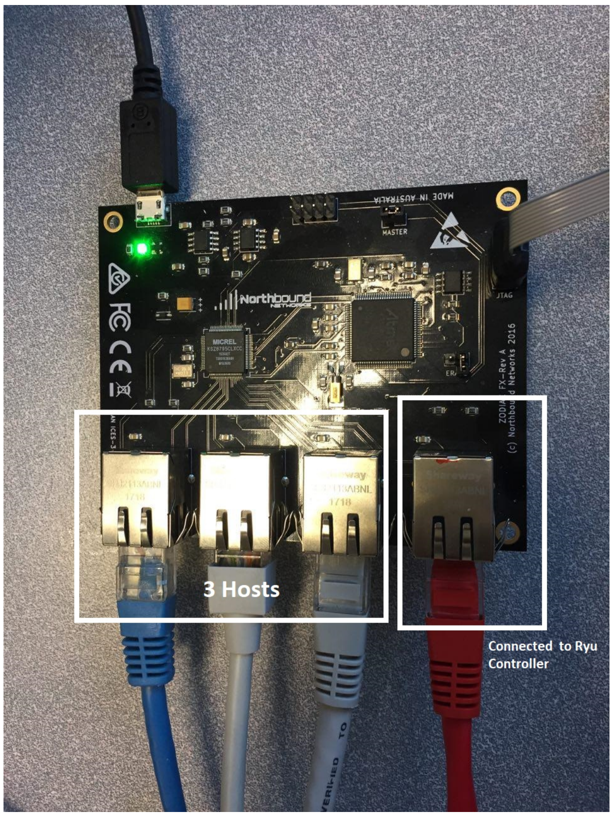


Figure 3.12: OpenFlow-enabled Zodiac FX switch

The routing table of our Zodiac FX switch can store a maximum number of 448 rules using simple or *Double Masks*. The timeout for each rule is set to 30 seconds. After that, the rule will be automatically removed from the table. The switch has two matching algorithms for matching a packet. The first algorithm is the standard algorithm used to match a received packet with a *Simple Mask* rule in the routing table, and the second one is used for the *Double Mask* rules.

Algorithm 4 implemented both in the switch, and the controller applies the double mask matching process between a source IP and a rule in the flow table. For example, if the value of $S0$ is zero, that means the IP source matches the network IP. If the value of $S1$ is different from zero, that means the IP source is not included in the set of rejected IPs by $mask2$, in this case, the IP source will match the rule.

Fig. 3.13 shows the logical representation for the simple matching function used for the standard CIDR notation. While Fig. 3.14 shows the logical representation of our matching function for *Double Mask* represen-

Algorithm 4 Matching($netref, mask1, mask2, ip\_source$)

```
1:  Input: $netref, mask1, mask2, ip\_source$
2:  Output: $accept$ or $deny$
3:  $X1 \leftarrow mask1 \wedge netref$
4:  $X2 \leftarrow X1 \oplus ip\_source$
5:  $S0 \leftarrow X2 \wedge mask1$
6:  if $S0 = 0$ then
7:                                   ▷ip_source match the ip of the network
8:    $X3 \leftarrow mask1 \vee mask2$
9:    $X4 \leftarrow X3 \wedge ip\_source$
10:   $X5 \leftarrow X3 \wedge netref$
11:   $S1 \leftarrow X4 \oplus X5$
12:   if $S1 \# 0$ then
13:                          ▷ip_source not included in IPs rejected by mask2
14:     return $accept$
15:   else
16:     return $deny$
17:   end if
18: else
19:   return $deny$
20: end if
```

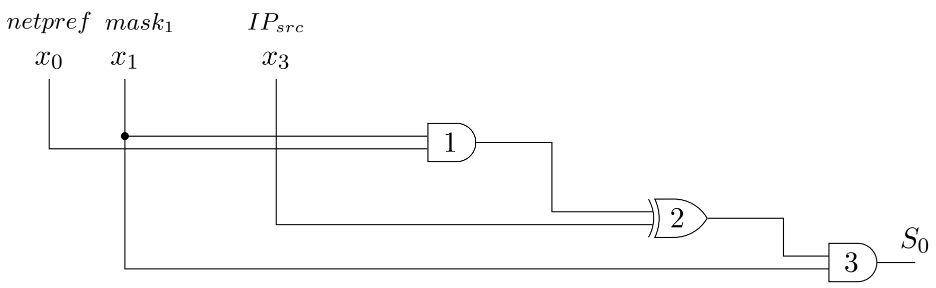


Figure 3.13: Logical representation of the simple matching function for CIDR notation.

tation implemented in both the switch and the controller with the added part in red. This new function has four inputs and two outputs.

Let us take the rule with a *Double Mask* notation (192.168.100.0/24/6−− *accept*) as an example. This representation means that all IP addresses between 192.168.100.5 and 192.168.100.255 are accepted while addresses between 192.168.100.0 and 192.168.100.4 are rejected. And let p a packet with an address 192.168.100.1. In this case, x_0 is used to represent $netpref$ or 192.168.100.0. x_1 is used to represent $mask1$ and is equal to 255.255.255.0 (i.e. the first 24 bits) while x_2 represent $mask2$ and is equal to 0.0.0.252 (i.e. the next 6 bits after $mask1$). Finally, x_3 will be equal to

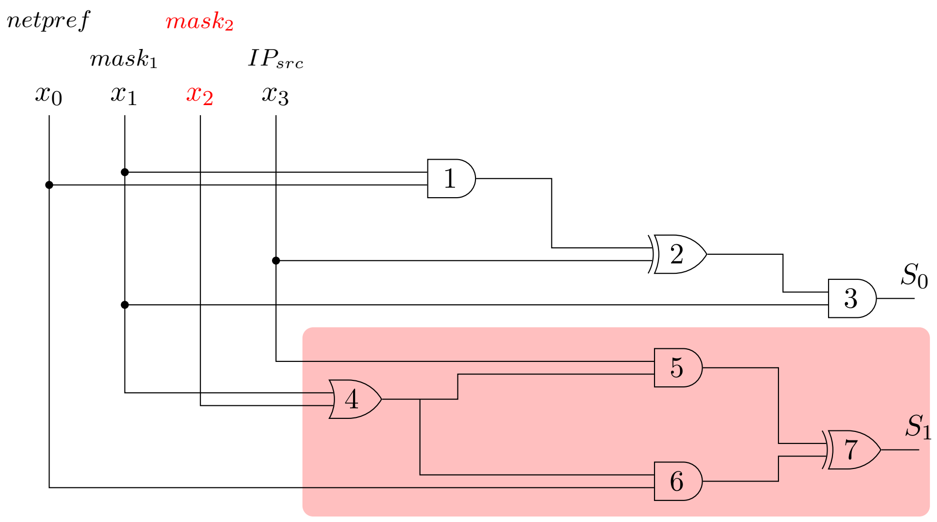


Figure 3.14: Logical representation of the matching function for *Double Mask*.

the IP address of p (i.e. 192.168.100.1). Since the address of p is a part of the accepted IPs of the network represented by $mask1$ the value of S_0 will be equal to 0. S_2 will be equal to 0 since the IP of p is included in the rejected part represented by $mask2$ and the packet will not match the rule.

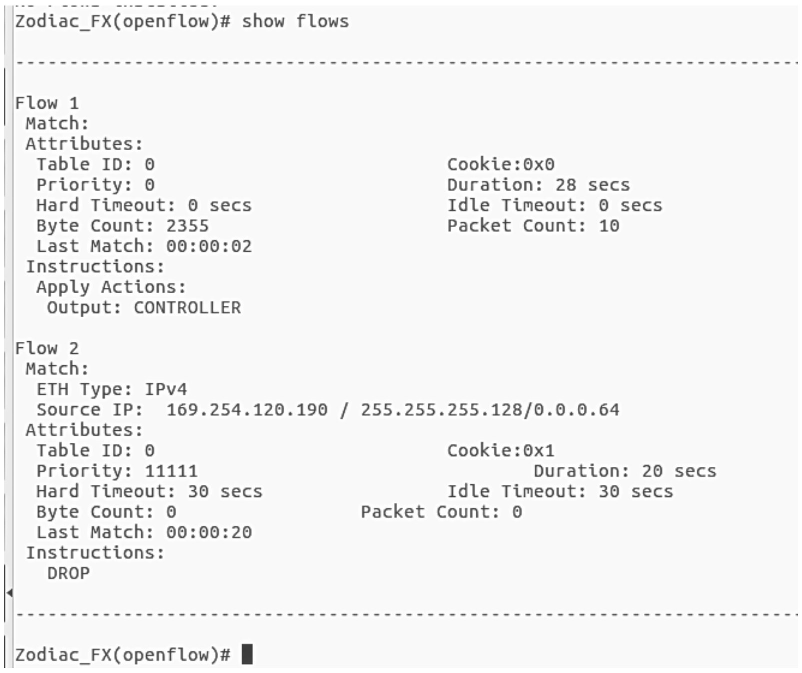
Table. 3.2 shows all the different combinations from the output of the logical circuit in Fig. 3.14. An IP can match a *Double Mask* rule if the output of S_0 is equal to 0, meaning that the IP of the packet matches the accepted part of the rule, and if S_1 is different from 0, meaning that the IP does not belong to the rejected part of the network.

S_0	S_1	**Match**
0	0	no
0	$\neq 0$	yes
$\neq 0$	0	no
$\neq 0$	$\neq 0$	no

Table 3.2: All matching outputs from the logical circuit in in Fig. 3.14.

Although the logical circuit for matching *Double Mask* is bigger than *Simple Mask*, we hope that the compression ratio can compensate for the additional time. To test our theory, we need to run experiments on the real testbed shown in Fig. 3.9 that supports *Double Mask*.

When the controller sends an OpenFlow message with the match field represented in *Double Mask* notation to the Zodiac FX switch. The switch adds the rule to its flow table as shown in Fig. 3.15. When a packet is received, for example, from a host with an IP 169.254.130.226, the switch performs matching between the receiving packet and the rules in the flow table. Since the source IP belongs to the network 169.254.130.128/25, the value of S0 in the matching Algorithm 4 will be 0. However, the value of S1 will be different from 0 since the IP of the source is not included in the set of IPs rejected by the *Double Mask* (i.e., All IPs on the network 169.254.130.128/25 except those with a bit 0 in the 26 places will be dropped). In this case, the switch finds a match between the received packet and the rule in the flow table, and the action associated with the rule will be performed.



```
Zodiac_FX(openflow)# show flows

-------------------------------------------------------------------

Flow 1
 Match:
 Attributes:
  Table ID: 0                          Cookie:0x0
  Priority: 0                          Duration: 28 secs
  Hard Timeout: 0 secs                 Idle Timeout: 0 secs
  Byte Count: 2355                     Packet Count: 10
  Last Match: 00:00:02
 Instructions:
  Apply Actions:
   Output: CONTROLLER

Flow 2
 Match:
  ETH Type: IPv4
  Source IP:  169.254.120.190 / 255.255.255.128/0.0.0.64
 Attributes:
  Table ID: 0                          Cookie:0x1
  Priority: 11111                              Duration: 20 secs
  Hard Timeout: 30 secs                Idle Timeout: 30 secs
  Byte Count: 0                Packet Count: 0
  Last Match: 00:00:20
 Instructions:
   DROP

-------------------------------------------------------------------

Zodiac_FX(openflow)#
```

Figure 3.15: The flow table of the Zodiac FX after inserting a double-mask based rule.

3.5.3 Experiments and Results

We use the testbed depicted in Figure 3.9 to validate the applicability of the double-mask representation in an OpenFlow network and to evaluate its performance. To generate our evaluation workload, we developed a packet generator that takes a list of IP addresses and then sends packets to each address in the list. We set the packet rate for each experiment to be 9, 12, or 15 packets/second. The packets are being sent to different destinations based on multiple rulesets. The number of packets per second is chosen so that the time needed for the saturation of the switch routing table is around 10 min. If the number of packets is too small, the table will never reach the maximum number of 448 rules, and if it is too large, the table will max out quickly. We repeat each experiment 6 times on 6 different rulesets then we compute the average matching time. The same rulesets are used to generate the different packets to match the sets of simple and *Double Masks* rules.

Matching Time in Switch

In these experiments, the average matching time is computed after using a simple or a *Double Masks* list. Our goal is to study the impact of the compression ratio on the matching time in the switch. We will use two sets of rules the first one with a compression ratio of 5% and the second one with 30%.

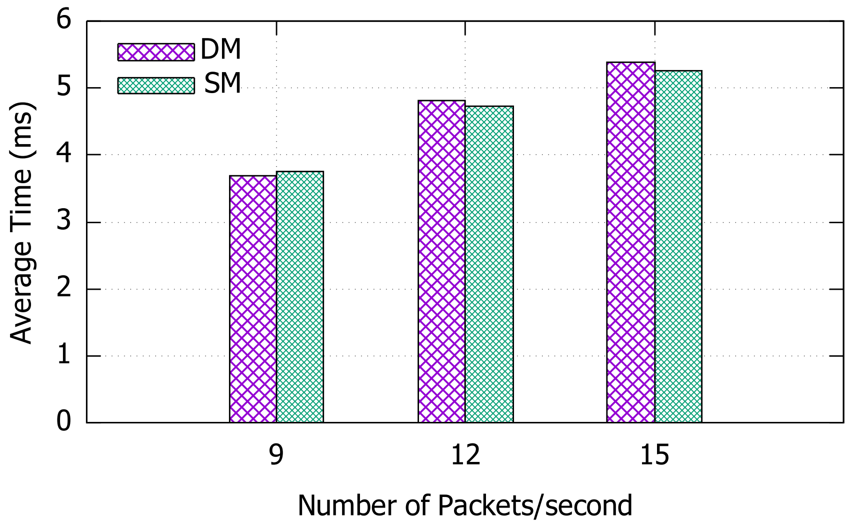


Figure 3.16: The difference in the average time for matching between using simple or *Double Masks* with a 30% compression ratio.

Figures 3.17 and 3.16 show the difference in the average matching time between simple and *Double Masks* while varying the number of the packets generated at Host 3. As shown in the two figures, the average matching time with simple and *Double Masks* are very close. Matching with a *Double Mask* is more costly than with a *Simple Mask*. However, this increase in time is compensated by a smaller number of rules in the routing table while using a *Double Mask*.

To see the real impact of the compression ratio on the global matching time, we evaluate the response time on the controller side using the two matching functions for simple and *Double Masks* with multiple sets of rules with a different compression ratio.

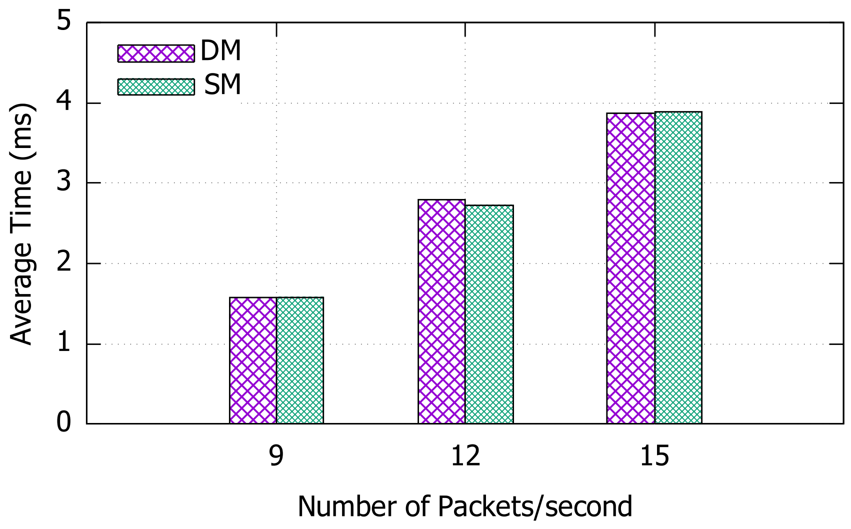


Figure 3.17: The difference in the average time for matching between using simple or *Double Masks* with a 5% compression ratio.

Matching Time in Controller

In a second experiment, we compare the matching time between a list of simple or *Double Masks* in the controller side. Our experiment uses 7 IP blacklists. Each blacklist generates two lists of rules, one with only *Simple Masks* and the other with both simple and *Double Masks*. The compression ratio for the different sets of lists varies between 0.5% and 83%. The goal here is to evaluate the effect of the compression ratio on the controller's response time. A set of 300K IPs is used to match each IP address with each rule for the different sets. We use this number of IPs to simulate heavy traffic to show the gain in response time.

First, we compute the response time of the controller using only *Simple Mask* rules. Then we compute the response time using the same sets of IPs on the second set of rules that uses simple and *Double Masks*. As shown in Fig. 3.18, when the compression ratio is at 0.5% it is better to use a *Simple Mask* over a *Double Mask* since the gain in space is low in comparison with what we lose by using the matching function for the *Double Mask* that takes more time than the matching time for *Simple Mask*. At 15%, we can see that the time needed for matching *Double* or *Simple Mask* is similar. When the compression ratio is higher than 15%, we obtain substantial gain in response time by using *Double Masks*. From a 30% compression ratio, the results show a gain in the controller's matching time. However, at the same ratio, the gain in space is obtained in the switch and the controller.

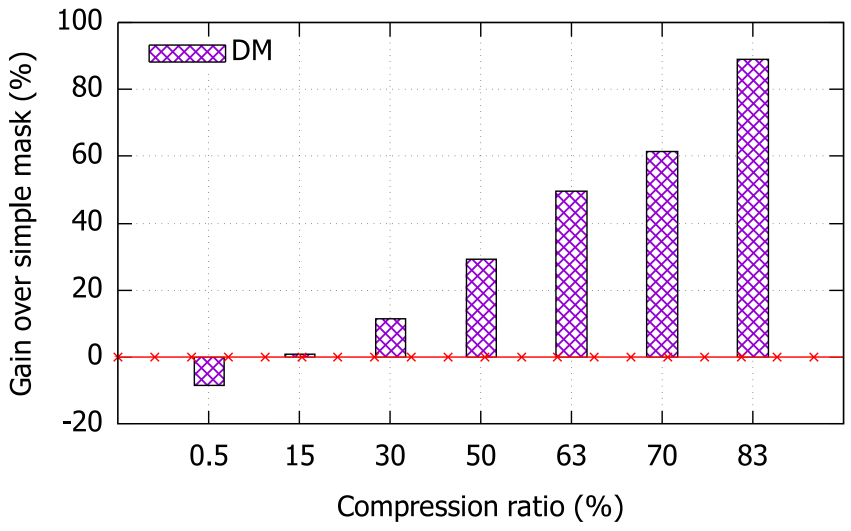


Figure 3.18: Gain in the response time in the controller side when using *Double Masks* filters.

3.6 Discussions

As shown in previous sections, our linear algorithm computes a set of masks covering a range given as input. If the range is not perfect, a *Double Mask* is generated. Moreover, in this technique, each set of masks for a range is independent of other sets belonging to other ranges. However, if we compute the union of ranges, we can achieve a higher compression ratio.

For example, let us consider the example in Fig. 3.19 with two perfect ranges $[a, b]$ and $[e.f]$. Normally, since each range is independent, our algorithm return all masks covering the two ranges. In this case *Simple Mask* $= bin_v(a)/|r| + 1$ for range $[a, b]$, and *Double Mask* $= bin_v(c)/|r| + 1/|s| - 1$ for range $[e.f]$. However, one *Double Mask* $= bin_v(c)/|r|/|s|$ can be generated to cover the two ranges.

3.7 Summary

The *Double Masks* is a new representation used to reduce the number of rules in firewalls, IDS's or routing tables to make their configuration, management, and deployment easier. In this chapter, we formally propose the first linear algorithm to compute a set of *Double Masks* covering a range of IPs. Note that our algorithm can be applied after or in combination with known redundancy removal techniques like [52] to further reduce the number of entries in filtering rule tables. Then we conducted a series of experiments on real and synthetic rulesets. According to our experiments,

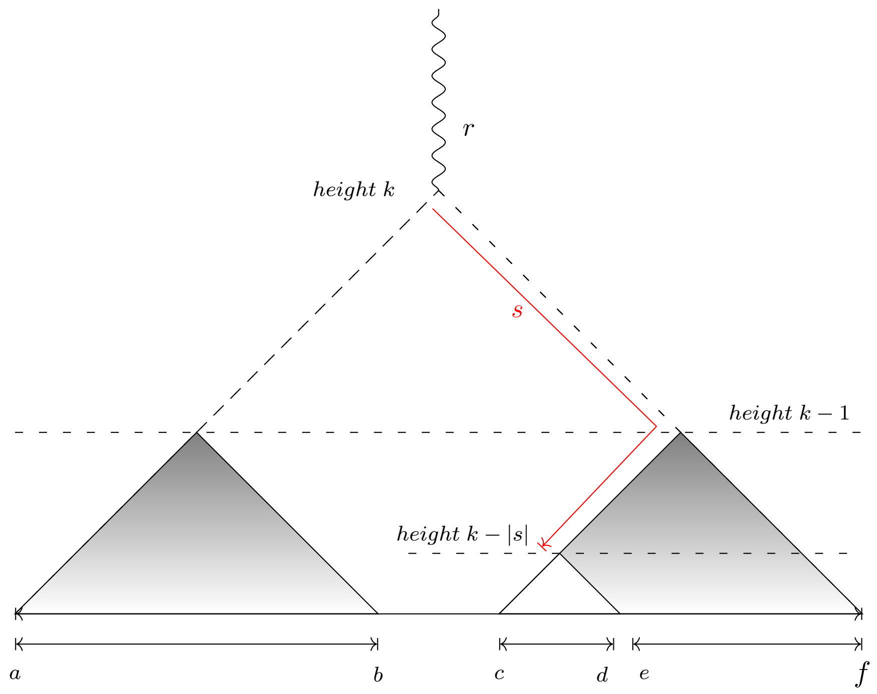


Figure 3.19: Extending Double Mask

using the *Double Mask* representation allows one to reduce the number of rules needed to cover a set of ranges by more than 18% on a real-life ruleset (after removing the redundant rules) and more than 74% on synthetic data. The algorithm is not limited to IP ranges, and it can be applied to port ranges too and to reduce the range expansions in TCAM. We also evaluate the effectiveness of *Double Masks* using an OpenFlow based implementation and evaluate its matching time using a physical SDN testbed. Although we obtain a similar matching time in the switch for the simple and *Double Masks* representations, the storage space of rules is reduced. On the controller side, with compression ratios higher than 15% we observe a substantial gain in the matching and response times. However, compression techniques cannot always be efficient. In some cases, the size of switches is too small to fit all rules even after compression. To deal with this problem, we will introduce in the following chapter a new approach for distributing rules over multiple switches.

Chapter 4

Rules Distribution Over a Single Path Topology

Contents

4.1 Introduction

The previous chapter tries to solve the rule management problem in each switch by relying on a new representation for the IP address field called *Double Mask*. However, the compression technique cannot always be efficient either because the compression ratio is minor or because the memory capacity of switches is small. Therefore, to deal with this problem, we can rely on software-defined networks (SDN) to distribute rules over multiple switches. In SDN, the filtering requirements for critical applications often vary according to flow changes and security policies. SDN addresses this issue with a flexible software abstraction, allowing simultaneous and convenient modification and implementing a network policy on flow-based switches. This single-point deployment approach constitutes an essential feature for complex network management operations.

The growing number of attacks from diverse sources increases the number of entries in access-control lists (ACL). To avoid relying on large and expensive memory capacities in network switches, a complementary approach to rule compression [13; 39; 41] would be to divide the ACLs in smaller switch tables to enforce the access-control policies. It paves the way towards distributed access-control policies, which have been the topic of many previous studies [86; 89; 90]. However, most of their proposals give rise to a large rules replication rate [86; 89]. Some proposal even needs to modify the header of the packet [90] to prevent it to match a second filter rule in a following switch.

This chapter is organised as follows. Section 4.2 introduces the general principles and distribution constraints in this problem, like switch capacity. Next, in Section 4.3 we introduce new techniques to distribute filtering rulesets over a single path topology. Our approach is to design distribution schemes for simple and complex policies that rely on single dimensions to forward packets to the destination. In Section 4.4 we evaluate the performance of our distribution algorithm. Section 4.5 summarizes and concludes this chapter.

4.2 Problem Statement

4.2.1 Problem Definition

We consider a network N with SDN-enabled switches that are connected to each other using their respective ports. Each of the switches can store in its flow table different types of access control and filtering rules. Every flow table has a limited capacity regarding the number of stored rules. We assume that an SDN policy is a collection of rules generated by an administrator or an external module. The size of the set of rules of the policy exceeds the capacity of a single switch table. In this chapter, we are mainly concerned with decomposing and distributing a set of filtering rules along with the switches in the network N to implement an SDN policy while preserving its general semantics and meeting the capacity limitation of each of the switches.

In the network N, packets are controlled by rules stored in their Forward Information Bases or flow tables. A rule is specified by a priority, matching patterns on packet fields and an action. A matching pattern of a source (resp. destination) field is given by a list of $0, 1$ followed by $*$'s (*don't care bits*), of global length w (word length) called a prefix p. Any bit matches wildcard character $*$. A rule with prefix p for the source (resp.

destination) applies to a packet if this packet source (resp. destination) field matches p, bit by bit. We call bit-prefix of the rule the sublist of $0, 1$ of p.

The matching process for a given set of rules can follow different strategies. A simple strategy prioritizes the rules by their order in the ruleset. For example, a packet matching the first rule will apply the action of that rule without considering the following ones. With a Longest Prefix Matching (LPM) strategy, one packet can match multiple rules, but only the one with the most specific matching prefix (i.e., the longest prefix) will be selected. In the example shown in Table 4.1, if a switch receives a packet with 0001 as an address, and using a prioritized list strategy, action A_1 of the first rule is applied. However, with an *LPM* strategy, the packet matches both Rules 1 and 2, but only action A_2 will be applied since Rule 2 covers the address field with a longer prefix.

Rule	Address field				*Action*
1	0	0	*	*	A_1
2	0	0	0	*	A_2

Table 4.1: Example of a ruleset in a switch table.

4.2.2 Requirements

To ensure that distributing the rules along different switches does not change the initial policy compared to a single-switch placement, we must preserve its overall semantics, i.e., *the action applied on a matched packet in a single switch with the initial policy ruleset should be the same action in a chain of switches with the distributed policy ruleset.* Here, we consider that filtering rules have two possible actions: "Forward" or "Deny". In the following, the prefix of a rule r is denoted by $Pref(r)$ and its action by $Action(r)$. If $Pref(r_1)$ matches $Pref(r_2)$ and has a shorter bit-prefix than $Pref(r_2)$, like Rule 1 with Rule 2 in Table 4.1, then we say that $r_1 > r_2$ or r_1 overlaps r_2. In the example of Table 4.1, it is also true that $Pref(r_1)$ is the next longest matching prefix i.e there is no r such that $r_1 > r > r_2$. We express this by $r_1 >: r_2$.

Let R be the initially given ruleset of a policy. Let R_1 and R_2 be two subsets of R located in different switches along a path, with R_2 being located in a switch after the one of R_1 as illustrated in Fig. 4.1.

When applying the *LPM* strategy in a switch, a packet must be processed by the most specific matching rule. Thus, **if $r_2 \in R_2$, there must not exist any $r_1 \in R_1$ such as $r_1 > r_2$**. To enforce this precedence property, when we place a rule r in a switch, we must also place in the same switch any rule r' such as $r > r'$.
When a packet has been accepted by a switch containing ruleset R_1 and

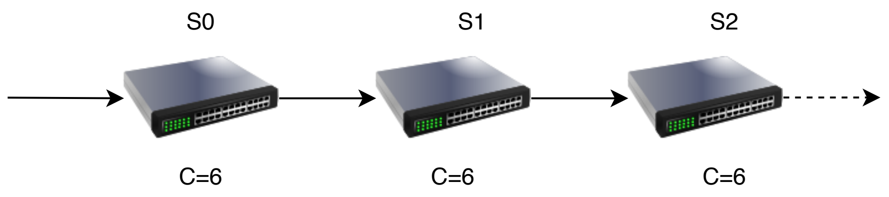


Figure 4.1: Flow table capacities of switches along a single path.

enters a switch containing ruleset R_2, if r_1 and r_2 are matching rules in R_1 and R_2 respectively, and $r_2 > r_1$ then r_2 should not block it. Thus, we must prevent blocking a packet when there is an action conflict. Given two rules $r_1 \in R_1$ and $r_2 \in R_2$, we have an action conflict iff $r_2 >: r_1$, $Action(r_1) = "Forward"$ and $Action(r_2) = "Deny"$.

In this thesis, we rely on forward rules to solve action conflicts: if two conflicting rules r_1 and r_2 belong respectively to two successive switch tables R_1 and R_2, a rule fw with $Action(fw) = "Forward"$ and $Pref(fw) = Pref(r_1)$ is added to R_2. When the policy preserving condition is respected, $Pref(fw)$ has a higher specificity, thus priority, than $Pref(r_2)$ by construction.

When decomposing and distributing R into subsets to be stored in the different switches and adding forward rules to solve action conflicts, the distribution problem is formulated as a Bin Packing problem with fragmentable items [124] where each fragmentation induces a cost.

4.3 Distribution Over a Single Path

Networks often have a blacklisting policy that specifies that packets originating from specific IPs are potentially harmful and need to be dropped, for instance, when handling denial of service attacks [125]. Thus, in a first step, we resolve the distribution problem with a single filtering field and by using an LPM strategy, which could be sufficient for placing a blacklisting policy in the network.

4.3.1 Rules Representation

We use a binary tree to represent the rules ordered by their prefix specificity. We study the single field case, so each rule contains one filtering field, and there is at most one rule associated with a tree node. The sequence of edge labels taken from the root to a node represents the bit prefix of the rule linked to this node.

As shown in Fig. 4.2, the pattern $00*$ of a rule r is at a distance of two from the root, reachable through the leftmost branch of the tree. The complete prefix with wildcards of a node n represents a rule pattern. It is

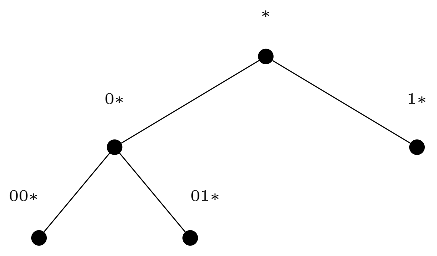


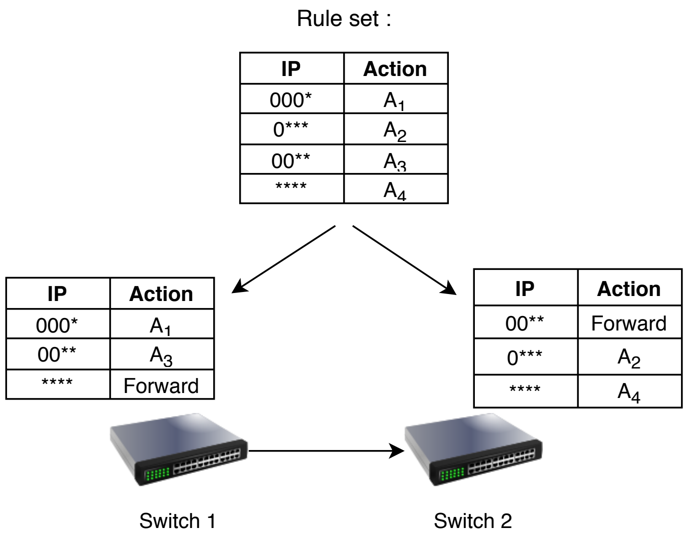
Figure 4.2: Compact representation of rules prefixes in a binary tree.

denoted as $NodePrefix(n)$ and is equal to $Pref(r)$, when a rule r is contained in n. We denote by $fw(n)$ the forward rule with a matching pattern $NodePrefix(n)$. Once the rules are distributed among the switches, if in any switch no matching rule is found for an incoming packet, the packet will be forwarded to the next switch using a default rule.

4.3.2 Forward Rules Generation

We assume the set of rules R has been distributed along successive switches $s_1, \dots s_n$ forming a path in the network graph. For this distribution to be correct with respect to the initial R and LPM semantics, we need to introduce additional forward rules in some switches.

The forward rule generation is illustrated in Fig. 4.3, where we install half of the rules in the first switch.



IP	Action
000*	A_1
0***	A_2
00**	A_3
****	A_4

IP	Action
000*	A_1
00**	A_3
****	Forward

IP	Action
00**	Forward
0***	A_2
****	A_4

Figure 4.3: Illustration of forward rule generation with a ruleset and two successive switches.

We have selected rules with 00** as a common prefix for the first switch, so accepted packets with an IP matching this pattern must be accepted in the second switch and forwarded. Thus, we have generated a forward rule with the prefix 00** in the second switch. A default forward rule must also be added to the first switch to allow the second one to receive any unmatched packets.

Given a set of rules R, two rule prefixes $p_1 = w0*, p_2 = w1*$ can be merged to obtain a forward rule prefix $p' = w*$. This operation can be iterated. Fig. 4.4 illustrates a rule tree with merging possibilities. This tree contains four rules with prefixes $00*, 01*, 1*, 10*$. We iteratively merge $00*$ and $01*$ to form a prefix $0*$, which is merged then with $1*$ into a common forward prefix. In this example, only one forward rule is needed for this set. As shown in Fig. 4.4, the forward rule covers the subtree with rules at every branch above the blue line.

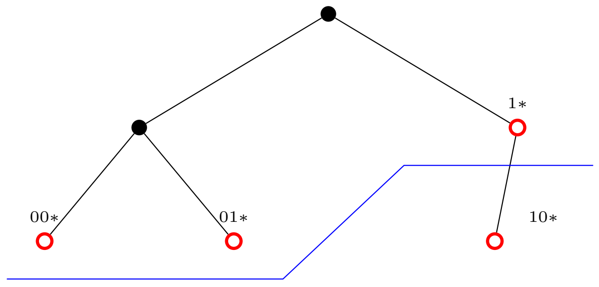


Figure 4.4: Single-forward rule tree.

The set of resulting prefixes obtained by iterating the merging operation in R and that are maximal for $>$ is called $MaxFwdMerges(R)$.

On a filtering path of length n, let R_i be a rule subset placed in switch $s_i, 0 < i < n$. The potentially necessary forward rules in $s_{i+1}, i+1 \leq n$ after adding rules in s_i is composed of successive $MaxFwdMerges$ results and called $PotentialFwds$. Considering R_{i+1}, only forward rules needed to resolve action conflicts with $R_{1,\ldots,i}$ are added in s_{i+1}. Thus, the forward ruleset needed in switch s_{i+1} is a subset of $PotentialFwds$. If a prefix covers only deny rules in R, packets matching this prefix in s_i will not travel to the next switch. Therefore, such a prefix is not added to $PotentialFwds$ of s_{i+1} since there will be no packet to match.

4.3.3 Distribution Algorithm

Let us consider the binary tree shown in Fig. 4.5 which represents an ordered ruleset. As an illustration example, each node contains one rule, and we need to distribute the ruleset along the path shown in Fig. 4.1. In this path, each switch has a maximal capacity of 6 rules. If we consider the default rule in each switch, we need to find a set of 5 rules to fill the first switch. As depicted in Fig. 4.5, we select the set of nodes containing a total of 5 rules maximum (all nodes in red). After this selection step, all rules added to the switch will be removed from the binary tree. If some space remains in the switch, we will try to find other rules candidates.

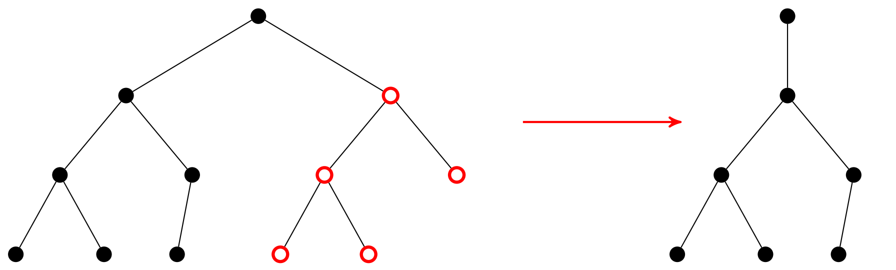


Figure 4.5: Choosing a set of candidate rules from the binary tree to be placed in a switch.

The rule distribution is performed in one pass following a bin-focused approach and a **Minimum Bin Slack** heuristic [126]. Thus, each switch is filled as much as possible with a set of rules while trying to minimize the overhead caused by action conflicts with rules in previous switches. The required extra forward rules in switch i for a ruleset R_i is defined as $NeededFwds(R_i, PotentialFwds) = \{fwd \in PotentialFwds \mid \exists r \in R_i$ such that $r >: fwd \wedge Action(r) = "Deny"\}$.

Algo. 5 is firstly called with an empty $PotentialFwds$ and it describes the distribution of a given ruleset over switches in a single path.

Algorithm 5 DecPath($R, \langle s_1, s_2, \ldots, s_k\rangle, PotentialFwds$)
Input: set of rules R, k switches s_i of capacities c_i, set of forward rules $PotentialFwds$
Output: $ChosenRules$, the rules added to the k switches;
R, the remaining rules;
$PotentialFwds$, the forward rules computed after adding $ChosenRules$.

```
1: Let $ChosenRules \leftarrow \varnothing$
2: for $i = 1$ to $k$ do
3:     Add a default forward rule to $s_i$
4:     Let $R_i \leftarrow ChosenCandidates(R, s_i, PotentialFwds)$
5:     $ChosenRules \leftarrow ChosenRules \cup R_i$
6:     Add $R_i \cup NeededFwds(R_i, PotentialFwds)$ to $s_i$, replacing the default forward rule if $R_i$ contains the default rule
7:     $PotentialFwds \leftarrow MaxFwdMerges(R_i \cup PotentialFwds)$
8:     $R \leftarrow R \setminus R_i$
9: end for
10: return $\langle R, PotentialFwds, ChosenRules\rangle$
```

The function $ChosenCandidates$ searches in the rules tree R for a rule subset R_i that can fit in switch s_i, and has a maximum of:

$$|R_i| - |NeededFwds(R_i, PotentialFwds)|$$

4.3.4 Algorithmic Complexity

When building the rules tree, we can efficiently add information that will speed up the set selection part. In that way, we record with every node *i)* the number of rules in its subtree, *ii)* whether this subtree is a blacklist, and *iii)* whether this node prefix is a valid result of prefix merges. When a rule r is inserted in a given node t_n, every parent of t_n can update these variables in $O(w)$ since w (word length) is an upper bound for the length of the bit-prefix of r. The rule tree construction part is then achieved in $O(nw)$ time.

Selecting the rules to be stored in a switch requires at most an entire tree traversal in $O(nw)$ time. Given a node prefix p, searching for a rule conflict with it can be done in $O(w)$ time, as the forward rules can also be represented in a tree data structure. The distribution can be done in $O(nw)$ time for every switch.

We adopt a greedy approach where the potential forward rules tree representation of $O(nw)$ size is the main runtime memory cost.

4.4 Evaluation

4.4.1 Simulation Setup

In our evaluation of Algo. 5, we rely on 2 rulesets available in [127]. These rulesets are generated with ClassBench [128]. The first one, called "fw1" contains 8902 forwarding rules, while the second one, called "acl1", contains 9928 access control rules. Our algorithms are implemented in Java with single-threaded programs. The evaluation has been performed on a desktop computer with Intel Core i7-7700 3.6-GHz CPU, 32 GB of RAM, and running the last version of Windows 10 operating system.

We define the overhead OH in terms of additional forward rules placed on the switches of a path for a given ruleset as follows:

$$OH = \frac{N_t - N_i}{N_i}$$

where

> N_t is the total number of rules used in a path (initial ruleset plus extra forward rules).
>
> N_i is the number of rules in the initial ruleset.

The overhead depends on the length of the path, the capacity of the switches on the path, and the diversity of rules' actions in the ruleset.

4.4.2 Simulation Results

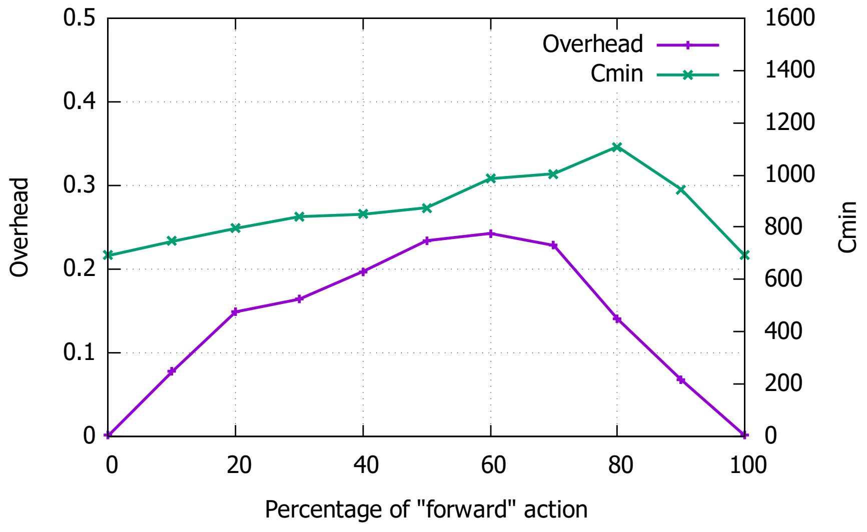
We decompose the rulesets using the source IP address as a filtering field. Even with a synthetic ruleset of 64000 random rules, no ruleset takes more than 150ms to be decomposed and distributed. This processing time includes tree building, ruleset decomposition, and rules distribution. This level of performance should enable a network administrator to react and deploy almost immediately an updated policy suited to a new kind of attack flow.

Applying our approach to two classical rulesets, white-lists and black-lists, the forward rules overhead is very low since such rulesets contain few actions conflicts. For example, only the last default rule has a "Deny" action with a white-list policy, so only the last switch will need additional rules in its table. In terms of a blacklist policy, no forward rules are required, so the tree structure is only useful as a sorting and updating mechanism. The decomposition of multi-field filtering policies is possible when dealing with blacklists, with only one default forward rule added to each switch except the last one.

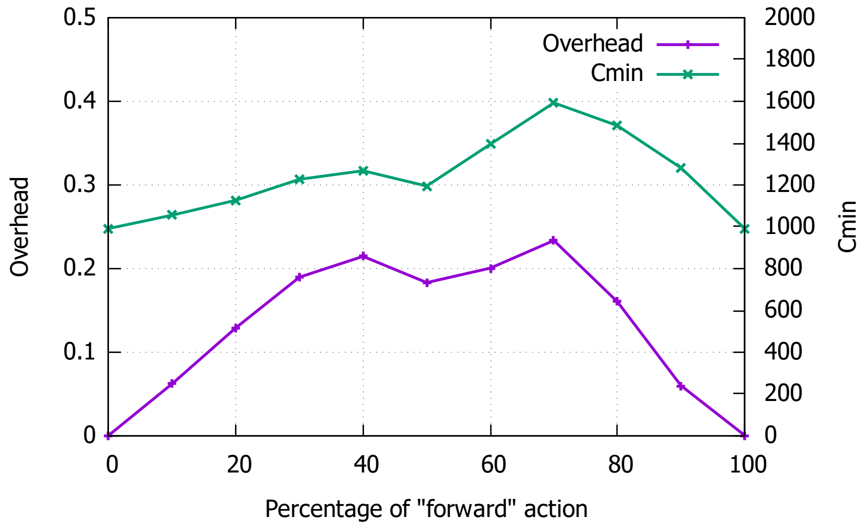
If a set of rules R1 (0*,deny) and R2 (*,forward) need to be divided using OBS, a forward rule for R1 will be generated to avoid conflict with R2. However, using our approach, no forward rule should be generated since a packet matching R1 in a switch does not continue to the next one. To show the effect of the action field on the overhead OH and the minimal capacity C_{min}, we use two rulesets (fw1 and acl1), from the 12 rulesets [90], with 0 to 100% percentage of rules having "Forward" action. The average values of OH and C_{min} are computed by running 10 simulations for each percentage value since the "Forward" actions will be distributed randomly among prefixes.

Fig. 4.6 shows the effect of the action field on both OH and C_{min} metrics. When a set of rules has very low action diversity, the overhead is very low since the number of conflicting actions is small. The situation where half of the rules have a "Forward" action introduces the highest overhead, and we obtain similar results with the other rulesets.

Fig. 4.7 shows the overhead distribution using the 12 rulesets from ClassBench while varying the path length. The overhead in the worst case is around 30% with 8 switches and for 50% of accepting rules. By using LPM, our approach does not necessarily introduce rule duplication with sets having overlapping rules like the OBS approach. For example, if a rule r is selected, no rule r' such as $r' > r$ is needed in the current switch. Only forward rules fw such as $r > fw$ are added to resolve action conflicts. However, the performance of our LPM based method can not be compared to OBS in terms of placement results because of the single filtering field that we use. In addition, the OBS approach operates on both source and destination fields, which is not the case for our method.



(a) fw1



(b) acl1

Figure 4.6: Effect of action field on overhead OH and C_{min} using respectively two different rulesets (fw1 and acl1).

Contrarily to Palette [89], we do not cut the packet header space in two, so our performance is not affected by the path length being or not a power of two, as highlighted in OBS benchmarks. We do not need either to build classifiers in such a way that rules from two different classifiers do not intersect. We only have to respect the rule precedence relation, which enables us to decompose a given ruleset without the overhead of rule bits

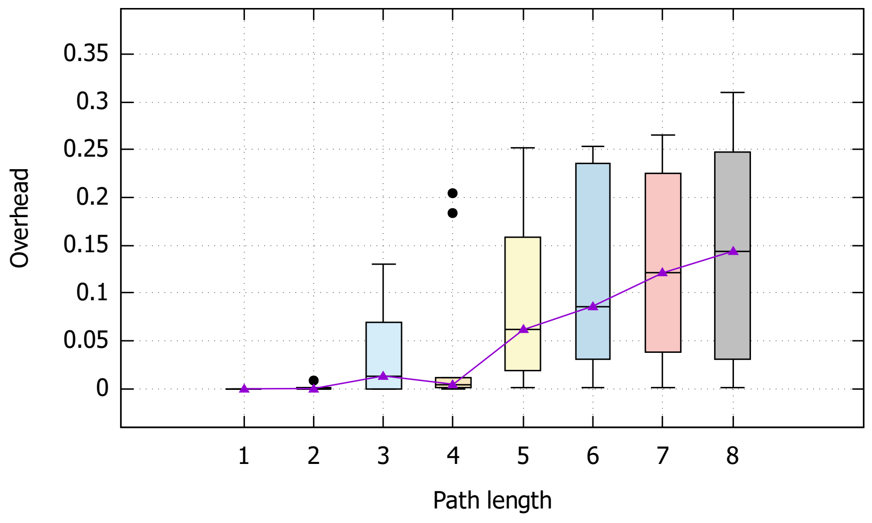


Figure 4.7: Rule space overhead while distributing a ruleset with a 50% of rules having Forward actions.

expansion. While Palette assumes that the classifiers have the same size at the end of the decomposition, our approach is not dependent on switch capacities, as each next switch receives as many rules as it can with our algorithm.

Our technique based on LPM is limited to rules composed of a bit prefix followed by wildcards. For general rules with wildcards possibly occurring at any position, we have to perform a rule expansion like in Palette Pivot Bit Decomposition [89].

4.5 Summary

In this chapter, we introduce a new LPM based distribution algorithm. Like OBS [86], and Palette [89], we rely on rule dependency relations, but we generate fewer forward rules by considering the action field. Indeed, a packet can be matched by rules with two prefixes from successive switches if their resulting action does not violate the initial policy semantics. As previous works we adress the problem of distributing a ruleset among network switches to meet a policy. However, in our approach, we rely on the properties of the LPM strategy and leverage action field information to handle overlapping rules and avoid replication in switch tables. Furthermore, unlike [90] our solution does not need packet or rule modifications. Our simulations show that when rules have the same action field, the overhead is close to null. This technique can be helpful if we use a blacklist or a whitelist where all rules have the same action. In the next chapter, we will introduce a general approach for all types of rules strategies. We will also introduce distribution techniques for rules over more complex network topologies.

Chapter 5

Rules Distribution Over a Graph

Contents

5.1 Introduction

The previous chapter introduces a distribution algorithm for one dimension rules that work with LPM strategy. The algorithm generated forward rules in each switch to preserve the semantics of the rulesets. In addition, the algorithm is developed for single path topologies. However, network topologies can be more complex with multiple overlapping or crossing paths. While a distribution on one path can be relatively easy to deploy, with multiple paths scenarios, rules in nodes belonging to multiple routes must be carefully managed to serve all of them and limit redundancies.

In this chapter, we present a technique to enforce the same filtering policy across all paths in the network while limiting rules redundancies. First, in Section 5.2 we present an algorithm for distributing a set of rules on a network whose topology is a two-terminal series-parallel graph [129]. Compared to the algorithms presented in Chapter 4, this algorithm is not specific to LPM policy. It can be applied to all rule matching strategies and to all dimensions. Then we show how to handle more general two-terminal directed acyclic graphs. Next, in Section 5.3 we introduce a more general approach to the decomposition problem that does not rely on any additional rules. Section 5.4 evaluates the performance of our distribution technique. In Section 5.5 we introduce an update strategy to handle modifications in rulesets and topologies, and we evaluate the performance of our technique. Finally, Section 5.6 summarizes and concludes this chapter.

5.2 Two-Terminal Series-Parallel Graph

Series-parallel graphs are widely used in telecommunication networks [130], since the failure of a network part can be mitigated by using a parallel path, especially in critical applications with low tolerance to network paths failure. Moreover, these network types can be updated to easily add or remove some parts by dividing the network into parallel or series compositions.

Definition 2. *A two-terminal directed acyclic graph (st-dag) has only one source s and only one sink t. A series-parallel (SP) graph is an st-dag defined recursively as follows:*
(i) A single edge (u, v) forms a series-parallel graph with source u and sink v.
(ii) If G1 and G2 are series-parallel graphs, so is the graph obtained by either of the following :
(a) Parallel composition: identify the source of G_1 with the source of G_2 and the sink of G_1 with the sink of G_2. (b) Series composition: identify the sink of G_1 with the source of G_2.

A two-terminal series-parallel graph is a graph with distinct source and destination vertices, and obtained by a set of series and parallel compositions, merging the source and destination of components in case of series decomposition or by merging the two sources and the two destinations in case of parallel decomposition as shown respectively in Fig. 5.1 and Fig. 5.2.

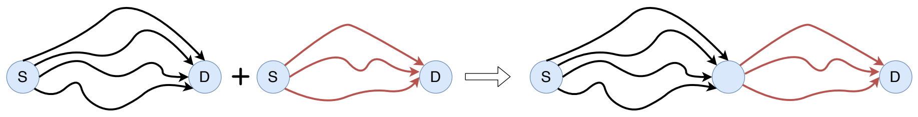


Figure 5.1: Series composition.

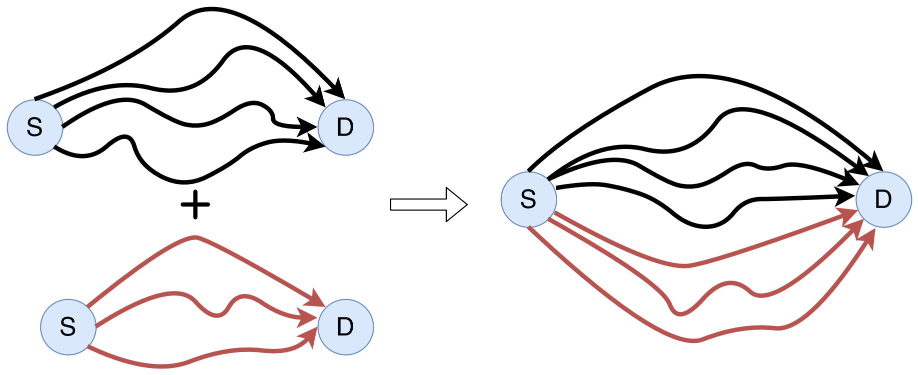


Figure 5.2: Parallel composition.

We define an S-component as an oriented path, that is, either an edge or a series composition of edges as shown in Fig. 5.3. A series-parallel graph can be represented by a binary tree [131], where each internal node of the tree is a series or a parallel composition operation and each leaf is an edge of the graph. Fig. 5.4 shows a binary tree representation of a series-parallel graph.

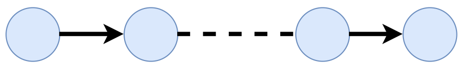


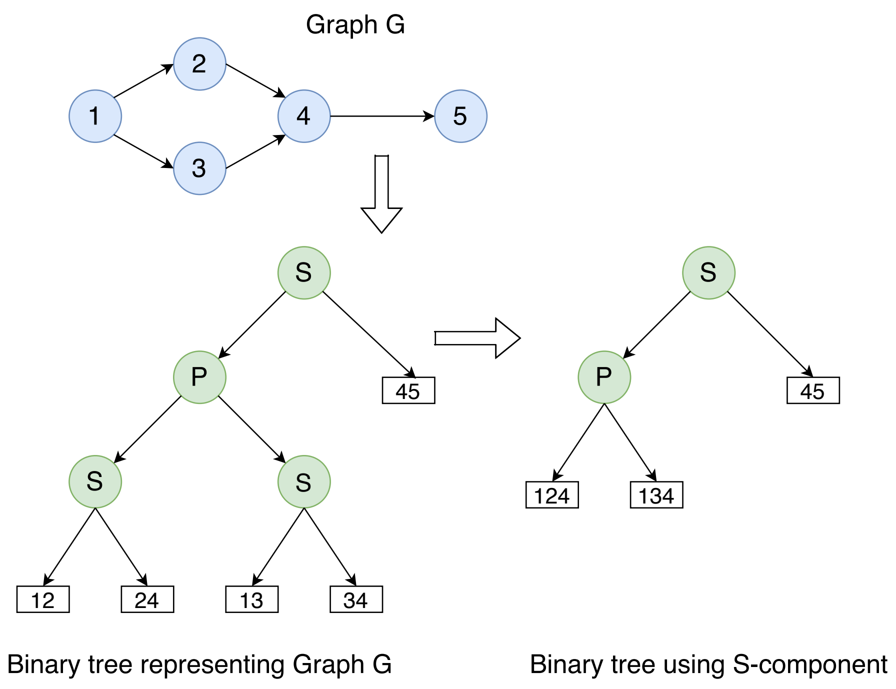
Figure 5.3: S-component.

We can derive a more compact representation of series-parallel graphs by replacing maximal subtrees built solely from series operators by the S-components they represent. Hence, leaves are S-components in this alternative representation.

In the example of Fig. 5.4, edges $1 \to 2$ and $2 \to 4$ can be merged into one S-component, as edges $1 \to 3$ and $3 \to 4$.

5.2.1 Distribution Algorithm

To determine if our algorithm can find a solution given a ruleset R to decompose, we need to try the decomposition on all paths of the directed network graph. We will then find the smallest ruleset that can fit in all paths, considering the overhead in terms of forward rules and the compatibility between different paths that share some switches.



Graph G
1
2
3
4
5
S
P
S
S
45
12
24
13
34
S
P
45
124
134
Binary tree representing Graph G
Binary tree using S-component

Figure 5.4: Tree representation of a series-parallel graph.

Algo. 6 outputs a ruleset placed in a specific path from s to t using a representation of the graph as a binary tree with S-components. Let R_0 be the initially given policy ruleset, and N_0 the root of a binary tree T_0 representing the series-parallel network graph. Each node N of the binary tree will be labelled with a triplet $\langle R^{'}, F^{'}, R\rangle$, where:

$R^{'}$ is the remaining rules to place in next switches.

$F^{'}$ represents the $PotentialFwds$ set computed from $R_0 \setminus R^{'}$.

R is the subset of rules from R_0 occurring in the subtree rooted at N.

The obtained labeling can be interpreted as a solution to the distribution problem. We define the solution affected to a node n in T_0 to be the label of the subtree rooted at n.

If node N corresponds to an S-component, the field $N.switches$ in Algo. 6 is by definition the sequence of switches in the S-component minus the first one if that S-component is in series composition with a previous one.

Algo. 6 has as parameters a node of T_0, a set of rules $R \subseteq R_0$, a set of forward rules F. In the main program the recursive procedure Algo. 6 is called with N equal to the root of T_0, F empty, R equal to R_0. This algorithm terminates successfully when at the end the first field R' of

Algorithm 6 DecGraph(N, R, F)
Input: N, node in T_0; R, set of rules;
F, set of forward rules to be installed in the following switches.
Output: label of N

```
1: if N.isS-Component then
2:     $\langle R^{'}, F^{'}, R\rangle \leftarrow$ DecPath($R, N.switches, F$)          ▷refer to Algo. 5
3:     return $\langle R^{'}, F^{'}, R\rangle$
4: else if N.isSeries then
5:     $\langle R^{'}, F^{'}, leftR\rangle \leftarrow$ DecGraph($N.leftChild, R, F$)
6:     $\langle R^{''}, F^{''}, rightR\rangle \leftarrow$ DecGraph($N.rightChild, R^{'}, F^{'}$)
7:     return $\langle R^{''}, F^{''}, leftR \cup rightR\rangle$
8: else if N.isParallel then
9:     $\langle R^{'}, F^{'}, leftR\rangle \leftarrow$ DecGraph($N.leftChild, R, F$)
10:    $\langle R^{''}, F^{''}, rightR\rangle \leftarrow$ DecGraph($N.rightChild, R, F$)
11:    while $leftR \neq rightR$ do
12:        if $|leftR| < |rightR|$ then
13:            $\langle R^{''}, F^{''}, rightR\rangle \leftarrow$ DecGraph($N.rightChild, leftR, F$)
14:        else
15:            $\langle R^{'}, F^{'}, leftR\rangle \leftarrow$ DecGraph($N.leftChild, rightR, F$)
16:        end if
17:    end while
18:    return $\langle R^{'}, F^{'}, leftR\rangle$          ▷equal to $\langle R^{''}, F^{''}, rightR\rangle$
19: end if
```

the root label is empty: this ensures that all rules have been distributed successfully and no rule has been left out.

This algorithm can be applied with other decomposition algorithms like those defined in [86; 89; 90] just by modifying the *DecPath* function that is called at Line 2. Note that, in our approach and unlike One Big Switch (OBS) [86] technique, we do not create several rule table partitions for each path traversing an intersection switch. Instead, a packet will be processed by the same rule table at an intersection, regardless of the path it came from. This also facilitates policy updating, as the places where some specific rules occur are easier to localize.

Proof of correctness for Algo. 6

By induction, we conclude that if we can build a solution for a graph obtained by eliminating Braess graphs successively, we can construct a solution for the initial st-dag. After applying Algo. 6 on the syntax tree of a series-parallel graph, the relations between the labels in each node are partially determined by which rule sets are placed in subtrees with a parallel node root.

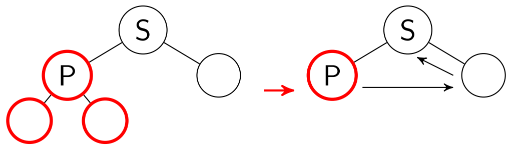


Figure 5.5: Dependency path between subtrees' labels.

In Fig. 5.5, the children of the parallel node contain the same subset of rules from R_0, so they share a common label with their parent, which is denoted by their red color. The simplified tree at the right is the minimal one necessary to establish the final labels. As stated, a subgraph G_2 labelling depends on the label affected to the subgraph G_1 whose switches lead to G_1's source. These labellings must summarize a correct LPM rule distribution. Assuming that *DecPath* procedure in Alg. 6 implements the LPM semantics of the initial rule set R_0 in the path directly given, we show it is also true for the series composition of path components on which *DecPath* is called:

Lemma 1. *Switches in a given path enforce a correct LPM policy.*

Proof: The first S-component encountered in a prefix traversal is constituted of the first switches in the path. Applying *DecPath* on this component will ensure that LPM is respected among its switches. The recursive calls of Algo. 6 follow the prefix order, so the precedence relation between switches is maintained. Lines 4-7 of Algo. 6 maintain the coherence between calls and resulting labellings under a series node. If N with label $\langle R', F', R\rangle$ has children, then for the left and right ones T_1 and T_2, if $T_1.L = \langle R'_1, F'_1, R_1\rangle$ then $T_2.L = \langle R'_2, F'_2, R_2\rangle$ with:

– $R' = R'_2 = R'_1 \setminus R_2$

– $F' = F'_2 = MaxFwdMerge(R_0 \setminus R'_2)$

– $R_2 \in R'_1$

– $R = R_1 \cup R_2$

With this precedence and forward relations, a series node's right child will receive the appropriate forward rules and rules from R_0 as base for distribution in its subtree. *DecPath* is called several times for different components of a path. The definition of $N.switches$ guarantees that in a series composition, the common vertex between two components is not filled twice. With a path $s_1 \rightarrow s_2 \rightarrow s_3$, *DecPath* could then be called on $s1 \rightarrow s_2$ then s_3, so s_2 would not be filled again. Consequently, the result of successive *DecPath* calls on path components is equal to the one obtained with a single call on the whole path. This path contains a rule set decomposed with respect to the intended LPM policy. □

When a packet has the option to take different paths towards a network location, each of these paths must be equivalent, filtering-wise. Algo. 6 must terminate correctly in this case.

Lemma 2. *Switches in parallel paths enforce the same filtering policy.*

Proof: Lines 8-18 ensure that a parallel node will share the same labelling as its two children, i.e its children subtrees will contain the same rules from R_0. Algo. 6 always terminates, because of the process of decomposition trial and error. After decomposing a part of R_0 in the two parallel subgraphs represented by T_1 and T_2, we obtain R_1 and R_2 respectively fitting in those subgraphs. Three cases can occur:

- R_1 is identical to R_2.
- R_1 or R_2 has smaller size than the other.
- R_1 and R_2 are different rule sets of the same size.

These last two cases indicate a decomposition mismatch. To correct it, Algo. 6 iteratively tries to propagate the smallest rule set resulting from T_1 or T_2. If R_1 has to be propagated on T_2, $R_1' \subseteq R1$ will fit in T_2. Because of this inclusion relation, R_1' can fit in T_1 and Algo. 6 returns it once propagated. The process is symmetrical if R_2 has to be propagated on T_1. Thus, Algo. 6 terminates after having ensured that two parallel subgraphs enforce the same filtering policy. Only one set of forward rules can be used at the common sink of these subgraphs. $\square$

5.2.2 Algorithmic Complexity

The labels computed by Algo. 6 when it is successful, defines a solution to the rule distribution problem. The labels are obtained after exploring the entire graph, and the tree representation allows for single processing of edges shared by several paths. Several trial and error steps can be necessary for rule propagation, but the process is substantially faster in many cases since only one propagation try has to be performed under a parallel node. Such cases include paths with switches of identical capacities or rulesets in which action conflicts cannot happen, for example, when the rules sets are blacklists. In this way, given a graph $G = (V, E)$, Algo. 6 computes a ruleset decomposition and distribution in $O(p|E|)$ time where p is the number of parallel nodes. Since the cost for distributing rules on a switch is $O(nw)$, we have an overall complexity of $O(p|E|nw)$.

5.2.3 Generalization to St-Dags

In Subsection 5.2, we discussed how to distribute rulesets in series-parallel graph. Here, we show that this technique can be applied to arbitrary two-terminal directed acyclic graphs or st-dags. Duffin [129] proved that a two-terminal st-dag is series-parallel if and only if it does not contain any subgraph homeomorphic to the Braess graph as shown in Fig. 5.6.

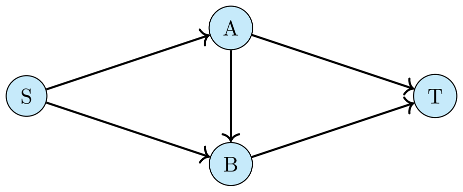


Figure 5.6: The Braess graph.

We note in such a graph that having the same rulesets in vertices A, and B still leads to a correct distribution: a packet traveling from A to B would be applied the same rule and action a second time. To generalize the distribution mechanism, A and B could be considered as one vertex, thus eliminating a Braess graph to obtain a series-parallel one. We then merge them as illustrated in Fig. 5.7.

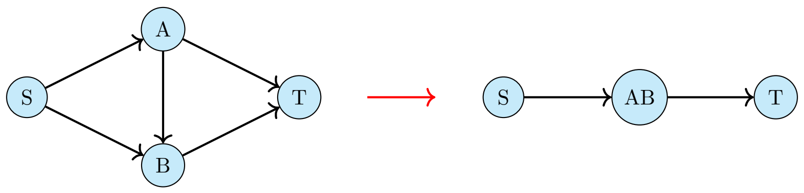


Figure 5.7: Merging of vertices A and B to eliminate Braess components.

Let $G = (V, E)$ the initial network graph, which is not series-parallel. Let a Braess component of G be a subgraph whose vertices are labeled as in Fig. 5.6. Let us define the set $Inter$ to be the set of vertices occurring on any path from A to B including A and B. We perform a merge on A and B to obtain the graph $G' = (V', E')$ defined as follows:

- V': $(V \setminus Inter) \cup \{AB\}$ with $capacity(AB) = min(capacity(A), capacity(B))$
- E':
  - if $x \to y \in E \wedge x = A \wedge y \notin Inter$ then $AB \to y \in E'$
  - if $x \to y \in E \wedge x = B$ then $AB \to y \in E'$
  - if $x \to y \in E \wedge y \in \{A, B\}$ then $x \to AB \in E'$
  - if $x \to y \in E \wedge \{x, y\} \cap Inter = \emptyset$ then $x \to y \in E'$

Iterating the merging operation and once every Braess component has been eliminated, we apply our distribution algorithms on the resulting series-parallel graph. The decomposition and distribution are valid as shown by:

Lemma 3. *If we have a solution for the distribution problem of the initial ruleset R_0 in G', then we can construct a solution for the graph G.*

Proof: Leaving empty the intermediate vertices between A and B, we know exactly where the rules will be located. As AB contains a ruleset that can fit in both A and B by definition, this ruleset is duplicated in A and B. The solution in G' is supposed valid, so precedence and intersection constraints at AB are satisfied. Thus, paths $S \to \cdots \to A$ and $S \to \cdots \to B$ contain the same rules from R_0 and the potentially necessary forward ruleset Fw is the same just after A and B. For the same reason, it is also true that paths $A \to \cdots \to T$ and $B \to \cdots \to T$ contain the same rules from R_0 and generate the same forward rules. Distribution constraints are then satisfied for paths $S \to \cdots \to T$ and the solution affected to G is valid. $\square$

Series-Parallel Network Derivation by Braess Subgraph Elimination

We show with an example how merging nodes in a network can transform any st-dag graph into a series-parallel graph. We consider a real network topology from [132] with 19 nodes, each representing a town, as shown in Fig. 5.8. Our objective is to distribute rules over this network so that the processing of any packet traversing any path from Frankfurt to Budapest will end with the same result. We added a capacity for each switch as shown in Fig. 5.8. Before running our distribution and decomposition algorithm, we need to eliminate any Braess subgraph to transform the graph into a series-parallel one as described in before. This can be done by merging some nodes. First, Bratislava and Vienna are replaced by one node BV. Then Salzburg and Villach are merged to generate SV.

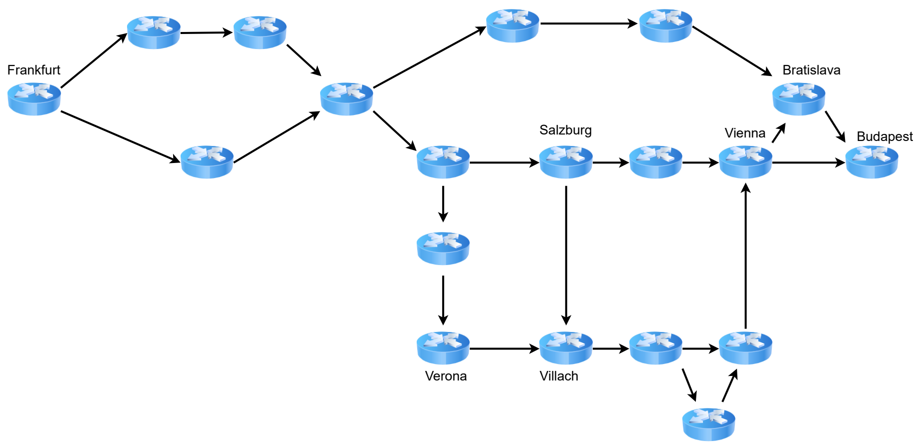


Figure 5.8: Memorex network.

Fig. 5.9 shows the final result of our transformation. Algo 6 can be applied to the binary tree of the derived series-parallel graph. After the decomposition is performed, nodes Salzburg and Villach get the same set of rules. Similarly, Bratislava and Vienna are assigned the same set of rules. In Fig. 5.10 we show an example of applying the distribution algo-

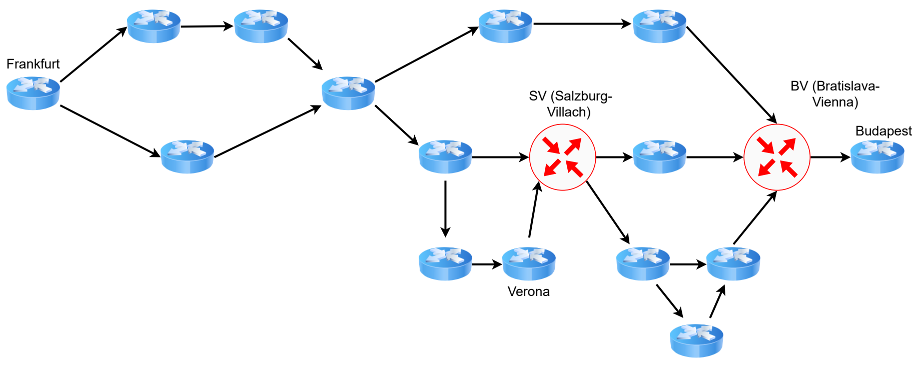


Figure 5.9: Series-parallel approximation of Memorex network.

rithm Algo. 6 to the series-parallel approximation of Memorex network. In this example, a prioritized list is given as input, and the decomposition algorithm does not need to generate forward rules. Each node i represents a switch with capacity $Capa_i$. Node 11 is a combination of two nodes from the initial network. The capacity of this node is taken to be the minimal capacity of the two nodes that have been merged since the solution for the initial network will be obtained by lifting the merged node solution to these nodes. The same is true for node 16. $Capa_i = 2$ for all $i \in \{1, 4, 5, 6, 7, 8, 9, 10, 16, 17\}$ and 1 otherwise. Fig5.10 shows the

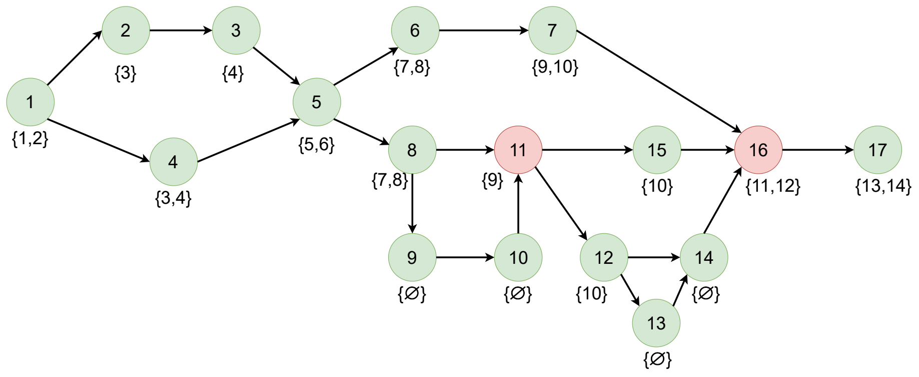


Figure 5.10: Distribution algorithm on the series-parallel approximation of Memorex network

Distribution of rules with Algo. 6 on the series-parallel approximation of Memorex network with a simple decomposition algorithm without forward rules. The first component has two parallel paths $1-2-3-4$ and $1-4-5$. The algorithm starts by computing a solution for each path (Lines 9-10). Since the two solutions are equal, one will be returned. Then the algorithm processes the next component with the set of rules not stored in the first component (Lines 5-6). Some switches like 9 and 10 have an empty set. The algorithm computes a solution for Path 8-9 with 3 rules, then for the parallel path $8-9-10-11$, which has 7 rules. According to Algo. 6 (Lines 11-18), the solution with the smaller set of rules should be projected to the other parallel paths of the component. In this case, Switches 9 and 10 remain empty.

5.3 Two-Tier Distribution Approach

In our first approach described in Section 4.3, we restrict the rules to be single field filtering rules, mainly to reduce the forward rules overhead for conflicting actions. Although this technique is helpful in blacklisting policies, it is limited and inefficient when decomposing and distributing complex policies that involve filtering rules with IPs, ports fields, and overlapping.

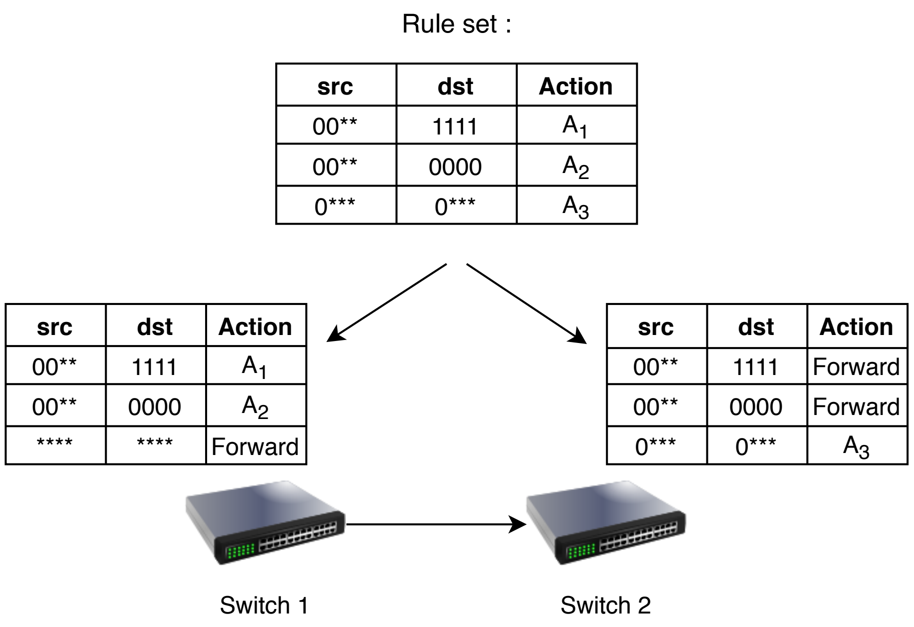
5.3.1 Distribution of Multi-Fields Rulesets

If we consider, for example, both source and destination IP fields, we can get a significant overhead in terms of forwarding rules when using our previous decomposition technique. As shown in Fig. 5.11, since for each rule added in Switch 1, we need to generate a forward rule for it in Switch 2. Unless we find mergeable prefixes on the second filtering field, we have to generate as many rules as we have removed from the initial ruleset, which limits the decomposition interest.

It is nevertheless feasible to apply this approach to several fields when the overlapping rules reside in the same switches or when we are dealing with a blacklist. With all actions of rules being "Deny", all packets matching some rules are stopped, and the other are captured by the default rule to be tested in the next switch, so non-default forward rules are unnecessary in a blacklist case. In [90], the authors propose a technique to deactivate rule matching on the following switches when a packet is matched in the current switch. However, their technique requires a non-standard modification on the packet structure by adding an additional bit.

5.3.2 Multi-level Distribution

We believe a more practical solution could be used in real-world environments and avoid modifications to the packet structure. To achieve this, we


Rule set :

src	dst	Action
00**	1111	A_1
00**	0000	A_2
0***	0***	A_3

src	dst	Action
00**	1111	A_1
00**	0000	A_2
****	****	Forward

src	dst	Action
00**	1111	Forward
00**	0000	Forward
0***	0***	A_3

Figure 5.11: A counter-productive example of forward rules generation applying Section 4.3 approach to multi-field filtering rules.

propose a novel approach with nearly zero rule space overhead in switch tables.

When distributing a ruleset in a network, a packet coming from the source switch needs to travel along a path towards the destination. Only a single filtering action needs to be applied to a given packet. Thus, if a packet is accepted in a switch, all the following switches must transfer the packet to the destination since a match with a higher priority rule has already been applied. In our first approach, the forward rules mechanism allows us to do that. However, it increases the total number of rules in the switches. To avoid this overhead problem and allow the distribution of multi-fields ruleset, we use two levels of switches: Forward Switches (FoS) and Filtering Switches (FiS) as shown in Fig. 5.12.

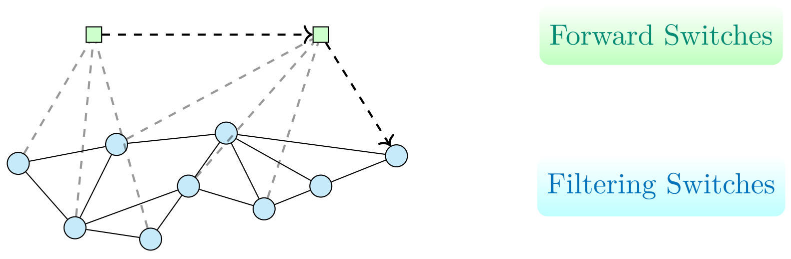


Figure 5.12: Network topology with multi-levels switches using FoS and FiS.

All FiS at a distance of 2 or more from the destination are connected to one FoS, as it would be impractical to connect all switches to the destination in practice. The role of an FoS is to transmit a packet directly to the destination to skip the next FiS on the path, reducing the path length and network latency. For the sake of simplicity, we use three types of actions in a rule: "Forward", "Deny" and "Default forward". If the selected action is "Forward", the packet is sent to an FoS, and if the action is "Deny", the packet is dropped. The default forward action sends the packet to the following filtering switch.

By using two types of switches, in this second approach, we eliminate the need for non-default forward rules, which reduces the total number of rules used over a path, and accelerates the processing time of each packet by forwarding it to the destination directly using FoS. A practical requirement for this technique is to rely on high throughput FoS. This technique is applicable in any priority strategy, assuming that every switch which enforces the network policy is linked to the destination, directly or through FoS.
A suitable networking architecture for our approach could be the Flat-tree structure [133], used in data centers. In a flat-tree architecture, we can perform matching tests over successive switches using neighbor-to-neighbor connections and benefit from a multi-tier topology. Indeed, once a packet matches a rule, it can be forwarded directly to the top-level switch instead of going through the other filtering switches. To distribute the rules on a network graph, the approach described in Section 5.2 can be applied without forward rules. Therefore the decomposition algorithm can be simplified since the F' component of node label of the binary tree in Algo. 6 can be eliminated.

5.4 Evaluation of Two-Tier Approach

5.4.1 Experimental Setup

In our evaluation of the two-tier approach presented before, we first consider a single path topology and then a tree-based topology with one source and different destinations. In those tests, we use 12 rulesets generated from ClassBench [128] and available in [127]. In this evaluation, all switches have the same capacity (tables size), and the minimal capacity required to find a rules placement solution by our distribution algorithm is defined as C_{min}. Mininet is used to build the topology, while all switches are connected to a RYU controller. The controller sends rules to all switches before the experiments and does not affect the results shown below. In each scenario, the bandwidth and the latency are computed by Qperf. Each experiment is launched 10 times on a desktop machine with Intel Core i7-9700 3.00-GHz CPU, 32 GB of RAM, and running Ubuntu 18.04

LTS.

Comparisons are made with three topologies, as shown in Fig. 5.13. Scenario 1 has only one FoS switch shared by all paths between source and destination nodes. In Scenario 2, each path will have its own FoS switch. Finally, in Scenario 3, there are no FoS at all. The latter Scenario aims to simulate other approaches like OBS [86] where a packet needs to traverse all switches to reach the destination node.

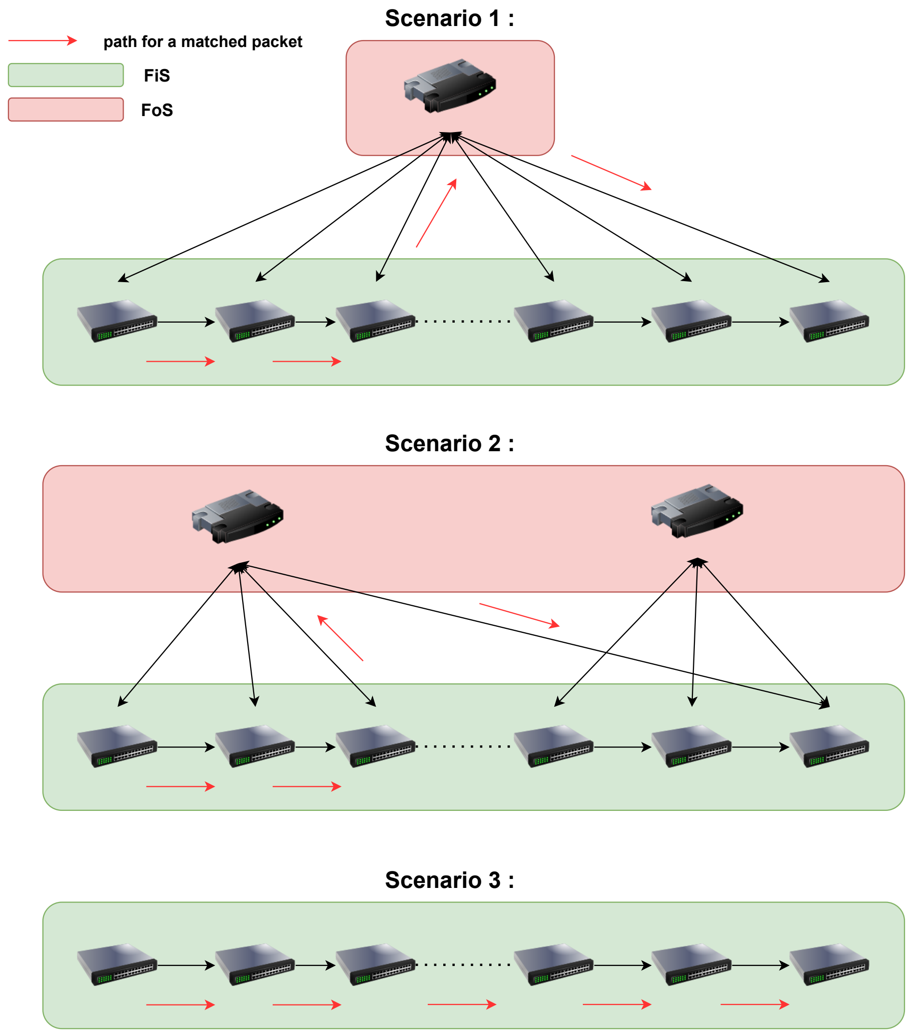


Figure 5.13: The three scenarios used in the experiments.

5.4.2 Overhead

Our two-tier approach for multi-field rulesets does not introduce rule space overhead except for the extra default rule in each switch and provides results similar to the OneBit [90] approach. However, our approach offers low rule space overhead. It preserves packets structure without introducing extra bits and ensures in a stateless way that packets can perform a network traversal without being matched.

Table 5.1: Values of C_{min} for different approaches on 12 rulesets with a single path of 10 switches.

		Palette	OBS	ONEBIT	TWO-TIER
rulesets	Size	C_{min}	C_{min}	C_{min}	C_{min}
acl1	9928	2480	1030	997	994
acl2	7433	2308	1214	746	745
acl3	9149	3622	2687	919	916
acl4	8059	2870	1706	809	807
acl5	9072	2265	912	910	909
fw1	8902	3776	2423	891	891
fw2	9968	4308	3688	997	998
fw3	8029	3877	2818	804	804
fw4	2633	1196	800	268	265
fw5	8136	2651	1780	814	815
ipc1	8338	2260	1088	837	835
ipc2	10000	4348	2406	1002	1001

Table 5.1 shows the different values of C_{min} obtained from [90], in addition to our two-tier approach results on the same rulesets with a single path of 10 switches topology. Our approach, similar to OneBit divides all rules over the path of 10 switches with an equivalent number of rules in each switch. In this case, each switch will be full. The only extra cost will be one default rule by switch. On the contrary, using Palette or OBS, for example, and a set of 9968 rules such as fw2, the value of C_{min} is around 4000. In this case, the number of used switches belonging to the path will be low, but these approaches introduce the cost of using switches with higher capacity and higher overhead because of rules duplication, which affects the performance of the matching rule process along the network.
Our two-tier approach can be used with arbitrary rules priority criterion and handle rulesets with multiple filtering fields. In addition, the forwarding switches (FoS) will decrease the time needed for a packet to reach its destination since an FoS does not perform any rule matching test. Additionally, since FoS is a straightforward forwarding device, the cost of adding a few FoS to the network remains reasonable.

5.4.3 Bandwidth and Latency

We consider a single path topology to study the effect of path length on bandwidth and latency. The size of the topology varies from 100 to 1000 switches.

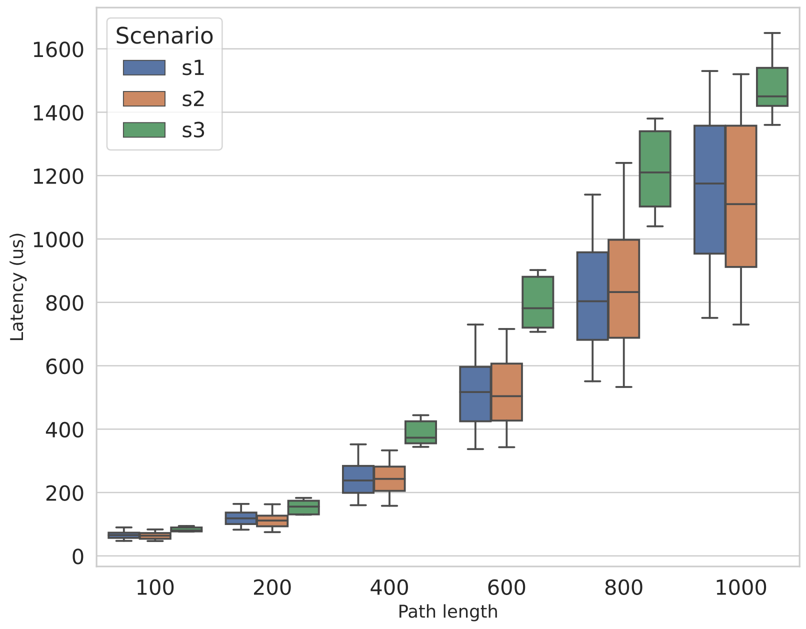


Figure 5.14: The latency on different single path topologies for each scenario.

Fig. 5.14- shows the latency computed for each topology on each scenario. The time needed for a packet to traverse a network increases with the topology length. Scenario 3 always has a higher latency than Scenarios 1 and 2 since a packet needs to traverse the entire network to reach the destination. The difference in latency between the three scenarios becomes noticeable, primarily starting from the 400 single switches topology. This is why we compute the packet loss on each one of the six topologies. The result in Fig. 5.16 shows that the proportion of packet loss in a topology with 200 switches or less is at most 6% in Scenario 3 and virtually non-existent in Scenarios 1 and 2. Beyond 200 switches, the differences in packet loss percentage start to appear, which explains why the difference in latency for the three scenarios is higher beyond a path length of more than 200 switches. Fig. 5.15 shows the bandwidth for each scenario on each topology. When the path length increases, the bandwidth decreases due to the added packet loss.

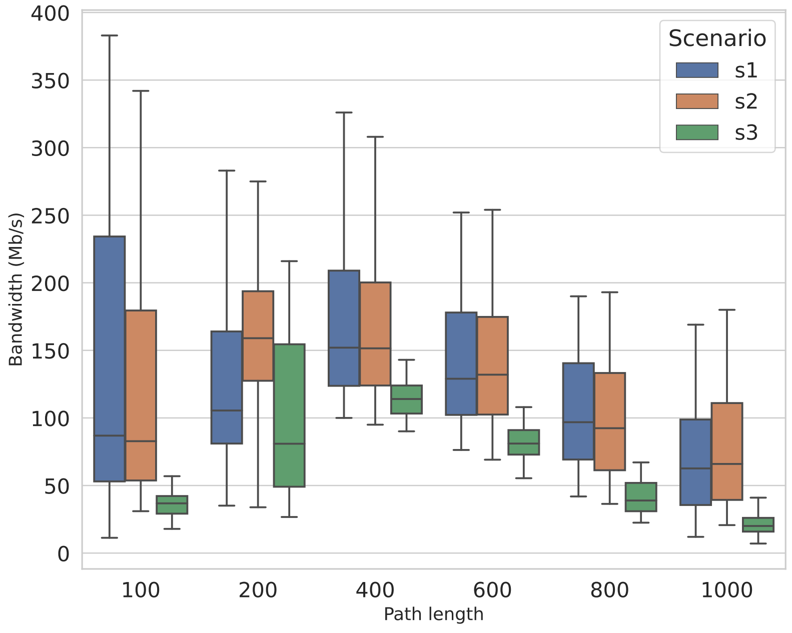


Figure 5.15: The bandwidth on different single path topologies for each scenario.

5.4.4 Multiple Destinations

In this subsection, we run several experiments on two topologies with one source and multiple destinations. The total number of switches in the first topology is 775, while the total is 774 switches in the second one. The size of the message sent from the source to the destination varies between 1 KB up to 1024 KB.

Fig. 5.17-(a) shows that the latency increases with the message size. This was expected since a larger message needs more time to traverse a path. In scenario 3, the path taken by a message to traverse is longer. Thus the latency is higher than scenarios 1 and 2.

In Fig. 5.17-(b), we observe that the bandwidth in scenarios 1 and 2 is always higher than scenario 3 regardless of the message size. This is because a smaller latency means that more individual packets can be sent from one point to another in a fixed time interval. With scenario 3, the highest latency and lowest bandwidth are obtained.

With 200 switches in a single path, scenarios 1 and 2 have around 10 to 15% latency and bandwidth improvements compared to scenario 3 due to fewer packet loss as shown in Fig. 5.16. Our second simulation is performed on a topology with longer paths than the first one. Each one of

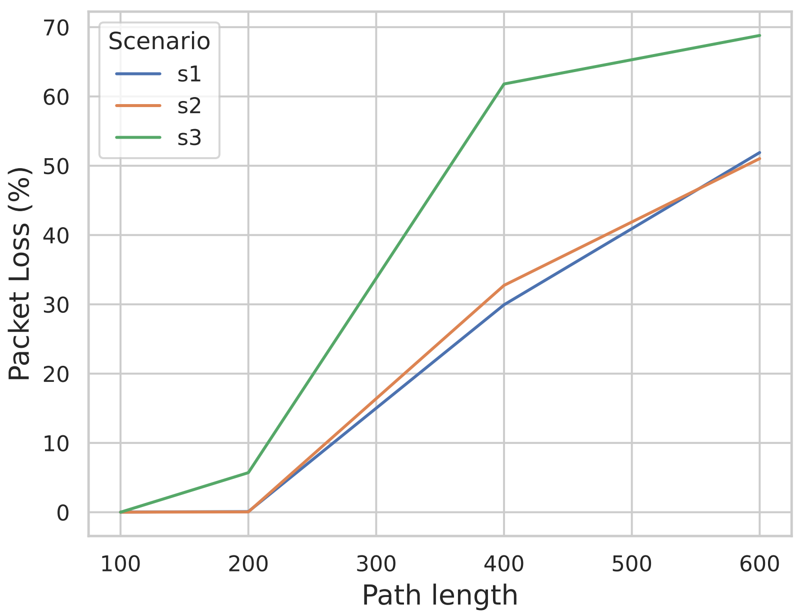
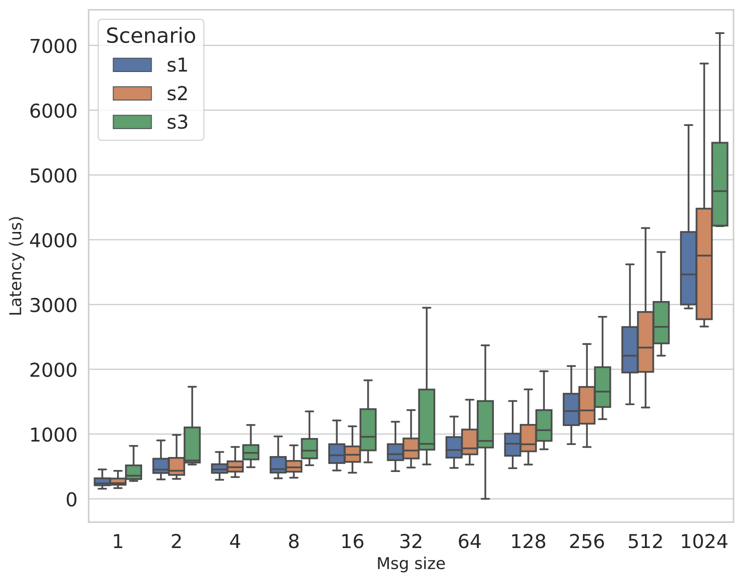


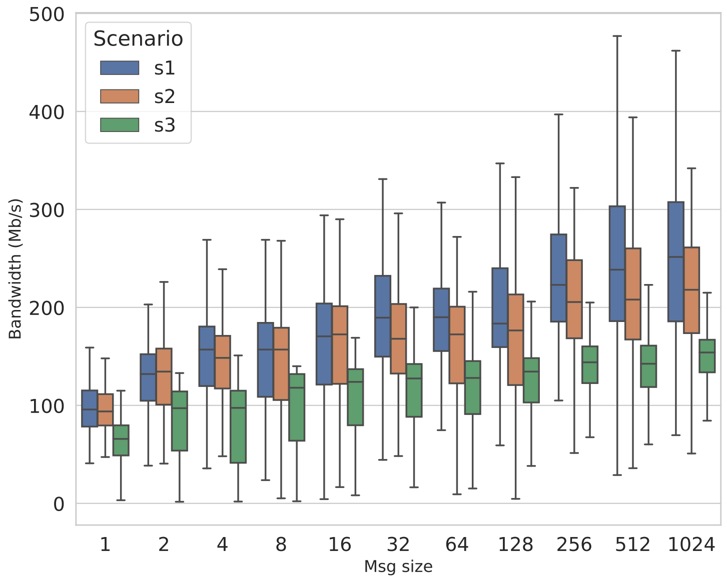
Figure 5.16: Percentage of packet loss on different scenarios.

the five paths contains 400 switches. We compute the bandwidth and the latency for the same configuration as in previous tests. Fig. 5.18-(a) shows the latency on the updated topology. The difference between scenarios 1,2, and scenario 3 is more than 30% in favor of the former on all different message sizes. Similarly, in Fig. 5.18-(b), the bandwidth of scenarios 1 and 2 is much higher than scenario 3 in the second topology. The difference in bandwidth also exceeds 30% here, our approach benefiting from the widening gap in packet loss on a path length of 400 switches. Regarding scenarios 1 and 2, the bandwidth loss worsens in similar proportions with the path's size as shown in Fig. 5.17-(b) and Fig. 5.18-(b). This is to be expected, as these scenarios introduce almost identical packet loss as in Fig. 5.16.

Our experiments show that the two-tier strategy minimizes the latency and increases the bandwidth compared to approaches where all switches have to be traversed between source and destination. This approach can be fruitful in other scenarios, where we can introduce several FoS, each one being dedicated to a specific application (VOIP, streaming, etc.). This strategy can also be used to prioritize a flow based on its importance or to prioritize some security applications by using FoS with an adequate security level for each type of flow.

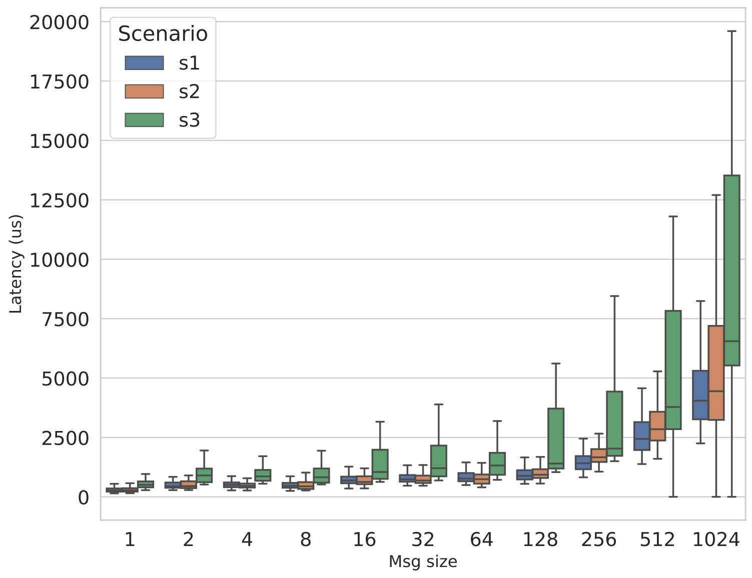


(a)

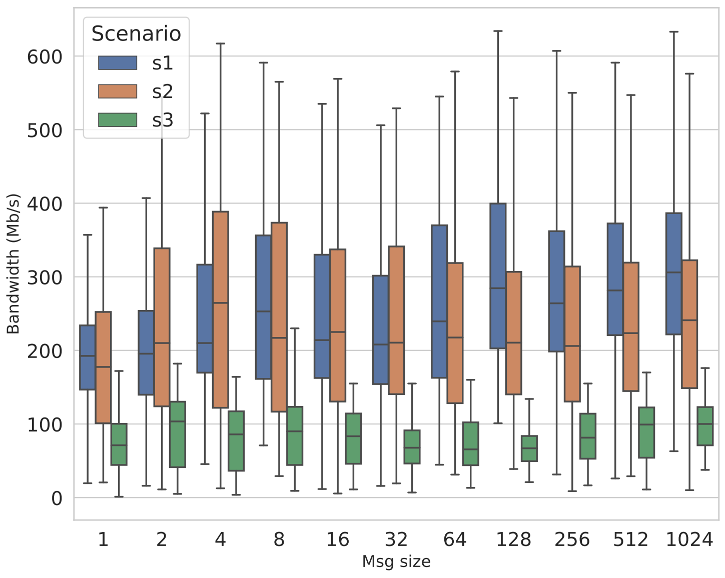


(b)

Figure 5.17: The latency and bandwidth using a topology with path lengths equal to 200 switches.



(a)



(b)

Figure 5.18: The latency and bandwidth using a topology with path lengths equal to 400 switches.

5.5 Rulesets Update Strategy

This section will introduce our update strategy to handle changes in the initial ruleset, i.e., adding, removing, or modifying a rule. A modification can be done to maintain the topology up to date, add more functionality, improve performance, add more links and paths to the network, or enable/block a specific flow. Updating a network can be done by adding or removing hardware equipment like switches and routers or updating the data like ACLs or forwarding rules inside existing equipment.

Security policies are constantly evolving and modifying the behavior to adopt towards some destination or origin addresses. An update needs to be deployed quickly before any security breach appears. Blocking suspicious traffic or routing the traffic to a firewall is a way to avoid a network policy violation. Implementing forward rules can be challenging, as it creates dependencies between switches. Any error in an R rule action can be reflected on all forward rules linked to R and can block most traffic, making the following switches useless.

The example in Fig. 5.19 shows how blocking a malicious traffic from IP address 192.168.100.1 will affect rules in next switches where forward rules had been installed before. Those rules need to be deleted since they are redundant. Deleting rules from switches minimize the consumed memory, giving space for future rules addition.

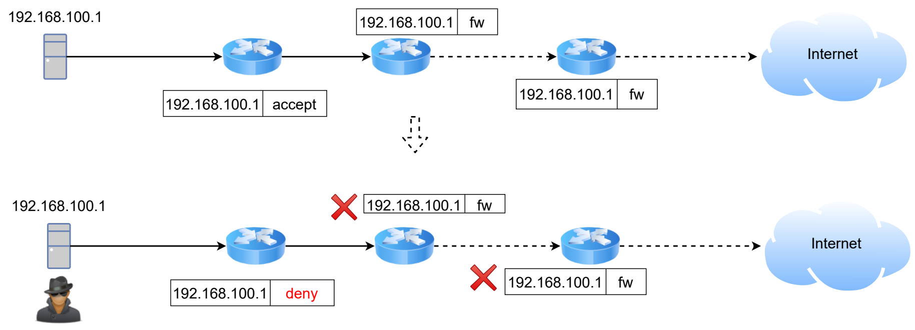


Figure 5.19: Blocking a malicious traffic from already allowed source.

Fig. 5.20 shows us how not updating a network after denying a host can lead to a security breach where an attacker can use old rules stored in switches.

The two examples show that a modification may affect all the network and must be handled carefully. In our work, we will focus on updating rules in series-parallel networks. In this type of network, a rule modification will affect all parallel paths previously enforced.

In a switch, rules are stored in a table and matched according to their priorities (e.g., given by their position in the table). In this case, adding, removing, or modifying a rule can affect all rules below the affected one and

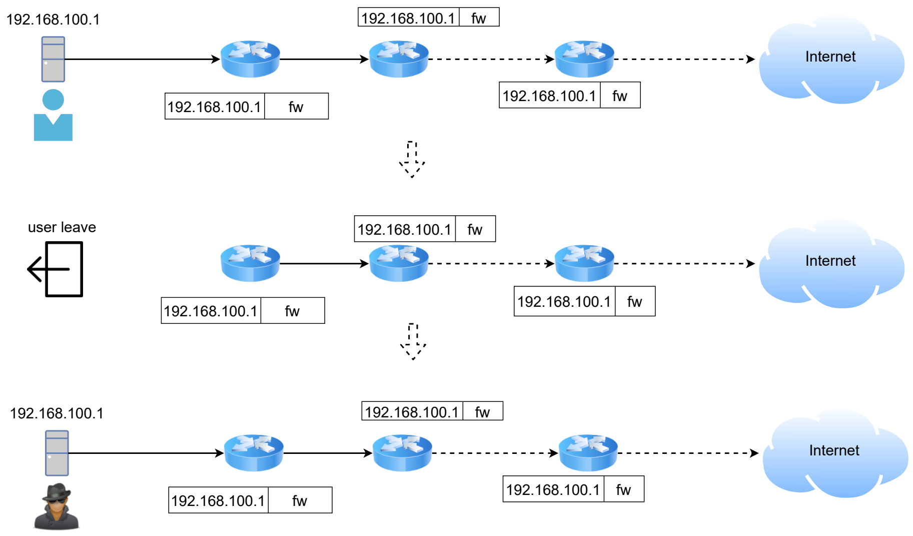


Figure 5.20: Illustration of malicious host gaining network access by using initially installed rules.

occur in the following switches. Some approaches like OBS [86] introduce forward rules in switches to represent rules added in previous switches, so a packet can only be applied one action and then forwarded to the destination. However, an update can then change the semantics of the filtering policy and create a security issue by allowing unwanted traffic to traverse the network. In this case, the administrator needs to run the decomposition and distribution algorithm from scratch on the entire network, which is costly, especially if updates are frequent. In our approach, we improve the performance of the update phase by only modifying partially the network rule tables.

5.5.1 Update Strategy With Generated Forward Rules

The easiest way to handle an update would be to re-run our distribution algorithm Algo 6 with the new ruleset. This solution is not efficient since many switches might not be affected by the modifications in the ruleset. Our approach permits that only switches concerned by adding, removing, or modifying a rule are considered. As a consequence, the time needed to perform the update is generally shorter. Moreover, our distribution algorithm can be applied almost directly by carefully selecting the affected subnetwork and rules as inputs.

Let R_i be a rule in the ruleset with priority i. We assume that the priority of R_i is higher than R_j if $i < j$. Let S_j represent Switch j. In the distribution phase, the controller maintains the locations where each rule has been added to the network. This information is specified by a couple

$(R, \mathcal{S})$ where R is a rule and $\mathcal{S}$ is the set of switches where R is installed.

To add a new rule R in a ruleset D, one must ensure that this rule will be installed on every path leading from the source node to the destination. First, we need to determine the set of switches where R needs to be installed. Based on the priority of R, we can determine the position of this rule in the distributed ruleset. If R needs to be installed between rules R_k and R_{k+1}, and based on $(R_k, \mathcal{S})$, the set of switches that may be affected by the update are the elements of $\mathcal{S}$ and all the switches that can be reached from them. S will be called *affect set* and denoted by $ASet$. In series-parallel graphs, each series component has one source and one destination node. To optimize the updating process, we only need to start the decomposition algorithm on the source switch of the minimal series component that contains $ASet$.

For instance, let $ASet = \{S_k, S_m, S_q\}$ be the *affect set* when adding a rule R_k as shown in Fig. 5.21 and let C_i be the smallest series component that contains $ASet$.

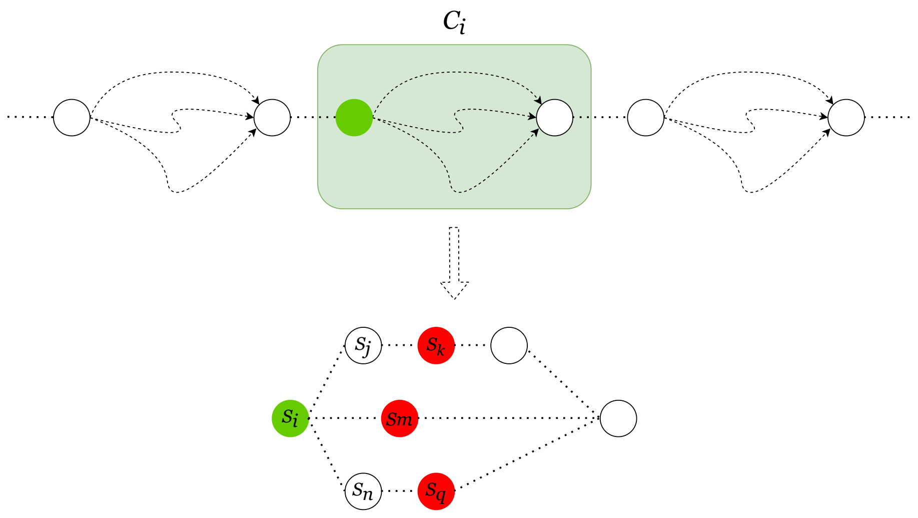


Figure 5.21: The smallest component C_i containing $ASet$.

We can find C_i as shown in Fig. 5.22 by running a preorder search in the binary tree representation of the network till we find the first node rooting a subtree that contains $ASet$. From this node, we can extract the first switch S_i, which will be a common switch of all paths from the source to the destination of C_i as shown in Fig. 5.21. Algo. 6 will be applied only to the (maximal) subnetwork with source S_i. We can now compute $D^{'}$, which contains the subset of D that can be encountered from the first rule inside S_i plus the new rule R, which is placed in the correct location based on its priority. After computing C_i and $D^{'}$, we run Algo. 6 to decompose and distribute the updated ruleset over the affected part of the network.

The algorithm will take as the first argument (N) the parent of C_i in the binary tree to decompose and distribute $D^{'}$ over all components starting from C_i to the last one.

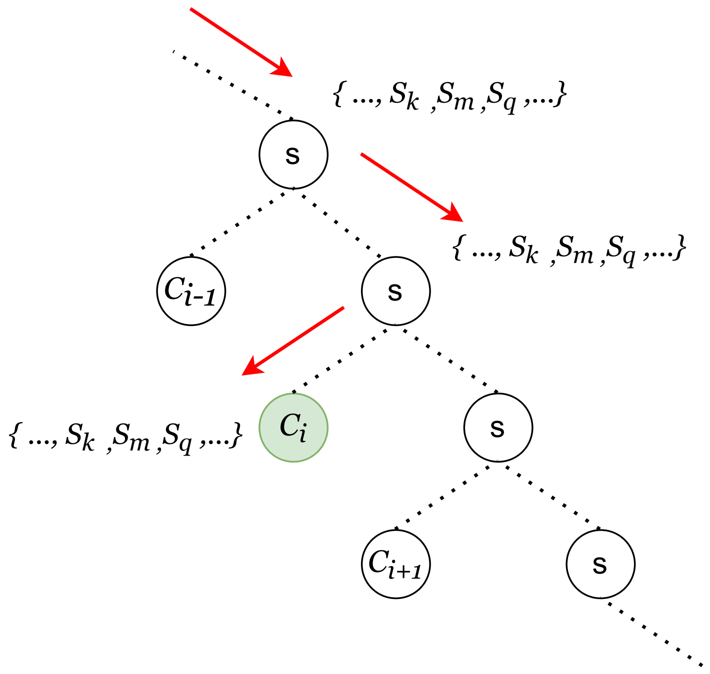

{ ..., S_k ,S_m ,S_q ,...}
s
{ ..., S_k ,S_m ,S_q ,...}
C_{i-1}
s
{ ..., S_k ,S_m ,S_q ,...}
C_i
s
C_{i+1}
s

Figure 5.22: Search the binary tree of the topology for C_i.

When deleting a rule R_i, its action field determines the update operations. For example, if R_i has a "Deny" action, no forward rule has been generated for it, so removing R_i from its switches is the only necessary operation.

When R_i has a "Forward" action, we need to delete every obsolete forward rule generated for R_i. The packets previously matching R_i are then handled by the rule R_j, $R_j >: R_i$ in the original ruleset excluding generated forward rules. Let R_k be the closest parent of R_i being an accepting rule. Two cases are possible:

- We have $R_k >: R_i$ and there are no additional forward rules with R_i's prefix (as they would be unnecessary), so we delete R_i and stop there.

- Some "Deny" rules $D = \{d \mid R_k > d > R_i\}, R_j \in D$ exist. They create conflicts with the rules $A = \{a \mid R_i >: a \wedge Action(a) = "Forward"\}$. A contains several elements if R_i is a single rule summarising higher-priority accepting rules as shown in Fig. 4.4. Let the generated rules $F = \{f \mid Action(f) = "Forward" \wedge (Pref(f) =$

$Pref(R_i) \vee R_k > f > R_i)\}$. If A is empty, every member of F is obsolete and must be removed from its switch. If A has only one element a, then a must replace every rule of F. Finally, when A contains several elements, each of them embodies a new forward rule. Thus, these elements of A must be handled via the adding strategy after removing the rules belonging to F from their respective switches.

We note that when no new forward rules must be added, the network is only affected between the switches, respectively containing R_i and R_k.

The modification of R_i's action follows a similar process. The directly related forward rules must be removed when switching from "Forward" to "Drop". Switching from "Drop" to "Forward" is analog to adding a rule because of the need to create forward rules, so the adding strategy will be applied. If the prefix of R_i changes, R_i would be deleted and then added with the new prefix.

5.5.2 Update Strategy With Two-Tier Approach

In the two-tier approach presented before, no forward rules are being generated. This means that the ruleset in S_i does not have dependency relations with the previous ones. This is particularly helpful in our update strategy to reduce the time needed to add, update or remove a rule from the ruleset.

If all the switches in $ASet$ have some space left when adding a new rule, the new rule can be added to these switches. In that case, there is no need to apply Algo. 6. If there is a switch in $ASet$ that cannot allocate space to this new rule, then one needs to call for Algo. 6.

When a rule modification occur, if the priority i of a rule R has been changed to j, and if $i > j$, $ASet$ will be computed from the new position of R, else if $i < j$, $ASet$ start from the initial position of R. In these two cases Algo. 6 must be used. This is because changing the priority of R will affect the order of all rules in the ruleset. Finally, removing a rule from the ruleset does not affect rules in other switches since we do not use forward rules. It can be done without the need to apply Algo. 6.

5.5.3 Evaluation

We evaluate the performance of the update strategy on series-parallel graphs while adding a new rule. The graphs are generated by varying the number of series components, the number of parallel paths in a component, the number of switches in a path, and the capacity of each switch. In the experiments, we compare the performance of our update approach, which relies on Algo. 6, with two strategies: Strategy 1 explores the whole network, while Strategy 2 explores only the part of the network affected by the update.

Rules added in switches are generated with ClassBench [128]. Each experiment is launched 50 times on each strategy with the same added rule and a different random priority. Experiments are conducted on a machine with AMD Ryzen 5 3600XT 3.79-GHz CPU, 16 GB of RAM, and running Windows 10 Pro. In our first experiment, the number of components in series varies from 10 to 50 with a set of 10k rules. Fig. 5.23 shows that Strategy 2 is always better than Strategy 1 with an update time reduction of up to 90% in some cases. When the priority of a new rule is low, the number of affected components is small, as well as the input binary tree of Algo. 6, resulting in a fast update.

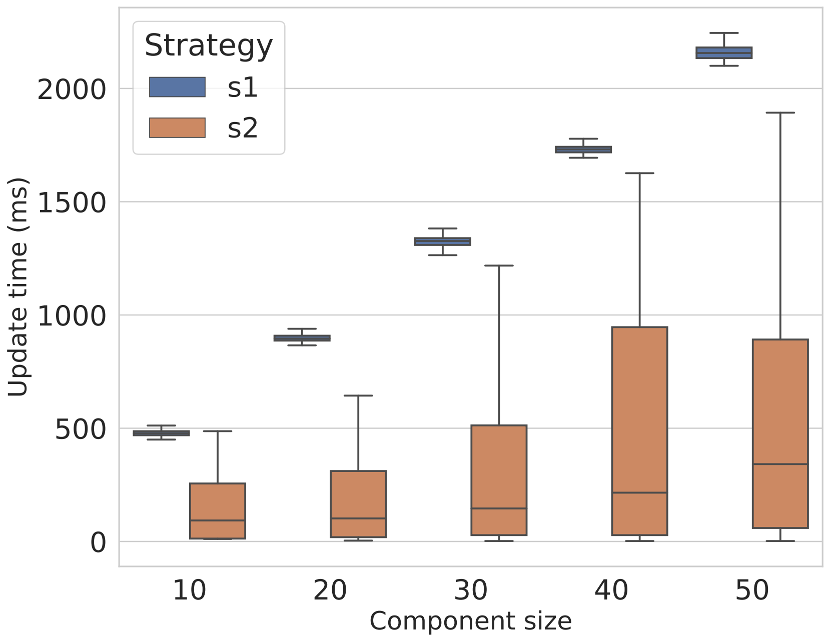


Figure 5.23: Update time using different series-parallel topologies size.

Fig. 5.24 shows the update time between respectively the two strategies of the update strategies while increasing the number of parallel paths in components. When the number of parallel paths increases, so does the size of the binary tree, which affects the time needed by the update algorithm. Since our approach uses a smaller binary tree, the update time will be smaller than Strategy 1.

In Fig. 5.25 we show the effect of the capacity of switches varying from 20 to 50 on the update time for each strategy. This experiment uses a

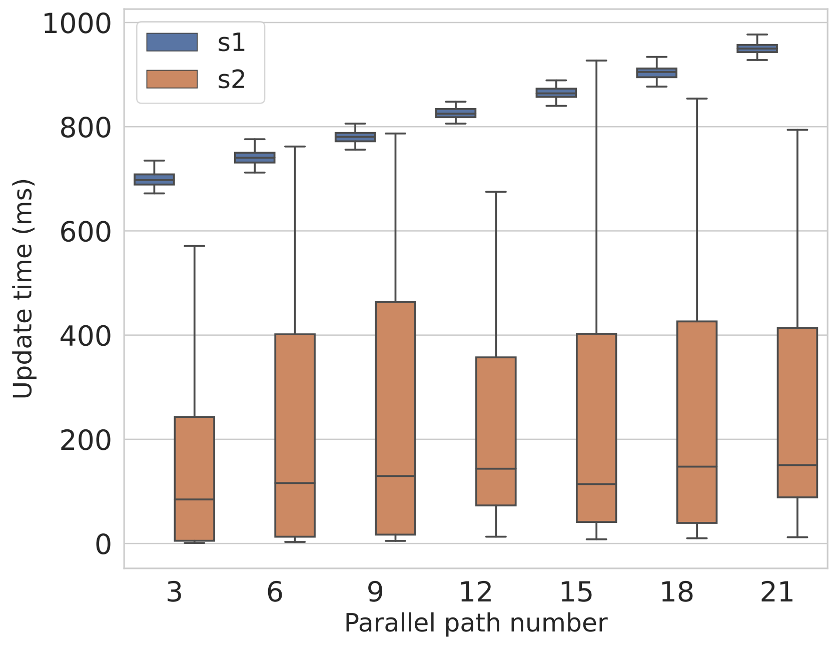


Figure 5.24: Update time of the two update strategies while increasing parallel path number.

topology with 30 components and three parallel paths, each containing five switches. When the capacity of switches increases, the time of the update also increases. Moreover, we rely on Algo. 6 to perform the changes in the switch tables. However, this algorithm uses a decomposition algorithm (Line 2 in Algo. 6) that needs more time to run if switches store more rules in their tables. As Strategy 1 recomputes every rule location, the time for the update will be higher than Strategy 2.

Updating a ruleset may require to remove some rule r. In the FoS approach, there will be no forward rules representing r. In this case, we only need to remove r from all rule tables in $ASet$. In the other approaches with forward rules, we need to apply our update strategy shown above on all affected parts of the network.

5.5.4 Network Topology Update

An update of the network topology can be done by adding or removing a switch in the network. Assuming that the modification preserves the series-parallel graph nature of the network, it triggers an update of the binary tree linked to that topology. A simple updating method would be to apply Algo. 6 with the same ruleset on the new binary tree. However,

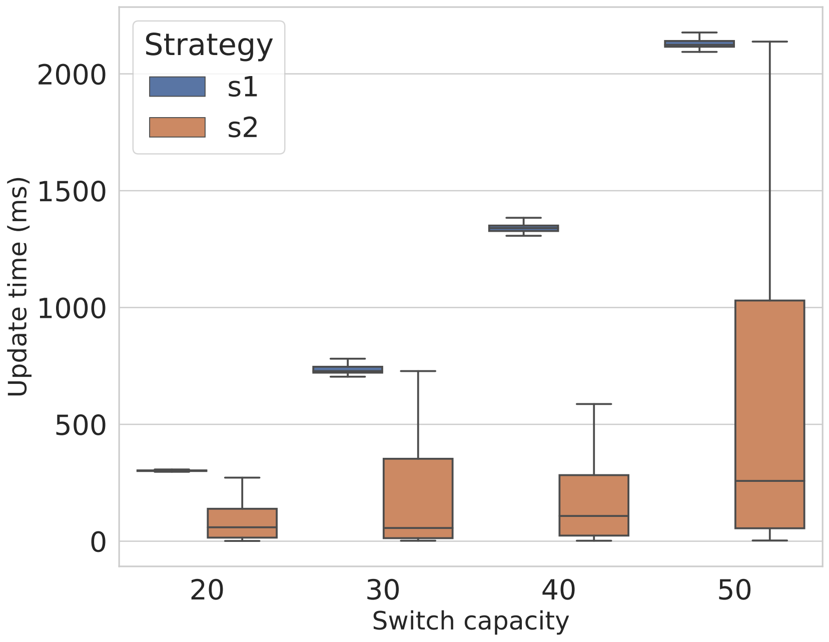


Figure 5.25: Update time of the two update strategies while increasing switch capacity.

with the same strategy as in subsection 5.5.1, we need only to apply the distribution algorithm to the smallest component where the modification occurs, and with a new ruleset not containing the rules stored in switches occurring before the first switch of that component.

5.6 Summary

In this chapter, we start by introducing our rule distribution algorithm for parallel series graphs. This algorithm handles any rules strategy and can work with existing decomposition algorithms like OBS [86]. Next, we generalize our distribution approach to work with all st-dags graphs, and we test our techniques on a real network called memorex. We then introduce a TWO-TIER approach for rule decomposition and distribution that uses two types of switches and does not generate additional rules. This approach is complementary to our distribution algorithm. Both can be used together to distribute rules over any st-dags without any additional rules. Next, we evaluate the performance of our TWO-TIER approach and compare the result with existing techniques. Since we do not generate or duplicate rules, the overhead of our approach is close to null. Furthermore, we introduce an update strategy for series-parallel graphs that rely on the network's affected part by an update instead of the whole network. Finally, we evaluate the performance of our update technique using our distribution algorithm. The results show a reduction of up to 90% in update time.

Chapter 6

General Conclusion

With the growth of users, connected devices, and data sizes on the internet, efficient rules management is necessary. Incorrect or neglected management can cause millions of dollars in loss, leading to a security breach or delay sensitive information. In addition, this growth leads to increased demand for resources like memory capacity, process power, electricity, etc., for network devices. Moreover, the number of entries in network devices has been increasing rapidly. As seen in Fig. 1.1, the number of entries has reached 1 million in BGP tables and keeps growing.

For an network management process, Software-Defined networks have been proposed. This technique decouples the control layer from the data layer and offers a global view of the network for easy management. However, the increase in the number of entries makes the classification process a bottleneck. Moreover, some SDN switches rely on TCAMs memory to store rules. Although TCAMs permit a high-speed classification process, they are expensive and suffer from high power consumption leading to heat problems affecting the performance. On the other side, the controller decides the position of rules for any new flow of data based on rulesets already defined. Minimizing the size of these rule sets will accelerate the classification process and make rule management easier. For that, we need a new representation for addresses fields. This new encoding has to work for all types of intervals like port or IP ranges leading to higher compression in entry tables, blacklists, intrusion detection systems, etc...

However, the compression technique cannot always be efficient. For example, sometimes switch capacity is too small, and the compression ratio is not sufficient. In this case, the set of rules needs to be decomposed over multiple switches. Then, the compression technique can be applied to the subset of rules in each of the switches. However, we need first to decompose the original set into multiple subsets of rules and distribute them alongside the network. In addition, and since rules depend on each other, and the priority plus the order of rules installed in switches play a crucial role in handling incoming packets, suitable decomposition techniques need to be developed. These techniques have to respect the semantics of the original set. Many solutions and propositions have been developed in the literature, but they suffer from high overhead, security concerns, hardware modification cost, and thus affecting the performance of the classification process.

The problem of rule distribution, on the other hand, needs to be tackled. With the complex topologies nowadays, the position of each switch can have a significant impact on the rule placement. Switches belonging to multiple paths have to store the proper rules for each path. However, with the storage limitation of these devices, suitable techniques have to be developed to ensure that each rule is installed in the proper position. These problems can take advantage of the SDN network facilities by implementing solutions on the controller side for an easy global management.

6.1 Achievements

This thesis proposes several contributions to solve the rules management problem by relying on rules compression and distribution.

State of the art in Chapter 2 showed that the proposed approaches presented to resolve TCAMs problem could be improved to achieve better performance. For example, solutions regarding range expansion can benefit from replacing the standard ternary format with a new representation of port range.

Double Mask notation for IP addresses and/or ports range allows one to minimize the number of entries in rulesets. We have developed a linear algorithm that generates all masks covering a range. We have then showed by simulations that the *Double Mask* approach could indeed increase the compression ratio, especially when the number of exceptions is high. Next, we have developed a real SDN testbed with an OpenFlow-enabled Zodiac-FX switch and with a RYU controller. We have implemented *Double Mask* in Openflow protocol. We have also extended the matching function to handle rules with *Double Mask*. Our experiments show that the extra cost of matching *Double Mask* patterns is balanced by the gain in rulesets size reduction. Moreover, the gain in the controller response time could be more than 80% with a compression ratio of 83%, leading to faster installation of new flows in switches.

We have also tackled the rules distribution problem for LPM strategy for one dimension rulesets. Although we rely on forward rules as previous approaches, our solution leverages the action field of rules to reduce the number of required forward rules. For instance, we have observed that deny rules do not need to be represented in the following switches. If all rules have the same action field "deny", no forward rules will be generated. Moreover, if the action field for all rules is "accept", forward rules can be merged in the following switches to reduce the number of generated rules, also leading to a smaller overhead.

We have then developed a general ruleset distribution algorithm for series-parallel graphs. We have extended this algorithm to st-dags graph. In the next step, we have introduced Two-Tier approach for rule distribution. This simpler approach does not generate any forward rules nor decision tree, reducing by that the overhead plus the distribution time since. In simulations the Two-Tier approach has the lowest overhead compared to the literature and does not modify any packet unlike a close competitor. The simulations also show that the Two-Tier approach has lower latency and higher bandwidth than standard approaches, since a packet does not need to traverse all the networks to the destination. Finally, we introduce an update strategy that allows one to minimize the modification to the switches' memory. The simulations show that the update time can be reduced by more than 90%.

Table. 6.1 summarizes all developed solutions for the rule distribution problem with two rule strategy LPM and Priority.

	Dimension	**LPM**	**Priority**	
Single Path	1	Apply Algo. 5 with forward rules	Two-Tier	
	>1	N/A (Overlapping rules/ No decision can be made)		
St-Dags	1	Algo. 6 +Modified version of Algo. 5 without forwad rules +Two-Tier	Algo. 6	
			Either	Or
	>1	N/A (Overlapping rules /No decision can be made)	OBS/Palette With forward rules	Two-Tier Without forward rules

Table 6.1: Summary of all rule distribution solutions proposed in this thesis.

6.2 Limitations

A Double Mask rule can both accept and deny addresses with mask1 and mask2 respectively. If no address is denied, mask2 gets useless, and Double Mask is reduced to a simple mask. In this case, no compression can be expected. Moreover, since matching a simple mask is faster than with a Double Mask, it is simpler to stick to simple masks.

As for our distribution algorithm, we show that we can simplify an st-dag graph to a series-parallel graph up to merging some vertices of the st-dag graph. However, with some complex topologies where the degree of nodes is significant, the simplification needs many merging phases. In this case, the simplified graph will not resemble the initial one, and this entails a lower quality for the distribution solutions derived from the simplified graph.

In addition, as seen from Table. 6.1, no satisfying LPM strategy for multiple dimensions has been developed. This is due to the overlapping fields in rules. For example, let Rule R_1 have a more specific source field (resp. destination) and Rule R_2 with a more specific destination field (resp. source). In LPM, a packet matches the more specific field each time. However, in this example, the packet will match one field from each rule, and no decision can be made.

6.3 Future Work

This thesis opens to more improvements in rule management for software-defined networks. Furthermore, we can identify multiple research directions to overcome the limitations of the proposed solutions in this work.

First, we can extend our compression solution with Double Mask to port ranges and IPv6 addresses since they can also be represented as binary strings. By changing the length of the string in the input, our algorithm can quickly generate the corresponding masks. In addition, this representation can be enhanced by computing *Double Masks* for the union of ranges to achieve a higher level of optimization in routing tables. Our current technique compute masks for each of the ranges independently of the others. However, in some cases, *Double Masks* can be extended to cover multiple ranges simultaneously as discussed in Subsection 3.6. In addition, we can apply our approach alongside other techniques to remove redundant or shadowed rules. Thus reducing the original ruleset and accelerating the computation of Double Masks with our algorithm.

Regarding the distribution part of this thesis, we can automatize all the processes. For instance, we can rely on machine learning techniques to infer communication patterns and generate accordingly the required filtering rules to be distributed by our proposed algorithms. Moreover, our Two-Tier approach can be extended to multiple FoS switches for a specific application. For example, security applications can rely on switches with enforced security protocols, while for streaming applications FoS switches with fewer security requirements are sufficient. In this case, FiS switches will decide where to forward a matched packet based on the application.

This thesis can be extended by employing (deep) reinforcement learning for distributing rules. These techniques have been already applied to multiple network optimisation problems and could be good candidates for rules placement and distribution. Thus using both inference techniques to mine the required filtering policies from the network behavior and a reinforcement learning mechanism to enforce such policies will close the loop of an autonomous filtering function.

Bibliography

[1] Internet of things (iot) and non-iot active device connections worldwide from 2010 to 2025. Accessed : 2021-06-15. [Online]. Available: https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/

[2] Data never sleeps. Accessed : 2021-06-15. [Online]. Available: https://www.domo.com/solution/data-never-sleeps-6

[3] How much data is generated each day? Accessed : 2021-06-15. [Online]. Available: https://www.weforum.org/agenda/2019/04/how-much-data-is-generated-each-day-cf4bddf29f/

[4] The value of a millisecond: Finding the optimal speed of a trading infrastructure. Accessed : 2021-06-15. [Online]. Available: https://research.tabbgroup.com/report/v06-007-value-millisecond-finding-optimal-speed-trading-infrastructure

[5] Cost of a data breach. Accessed : 2021-07-15. [Online]. Available: https://www.venafi.com/yahoo-cost-of-data-breach

[6] H. Hammouchi, O. Cherqi, G. Mezzour, M. Ghogho, and M. El Koutbi, "Digging deeper into data breaches: An exploratory data analysis of hacking breaches over time," *Procedia Computer Science*, vol. 151, pp. 1004–1009, 2019.

[7] A. Niakanlahiji, M. M. Pritom, B.-T. Chu, and E. Al-Shaer, "Predicting zero-day malicious ip addresses," 11 2017, pp. 1–6.

[8] The 768k or another internet doomsday? heres how to deal with the tcam overflow at the 768k boundary. Accessed : 2021-06-22. [Online]. Available: https://www.noction.com/blog/768k-day-512k-tcam

[9] Active bgp entries. Accessed : 2021-06-22. [Online]. Available: https://www.cidr-report.org/cgi-bin/plota?file=%2fvar%2fdata%2fbgp%2fas2.0%2fbgp%2dactive%2etxt&descr=Active%20BGP%20entries%20%28FIB%29&ylabel=Active%20BGP%20entries%20%28FIB%29&with=step

[10] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, Y. Ganjali, and C. Diot, “Characterization of failures in an operational ip backbone network,” *IEEE/ACM Transactions on Networking*, vol. 16, no. 4, pp. 749–762, 2008.

[11] A. Abboud, R. Garcia, A. Lahmadi, M. Rusinowitch, and A. Bouhoula, “Efficient distribution of security policy filtering rules in software defined networks,” in *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*, 2020, pp. 1–10.

[12] A. Abboud, R. Garcia, A. Lahmadi, M. Rusinowitch, and A. Bouhoula, “R2-d2: Filter rule set decomposition and distribution in software defined networks,” in *2020 16th International Conference on Network and Service Management (CNSM)*. IEEE, 2020, pp. 1–4.

[13] A. Abboud, A. Lahmadi, M. Rusinowitch, M. Couceiro, A. Bouhoulal, and M. Avadi, “Double mask: An efficient rule encoding for software defined networking,” in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2020, pp. 186–193.

[14] A. Abboud, A. Lahmadi, M. Rusinowitch, M. Couceiro, and A. Bouhoula, “Poster: Minimizing range rules for packet filtering using a double mask representation,” in *2019 IFIP Networking Conference (IFIP Networking)*. IEEE, 2019, pp. 1–2.

[15] P. Gupta and N. McKeown, “Algorithms for packet classification,” *IEEE Network*, vol. 15, no. 2, pp. 24–32, 2001.

[16] C. R. Meiners, A. X. Liu, and E. Torng, “TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs,” in *2007 IEEE International Conference on Network Protocols*, Oct 2007, pp. 266–275.

[17] V. Fuller, T. Li, J. Yu, and K. Varadhan, “Classless inter-domain routing (CIDR): An address assignment and aggregation strategy,” United States, 1993.

[18] F. Yu and R. Katz, “Efficient multi-match packet classification with tcam,” in *Proceedings. 12th Annual IEEE Symposium on High Performance Interconnects*, 2004, pp. 28–34.

[19] F. Yu, T. Lakshman, M. Motoyama, and R. Katz, “Ssa: a power and memory efficient scheme to multi-match packet classification.” 01 2005, pp. 105–113.

[20] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat, "Optimal in/out tcam encodings of ranges," *IEEE/ACM Transactions on Networking*, vol. 24, no. 1, pp. 555–568, 2015.

[21] B. Schieber, D. Geist, and A. Zaks, "Computing the minimum DNF representation of boolean functions defined by intervals," *Discrete Applied Mathematics*, vol. 149, no. 1, pp. 154 – 173, 2005.

[22] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary cams," in *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 193204. [Online]. Available: https://doi.org/10.1145/1080091.1080115

[23] E. Spitznagel, D. Taylor, and J. Turner, "Packet classification using extended TCAMs," in *11th IEEE International Conference on Network Protocols, 2003. Proceedings.*, Nov 2003, pp. 120–131.

[24] V. Ravikumar, R. Mahapatra, and L. N. Bhuyan, "Easecam: an energy and storage efficient tcam-based router architecture for ip lookup," *IEEE Transactions on Computers*, vol. 54, no. 5, pp. 521–533, 2005.

[25] Y. Ma and S. Banerjee, "A smart pre-classifier to reduce power consumption of tcams for multi-dimensional packet classification," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, p. 335346, Aug. 2012. [Online]. Available: https://doi.org/10.1145/2377677.2377749

[26] W. Li, D. Li, X. Liu, T. Huang, X. Li, W. Le, and H. Li, "A power-saving pre-classifier for tcam-based ip lookup," *Computer Networks*, vol. 164, p. 106898, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128618311927

[27] B. Agrawal and T. Sherwood, "Modeling tcam power for next generation network devices," in *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, 2006, pp. 120–129.

[28] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite cacheflow in software-defined networks," *HotSDN 2014 - Proceedings of the ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking*, 08 2014.

[29] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM 2011*

Conference, ser. SIGCOMM '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 254265. [Online]. Available: https://doi.org/10.1145/2018436.2018466

[30] C. R. Meiners, A. X. Liu, and E. Torng, "Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs," *IEEE/ACM Transactions on Networking*, vol. 20, no. 2, pp. 488–500, April 2012.

[31] A. X. Liu, C. R. Meiners, and Y. Zhou, "All-match based complete redundancy removal for packet classifiers in tcams," in *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*, 2008, pp. 111–115.

[32] K. Kannan and S. Banerjee, "Compact tcam: Flow entry compaction in tcam for power aware sdn," in *International conference on distributed computing and networking.* Springer, 2013, pp. 439–444.

[33] P. Gupta, "Packet classification using hierarchical intelligent cuttings," 1999.

[34] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 213224. [Online]. Available: https://doi.org/10.1145/863955.863980

[35] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, "Packet classification algorithms: From theory to practice," in *IEEE INFOCOM 2009*, 2009, pp. 648–656.

[36] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "Efficuts: Optimizing packet classification for memory and throughput," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, p. 207218, Aug. 2010. [Online]. Available: https://doi.org/10.1145/1851275.1851208

[37] W. Li, X. Li, H. Li, and G. Xie, "Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 2645–2653.

[38] W. Lu and S. Sahni, "Packet classification using two-dimensional multibit tries," in *10th IEEE Symposium on Computers and Communications (ISCC'05)*, 2005, pp. 849–854.

[39] Y. Sun and M. S. Kim, "Tree-based minimization of TCAM entries for packet classification," in *2010 7th IEEE Consumer Communications and Networking Conference*, Jan 2010, pp. 1–5.

[40] H. Liu, "Efficient mapping of range classifier into ternary-cam," in *Proceedings 10th Symposium on High Performance Interconnects*, Aug 2002, pp. 95–100.

[41] A. Bremler-Barr and D. Hendler, "Space-Efficient TCAM-Based Classification Using Gray Coding," *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 18–30, Jan 2012.

[42] C. R. Meiners, A. X. Liu, and E. Torng, "Topological transformation approaches to optimizing tcam-based packet classification systems," in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 7384. [Online]. Available: https://doi.org/10.1145/1555349.1555359

[43] J. van Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 4, pp. 560–571, 2003.

[44] D. Pao, P. Zhou, B. Liu, and X. Zhang, "Enhanced prefix inclusion coding filter-encoding algorithm for packet classification with ternary content addressable memory."

[45] H. Che, Z. Wang, K. Zheng, and B. Liu, "Dres: Dynamic range encoding scheme for tcam coprocessors," *IEEE Transactions on Computers*, vol. 57, no. 7, pp. 902–915, 2008.

[46] N. B. Neji and A. Bouhoula, "Naf conversion: An efficient solution for the range matching problem in packet filters," in *2011 IEEE 12th International Conference on High Performance Switching and Routing*, July 2011, pp. 24–29.

[47] B. Leng, L. Huang, C. Qiao, H. Xu, and X. Wang, "Ftrs: A mechanism for reducing flow table entries in software defined networks," *Computer Networks*, vol. 122, pp. 1 – 15, 2017.

[48] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla, "Packet classifiers in ternary cams can be smaller," *SIGMETRICS Perform. Eval. Rev.*, vol. 34, no. 1, p. 311322, Jun. 2006. [Online]. Available: https://doi.org/10.1145/1140103.1140313

[49] A. X. Liu and M. G. Gouda, "Complete redundancy removal for packet classifiers in tcams," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 4, pp. 424–437, 2010.

[50] R. McGeer and P. Yalagandula, "Minimizing rulesets for tcam implementation," in *IEEE INFOCOM 2009*, 2009, pp. 1314–1322.

[51] M. Rifai, N. Huin, C. Caillouet, F. Giroire, D. Lopez-Pacheco, J. Moulierac, and G. Urvoy-Keller, "Too many sdn rules? compress them with minnie," in *2015 IEEE Global Communications Conference (GLOBECOM)*, 2015, pp. 1–7.

[52] A. X. Liu, E. Torng, and C. R. Meiners, "Firewall compressor: An algorithm for minimizing firewall policies," in *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*, April 2008, pp. 176–180.

[53] M. Yoon, S. Chen, and Z. Zhang, "Reducing the size of rule set in a firewall," in *2007 IEEE International Conference on Communications*, June 2007, pp. 1274–1279.

[54] A. Bouhoula, Z. Trabelsi, E. Barka, and M. Anis Benelbahri, "Firewall filtering rules analysis for anomalies detection," *IJSN*, vol. 3, pp. 161–172, 01 2008.

[55] M. Casado, *Architectural support for security management in enterprise networks.* Citeseer, 2007, vol. 68, no. 09.

[56] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, p. 6974, Mar. 2008. [Online]. Available: https://doi.org/10.1145/1355734.1355746

[57] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[58] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[59] H. Song, "Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 127132. [Online]. Available: https://doi.org/10.1145/2491185.2491190

[60] M. Smith, R. E. Adams, M. Dvorkin, Y. Laribi, V. Pandey, P. Garg, and N. Weidenbacher, "OpFlex Control Protocol," Internet Engineering Task Force, Internet-Draft draft-smith-opflex-03, Apr. 2016, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-smith-opflex-03

[61] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, p. 4451, Apr. 2014. [Online]. Available: https://doi.org/10.1145/2602204.2602211

[62] E. B. Pfaff, B. Davie, "The open vswitch database management protocol," 2013. [Online]. Available: https://www.hjp.at/doc/rfc/rfc7047.html

[63] B. Belter, A. Binczewski, K. Dombek, A. Juszczyk, L. Ogrodowczyk, D. Parniewicz, M. Stroiñski, and I. Olszewski, "Programmable abstraction of datapath," in *2014 Third European Workshop on Software Defined Networks*, 2014, pp. 7–12.

[64] R. Braga, E. Mota, and A. Passito, "Lightweight ddos flooding attack detection using nox/openflow," in *IEEE Local Computer Network Conference*, 2010, pp. 408–415.

[65] G. Yao, J. Bi, and P. Xiao, "Source address validation solution with openflow/nox architecture," in *2011 19th IEEE International Conference on Network Protocols*, 2011, pp. 7–12.

[66] N. Handigol, M. Flajslik, S. Seetharaman, N. McKeown, and R. Johari, "Aster* x: Load-balancing as a network primitive," in *9th GENI Engineering Conference (Plenary)*, 2010, pp. 1–2.

[67] C. A. Macapuna, C. E. Rothenberg, and M. F. Maurício, "In-packet bloom filter based data center networking with distributed openflow controllers," in *2010 IEEE Globecom Workshops.* IEEE, 2010, pp. 584–588.

[68] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, 2011, pp. 1–12.

[69] M. V. Neves, C. A. De Rose, K. Katrinis, and H. Franke, "Pythia: Faster big data in motion through predictive software-defined network optimization at runtime," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium.* IEEE, 2014, pp. 82–90.

[70] H. Ali-Ahmad, C. Cicconetti, A. de la Oliva, M. Dräxler, R. Gupta, V. Mancuso, L. Roullet, and V. Sciancalepore, "Crowd: an sdn approach for densenets," in *2013 second European workshop on software defined networks.* IEEE, 2013, pp. 25–31.

[71] J. Vestin, P. Dely, A. Kassler, N. Bayer, H. Einsiedler, and C. Peylo, "Cloudmac: torwards software defined wlans," in *Proceedings of the 18th annual international conference on Mobile computing and networking*, 2012, pp. 393–396.

[72] Y. Yamasaki, Y. Miyamoto, J. Yamato, H. Goto, and H. Sone, "Flexible access management system for campus vlan based on openflow," in *2011 IEEE/IPSJ International Symposium on Applications and the Internet*. IEEE, 2011, pp. 347–351.

[73] K.-K. Yap, M. Kobayashi, R. Sherwood, T.-Y. Huang, M. Chan, N. Handigol, and N. McKeown, "Openroads: Empowering research in mobile networks," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, pp. 125–126, 2010.

[74] N. L. Van Adrichem, C. Doerr, and F. A. Kuipers, "Opennetmon: Network monitoring in openflow software-defined networks," in *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 2014, pp. 1–8.

[75] C. Argyropoulos, D. Kalogeras, G. Androulidakis, and V. Maglaris, "Paflomon–a slice aware passive flow monitoring framework for openflow enabled experimental facilities," in *2012 European Workshop on Software Defined Networking*. IEEE, 2012, pp. 97–102.

[76] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "Payless: A low cost network monitoring framework for software defined networks," in *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 2014, pp. 1–9.

[77] G. Wang, T. E. Ng, and A. Shaikh, "Programming your network at run-time for big data applications," in *Proceedings of the first workshop on Hot topics in software defined networks*, 2012, pp. 103–108.

[78] T. Benson, A. Akella, A. Shaikh, and S. Sahu, "Cloudnaas: a cloud networking platform for enterprise applications," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011, pp. 1–13.

[79] E. Keller, S. Ghorbani, M. Caesar, and J. Rexford, "Live migration of an entire network (and its hosts)," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, 2012, pp. 109–114.

[80] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Avant-guard: Scalable and vigilant switch flow management in software-defined networks," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 413–424.

[81] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, “A security enforcement kernel for openflow networks,” in *Proceedings of the first workshop on Hot topics in software defined networks*, 2012, pp. 121–126.

[82] K. Giotis, G. Androulidakis, and V. Maglaris, “Leveraging sdn for efficient anomaly detection and mitigation on legacy networks,” in *2014 Third European Workshop on Software Defined Networks*. IEEE, 2014, pp. 85–90.

[83] K. Kogan, S. Nikolenko, W. Culhane, P. Eugster, and E. Ruan, “Towards efficient implementation of packet classifiers in sdn/openflow,” in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013, pp. 153–154.

[84] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, “Towards an elastic distributed sdn controller,” in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013, pp. 7–12.

[85] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, “Rules placement problem in openflow networks: A survey,” *IEEE Communications Surveys & Tutorials*, vol. 18, pp. 1–1, 12 2015.

[86] N. Kang, Z. Liu, J. Rexford, and D. Walker, “Optimizing the "one big switch" abstraction in software-defined networks,” 12 2013, pp. 13–24.

[87] W. Li and X. Li, “Hybridcuts: A scheme combining decomposition and cutting for packet classification,” in *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, 2013, pp. 41–48.

[88] J. Fong, X. Wang, Y. Qi, J. Li, and W. Jiang, “Parasplit: A scalable architecture on fpga for terabit packet classification,” in *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*, 2012, pp. 1–8.

[89] Y. Kanizo, D. Hay, and I. Keslassy, “Palette: Distributing tables in software-defined networks,” in *2013 Proceedings IEEE INFOCOM*, 2013, pp. 545–549.

[90] P. Chuprikov, K. Kogan, and S. Nikolenko, “How to implement complex policies on existing network infrastructure,” in *Proceedings of the Symposium on SDN Research*, ser. SOSR 18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3185467.3185477

[91] P. Kannan, M. Chan, R. Ma, and E.-C. Chang, “Raptor: Scalable rule placement over multiple path in software defined networks,” 06 2017, pp. 1–9.

[92] X. Li and W. Xie, “Craft: A cache reduction architecture for flow tables in software-defined networks,” in *2017 IEEE Symposium on Computers and Communications (ISCC)*, 2017, pp. 967–972.

[93] J.-F. Huang, G.-Y. Chang, C.-F. Wang, and C.-H. Lin, “Heterogeneous flow table distribution in software-defined networks,” *IEEE Transactions on Emerging Topics in Computing*, vol. 4, no. 2, pp. 252–261, 2016.

[94] A. Marsico, R. Doriguzzi-Corin, and D. Siracusa, “Overcoming the memory limits of network devices in sdn-enabled data centers,” in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2017, pp. 897–898.

[95] H. Li, S. Guo, C. Wu, and J. Li, “Fdrc: Flow-driven rule caching optimization in software defined networking,” in *2015 IEEE International Conference on Communications (ICC)*, 2015, pp. 5777–5782.

[96] A. Mimidis-Kentis, A. Pilimon, J. Soler, M. Berger, and S. Ruepp, “A novel algorithm for flow-rule placement in sdn switches,” in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018, pp. 1–9.

[97] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, “Scalable flow-based networking with difane,” *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 351–362, 2010.

[98] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, “Officer: A general optimization framework for openflow rule allocation and endpoint policy enforcement,” in *2015 IEEE Conference on Computer Communications (INFOCOM)*, 2015, pp. 478–486.

[99] H. Huang, P. Li, S. Guo, and B. Ye, “The joint optimization of rules allocation and traffic engineering in software defined network,” in *2014 IEEE 22nd International Symposium of Quality of Service (IWQoS)*, 2014, pp. 141–146.

[100] H. Li, P. Li, and S. Guo, “Morule: Optimized rule placement for mobile users in sdn-enabled access networks,” in *2014 IEEE Global Communications Conference*, 2014, pp. 4953–4958.

[101] W. Li, Z. Qin, K. Li, H. Yin, and L. Ou, “A novel approach to rule placement in software-defined networks based on optree,” *IEEE Access*, vol. 7, pp. 8689–8700, 2019.

[102] Latency is everywhere and it costs you sales - how to crush it. Accessed : 2021-07-5. [Online]. Available: http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it

[103] K. Qiu, J. Yuan, J. Zhao, X. Wang, S. Secci, and X. Fu, "Fastrule: Efficient flow entry updates for tcam-based openflow switches," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 484–498, 2019.

[104] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, ser. HotNets-XII. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2535771.2535791

[105] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 323334. [Online]. Available: https://doi.org/10.1145/2342356.2342427

[106] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 1526. [Online]. Available: https://doi.org/10.1145/2486001.2486012

[107] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 4954. [Online]. Available: https://doi.org/10.1145/2491185.2491191

[108] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zupdate: Updating data center networks with zero loss," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013, pp. 411–422.

[109] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. USA: USENIX Association, 2014, p. 533546.

[110] Y.-W. Chang and T.-N. Lin, "An efficient dynamic rule placement for distributed firewall in sdn," in *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, 2020, pp. 1–6.

[111] X. Wen, B. Yang, Y. Chen, L. E. Li, K. Bu, P. Zheng, Y. Yang, and C. Hu, "Ruletris: Minimizing rule update latency for tcam-based sdn switches," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016, pp. 179–188.

[112] Symantec, *Internet Security Threat Report*, April 2017. [Online]. Available: https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf

[113] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 4, pp. 3–14, Oct. 1997.

[114] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla, "Packet classifiers in ternary cams can be smaller," *SIGMETRICS Perform. Eval. Rev.*, vol. 34, no. 1, pp. 311–322, Jun. 2006.

[115] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat, "On finding an optimal TCAM encoding scheme for packet classification," in *2013 Proceedings IEEE INFOCOM*, April 2013, pp. 2049–2057.

[116] A. Bouhoula and N. B. Neji, "Double-masked IP filter," French Patent FR3011705, April 2015.

[117] R. Cohen and D. Raz, "Simple efficient TCAM based range classification," in *2010 Proceedings IEEE INFOCOM*, March 2010, pp. 1–5.

[118] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy, "Exact worst case TCAM rule expansion," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1127–1140, June 2013.

[119] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 191–202, Oct. 1998.

[120] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol. 37, no. 3, pp. 238–275, Sep. 2005.

[121] Ipv4 geolocation. Accessed: 07/01/2020. [Online]. Available: https://datahub.io/JohnSnowLabs/ipv4-geolocation

[122] NorthboundNetworks, *Zodiac Fx switch.* [Online]. Available: https://github.com/NorthboundNetworks/ZodiacFX

[123] *Ryu OpenFlow controller.* [Online]. Available: https://osrg.github.io/ryu/

[124] B. LeCun, T. Mautor, F. Quessette, and M.-A. Weisser, "Bin packing with fragmentable items: Presentation and approximations," *Theoretical Computer Science*, 01 2013.

[125] P. Ferguson and D. Senie, "Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing, bcp 38, rfc 2827," May 2000. [Online]. Available: https://www.rfc-editor.org/info/bcp38

[126] J. N. D. Gupta and J. C. Ho, "A new heuristic algorithm for the one-dimensional bin-packing problem," *Production Planning & Control*, vol. 10, no. 6, pp. 598–603, 1999.

[127] *Code for Palette, OBS and OneBit simulations*, 2017. [Online]. Available: https://github.com/distributedpolicies/submission

[128] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," *IEEE/ACM transactions on networking*, vol. 15, no. 3, pp. 499–511, 2007.

[129] R. Duffin, "Topology of series-parallel networks," *Journal of Mathematical Analysis and Applications.* [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022247X65901253

[130] P. Flocchini and F. L. Luccio, "Routing in series parallel networks," *Theory of Computing Systems*, 2003. [Online]. Available: https://doi.org/10.1007/s00224-002-1033-y

[131] J. Valdes, R. E. Tarjan, and E. L. Lawler, "The recognition of series parallel digraphs," in *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, ser. STOC 79. New York, NY, USA: Association for Computing Machinery, 1979, p. 112. [Online]. Available: https://doi.org/10.1145/800135.804393

[132] Reference network. Accessed : 2021-05-25. [Online]. Available: http://www.av.it.pt/anp/on/refnet2.html

[133] Y. Xia, X. Sun, S. Dzinamarira, D. Wu, X. Huang, and T. Ng, "A tale of two topologies: Exploring convertible data center network architectures with flat-tree," 08 2017, pp. 295–308.

List of Figures

List of Tables

Appendix A

Résumé de la thèse en français

A.1 Context

La taille et la complexité croissantes des topologies de réseau rendent difficile la gestion des périphériques réseau. En 2021, par exemple, le nombre estimé de dispositifs IoT embarqués avec des capteurs et des logiciels pour partager des données sur Internet est d'environ 13,8 milliards dans le monde. Ce nombre atteindra 30,9 milliards dans seulement quatre ans [1]. Ce nombre énorme d'appareils connectés génère plus de 2,5 quintillions d'octets de données par jour [2], et d'ici 2025, nous créerons 463 exaoctets de données chaque jour [3]. En outre, le type de données transférées sur Internet est important, et tout retard ou toute faille de sécurité peut entraîner des pertes de plusieurs millions de dollars. Une étude du TABB Group montre que si une plateforme de courtage électronique avait un retard de 5 millisecondes, elle pourrait perdre au moins 4 millions de dollars par milliseconde [4]. Par ailleurs, une violation des données de Yahoo coûte à l'entreprise environ 4,48 milliards de dollars [5], tandis que la perte résultant d'un autre piratage est estimée à 444 milliards de dollars [6] pour la même entreprise. Cela montre à quel point une attaque peut être grave et coûteuse et incite la communauté scientifique à trouver de nouvelles solutions pour s'adapter aux nouveaux défis.

Au cours de la dernière décennie, une nouvelle tendance en matière de réseaux, appelée "réseau défini par logiciel" ou "Software Defined Network" (SDN), a été développée pour améliorer la gestion des réseaux. Le SDN découple le plan de contrôle et le plan de données qui communiquent entre eux grâce au protocole OpenFlow. Grâce à ce découplage, les dispositifs du réseau sont réduits à de simples dispositifs de transfert, ce qui permet une classification plus rapide des paquets. Egalement, les tâches de contrôle du flux de données sont assignées à une entité appelée le contrôleur.

Cette nouvelle approche permet à l'administrateur d'avoir une vue globale du réseau et de simplifier la mise à jour et la résolution des problèmes, améliorant ainsi les performances des dispositifs et des contrôleurs compatibles avec le SDN.

Dans la même optique, la croissance du trafic a un impact sur la taille des tables de routage qui sont au coeur des routeurs de réseaux définis par logiciel basés sur OpenFlow. Ces tables visent à acheminer tout paquet vers la bonne destination. Les listes noires peuvent elles aussi être amenées à stocker des millions d'adresses IP de sources malveillantes. Certains règles admettent plus de 250000 instances correspondant à des domaines malveillants [7] et plus de 5000 nouvelles adresses IP malveillantes sont découvertes chaque jour. Le 8 août 2014, certains fournisseurs d'accès à Internet (FAI) ont connu des pannes de routeur provoquant des coupures de réseau dans le monde entier. Ce problème est apparu quand le nombre d'entrées dans une table de routage BGP globale a augmenté jusqu'à atteint plus de 512000 entrées, entraînant un problème dans les routeurs dotés d'une mémoire limitée à 512K entrées [8]. Si 512 000 semble considérable, le nombre d'entrées BGP actives aujourd'hui est de l'ordre de 900 000 et ne cesse d'augmenter avec le temps, comme le montre la figure A.1.

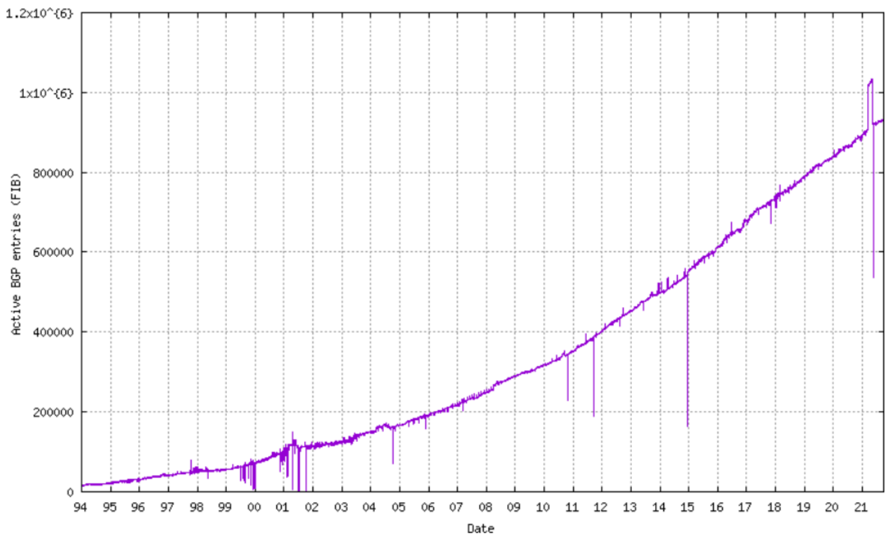


Figure A.1: Entrées BGP actives [9].

Ce nombre croissant d'entrées dans les dispositifs de mise en réseau peut créer des goulots d'étranglement et affecter les performances et la qualité de service (QoS). En outre, avec un nombre plus élevé d'entrées, la probabilité d'erreurs ou de mauvaise configuration est plus élevée. Selon [10], 20% de toutes les défaillances d'un réseau peuvent être attribuées à des mises à jour mal planifiées et mal configurées dans le réseau. Par conséquent, la gestion de ces grands ensembles de règles et du flux de données est complexe. Alors que l'automatisation et l'intelligence artificielle sont déployées dans de nombreuses applications de nos jours, de nombreux systèmes dépendent encore de l'administrateur pour le déploiement et la configuration du réseau, ce qui ajoute des délais supplémentaires et des sources d'erreurs et d'interruptions dans le réseau.

La réduction de la taille des ensembles de règles est un problème difficile pour les dispositifs de mise en réseau et les solutions de classification des paquets qui leur sont associées. De plus, le placement de toute entrée de règle dans le réseau en fonction de sa priorité est essentiel pour garantir que chaque flux inoffensif puisse atteindre la destination correcte tandis que les flux malveillants sont bloqués. Dans cette thèse, notre objectif principal sera d'aborder ces deux problèmes.

A.2 Problématique

Avec la croissance continue du nombre d'entrées dans les tables stockant les règles, c'est un grand défi de gérer les problèmes tels que le masquage des règles, la redondance des règles, les incohérences, etc. Une alternative à la résolution directe de ces problèmes est de tenter de les atténuer en s'appuyant sur une approche par compression afin de réduire la taille des ensembles de règles. CIDR n'est pas efficace pour traiter les ensembles de règles actuels, surtout lorsque le nombre d'exceptions est élevé. Nous avons donc besoin d'une nouvelle notation capable de gérer un grand nombre de règles. Cependant, il n'est pas évident de dériver un ensemble de règles sémantiquement équivalent à l'ensemble initial tout en économisant de l'espace. Par conséquent, le premier objectif de cette thèse est de minimiser les ensembles de règles en concevant une notation succincte pour compresser de grands ensembles d'adresses IP. Cette notation doit être généralisée pour fonctionner avec n'importe quel type d'intervalle, d'IP ou de port. De plus, nous montrons que la traduction entre la représentation CIDR et la nouvelle notation est rapide et linéaire en la taille des entrées.

Dans certains cas, la taille des routeurs est trop limitée pour contenir toutes les règles, même après compression. Dans d'autres cas, une application exige que les règles soient décomposées et distribuées sur le réseau au lieu d'être stockées au même endroit. Avec la grande taille des réseaux actuels et leurs topologies complexes, la distribution des règles est un défi. Par exemple, les paquets peuvent traverser plusieurs chemins et doivent être revérifiés avec les mêmes règles sur chaque chemin. Dans d'autres scénarios, un routeur appartenant à plusieurs chemins doit maintenir des règles spécifiques pour chacun d'eux, ce qui rend difficile la gestion de la mémoire du routeur qui est partagée entre les chemins.
La plupart des techniques de décomposition des règles utilisent des règles

supplémentaires dans les routeurs pour maintenir la même sémantique. Par exemple, considérons un routeur avec toutes les règles installées dans ce routeur. Nous avons deux scénarios: le paquet correspond à une règle, ou aucune correspondance n'est trouvée. Dans le premier scénario, si nous devons décomposer le même ensembre de règles sur plusieurs routeurs, le même paquet doit correspondre à la même règle dans un routeur puis traverser tous les routeurs suivants sans aucune correspondance. En revanche, dans le second cas, le paquet doit toujours correspondre à la règle par défaut dans tous les routeurs.

Les techniques de décomposition existantes reposent principalement sur la réplication des règles et génèrent de nouvelles règles pour préserver la sémantique. Cependant, l'ajout de règles supplémentaires augmente la charge des routeurs et affecte les performances du processus de filtrage et de classification des paquets. Pour surmonter ce problème, nous devons trouver un moyen de minimiser le nombre de nouvelles règles générées. En outre, la complexité des topologies de réseau rend la distribution des règles compliquée. En particulier, les noeuds admettant plusieurs flots entrant, les capacités des routeurs, la puissance de traitement, etc. doivent être pris en compte dans toute stratégie de distribution. Par conséquent, le deuxième objectif de cette thèse concerne principalement la décomposition et la distribution d'un ensemble de règles dans un réseau SDN tout en préservant la sémantique générale des ensembles de règles et en respectant la contrainte de capacité de chaque routeur.

A.3 Notre contribution

La classification des paquets peut être réalisée à l'aide de solutions logicielles ou matérielles. Les dispositifs qui utilisent des mémoires TCAM sont standard pour la classification des paquets en raison de leur temps de consultation plus rapide. Cependant, ces dispositifs ont un espace limité et nécessitent des quantités importantes d'énergie. D'autre part, les applications logicielles telles que celles fondées sur un arbre de décision sont plus faciles à mettre en oeuvre mais elles demeurent plus lentes que les TCAMs. En outre, pour diminuer le temps de classification, le nombre de règles dans les tables est contraint. Cela permet de réduire la consommation d'énergie et de rendre la recherche dans une structure de données plus efficace, ce qui se traduit par un meilleur processus de classification.
Cette thèse apporte deux contributions à la gestion des règles. La première est une méthode efficace pour réduire le nombre d'entrées dans les tables en utilisant une nouvelle technique d'encodage d'intervalle qui étend la notation standard CIDR. La deuxième contribution fournit des algorithmes pour distribuer les règles sur un réseau SDN tout en réduisant le surcharge dans les routeurs.

A.3.1 Compression des règles de filtrage avec Double Mask

Dans cette partie, nous concevons une représentation simple des règles de filtrage qui permet d'obtenir des tables de règles plus compactes, plus faciles à gérer tout en gardant leur sémantique inchangée. La construction des règles est obtenue avec des algorithmes raisonnablement efficaces également. Cette représentation s'applique aux adresses IP et aux intervalles de ports pour atténuer le problème d'expansion des intervalles. Pour cela, nous exprimons les champs du filtre de paquets avec ce qu'on appelle des *Double Mask* [116], où un premier masque est utilisé comme préfixe d'inclusion et le second comme préfixe d'exclusion. Cette représentation ajoute de la flexibilité et de l'efficacité dans le déploiement des politiques de sécurité puisque les règles générées sont plus faciles à gérer. La représentation *Double Mask* simplifie les configurations puisque nous pouvons accepter et exclure des IP au sein d'une même règle. Une règle *Double Mask* peut être considérée comme une extension d'une règle de préfixe standard avec des exceptions. Elle est souvent plus intuitive que les autres représentations et permet donc d'éviter les erreurs dans les opérations de gestion du réseau. Notre travail est basé sur une approche logicielle, et repose uniquement sur des règles acceptées, contrairement à [46; 117; 118] qui utilisent des approches matérielles. Notre notation peut réduire considérablement le nombre d'entrées dans les tables de routage. En comparaison, la représentation d'une intervalle de w bits peut nécessiter $2w-2$ préfixes [119]. Cette nouvelle notation a la même limite supérieure de $2w-4$ présentée dans d'autres articles [21; 41], mais dans certains cas, le nombre peut être réduit comme le montrent nos résultats expérimentaux.

A.3.2 Gestion des règles

La technique de compression ne peut pas toujours être efficace, soit parce que le taux de compression est faible, soit parce que la capacité mémoire des routeurs est faible. Par conséquent, pour faire face à ce problème, nous pouvons nous appuyer sur les réseaux définis par logiciel (SDN) pour distribuer les règles sur plusieurs routeurs. Dans SDN, les exigences de filtrage des applications critiques varient souvent en fonction des changements de flux et des politiques de sécurité. Le SDN résout ce problème grâce à une abstraction logicielle flexible, permettant de modifier et de mettre en oeuvre simultanément et facilement une politique de réseau sur des routeurs basés sur les flux. Cette approche de déploiement en un seul point constitue une caractéristique essentielle pour les opérations complexes de gestion de réseau.

Le nombre croissant d'attaques provenant de sources diverses augmente le nombre d'entrées dans les listes de contrôle d'accès (ACL). Pour éviter de dépendre des grandes et coûteuses capacités mémoire des routeurs de réseau, une approche complémentaire à la compression des règles serait de diviser les ACL en plus petites tables de routeur pour appliquer les politiques de contrôle d'accès. Cela ouvre la voie aux politiques de contrôle d'accès distribuées, qui ont fait l'objet de nombreuses études antérieures [86; 89; 90]. Cependant, la plupart des propositions donnent lieu à un taux élevé de réplication des règles [86; 89]. Certaines propositions nécessitent même de modifier l'en-tête du paquet [90] pour l'empêcher de correspondre à une deuxième règle de filtrage dans un routeur suivant.

Distribution des règles sur une topologie à chemin unique

Dans la première partie, nous introduisons un nouvel algorithme de distribution basé sur le LPM. Comme OBS [86], et Palette [89], nous nous appuyons sur les relations de dépendance des règles, mais nous générons moins de règles forward en exploitant la valeur du champ "action". Comme les travaux précédents, nous abordons le problème de la distribution d'un ensemble de règles parmi les routeurs du réseau pour répondre à une politique donnée. Cependant, dans notre approche, nous nous appuyons sur les propriétés de la stratégie LPM et nous exploitons les informations du champ d'action pour gérer les règles qui se chevauchent et éviter la réplication dans les tables des routeurs. De plus, contrairement à [90], notre solution ne nécessite pas de modification des paquets ou des règles.

Notre algorithme génère des règles de forward dans chaque routeur afin de préserver la sémantique des règles. En outre, l'algorithme est développé pour des topologies à chemin unique. Cependant, les topologies de réseau peuvent être plus complexes avec plusieurs chemins qui se chevauchent ou se croisent. Alors qu'une distribution sur un seul chemin peut être relativement facile à déployer, dans le cas de scénarios à chemins multiples, les règles dans les noeuds appartenant à plusieurs chemins doivent être soigneusement gérées pour les servir tous et limiter les redondances.

Distribution des règles sur un graphe

Dans la deuxième partie, nous présentons notre algorithme de distribution de règles pour les graphes séries-parallèles. Cet algorithme gère n'importe quelle stratégie de sélection des règles et peut fonctionner avec des algorithmes de décomposition existants comme OBS [86]. Ensuite, nous généralisons notre approche de distribution pour qu'elle fonctionne avec tous les graphes st-dags, et nous testons nos techniques sur un réseau réel appelé memorex. Nous introduisons ensuite une approche TWO-TIER pour la décomposition et la distribution des règles qui utilise deux types de routeurs et ne génère pas de règles supplémentaires. Cette approche est complémentaire à notre algorithme de distribution. Les deux peuvent être utilisés ensemble pour distribuer des règles sur n'importe quel st-dag sans règles supplémentaires. Ensuite, nous évaluons les performances de notre approche TWO-TIER et comparons le résultat avec les techniques existantes. Comme nous ne générons ni ne dupliquons de règles, le surcharge de notre approche est proche de zéro.

Stratégie de mise à jour des règles

Dans cette partie, nous présentons notre stratégie de mise à jour pour gérer les changements dans l'ensemble de règles initial, c'est-à-dire l'ajout, la suppression ou la modification d'une règle. Une modification peut être effectuée pour maintenir la topologie à jour, ajouter plus de fonctionnalités, améliorer les performances, ajouter plus de liens et de chemins au réseau, ou activer/bloquer un flux spécifique. La mise à jour d'un réseau peut se faire en ajoutant ou en supprimant des équipements matériels tels que des routeurs et des routeurs, ou en mettant à jour les données telles que les listes de contrôle d'accès ou les règles de transfert à l'intérieur des équipements existants.

Notre stratégie de mise à jour pour les graphes série-parallèle se concentre sur la partie du réseau affectée par la modification plutôt que sur le réseau entier. Enfin, nous évaluons la performance de notre technique de mise à jour en utilisant notre algorithme de distribution. Les résultats montrent une réduction jusqu'à 90 % du temps de mise à jour.

A.4 Limitations

Une règle Double Mask peut à la fois accepter et refuser des adresses avec mask1 et mask2 respectivement. Si aucune adresse n'est refusée, mask2 devient inutile, et le Double Mask se réduit à un simple masque. Dans ce cas, aucune compression ne peut être attendue avec notre algorithme. De plus, puisque la correspondance avec un masque simple est plus rapide qu'avec un Double Mask, il est plus simple de s'en tenir aux masques simples lorsque le taux de compression est proche de zéro, comme nous l'avons vu dans nos simulations.

Quant à notre algorithme de distribution, nous montrons que nous pouvons simplifier un graphe st-dag en un graphe série-parallèle en fusionnant certains sommets du graphe st-dag. Cependant, avec certaines topologies complexes où le degré des noeuds est important, la simplification nécessite de nombreuses phases de fusion. Dans ce cas, le graphe simplifié ne ressemblera pas au graphe initial, ce qui implique une qualité moindre pour les solutions de distribution dérivées du graphe simplifié.

En outre, aucune stratégie LPM satisfaisante pour des dimensions multiples n'a été développée. Cela est dû au chevauchement des champs dans les règles. Par exemple, si la règle R_1 avoir un champ source (resp. destination) plus spécifique et la règle R_2 un champ destination (resp. source) plus spécifique, pour un paquet qui correspond aux deux champs les plus spécifiques chaque règle est éligible et aucune décision ne peut être prise.

A.5 Travaux futurs

Cette thèse peut se prolonger par des améliorations dans la gestion des règles dans les réseaux SDN. En outre, nous pouvons identifier de multiples directions de recherche pour surmonter les limites des solutions proposées dans ce travail:

Premièrement, nous pouvons étendre notre solution de compression avec *Double Mask* aux intervalles de ports et aux adresses IPv6 puisqu'elles peuvent également être représentées comme des chaînes binaires. En modifiant la longueur de la chaîne en entrée, notre algorithme peut rapidement générer les masques correspondants. De plus, cette représentation peut être améliorée en calculant des *Double Masks* pour l'union des intervalles afin d'atteindre un niveau d'optimisation plus élevé dans les tables de routage. Notre technique actuelle calcule les masques pour chacune des intervalles indépendamment des autres. Cependant, dans certains cas, les *Double Masks* peuvent être étendus pour couvrir plusieurs intervalles simultanément comme discuté dans la sous-section 3.6. En outre, nous pouvons appliquer notre approche parallèlement à d'autres techniques pour supprimer les règles redondantes ou masquées. Cela permet de réduire le nombre des règles original et d'accélérer le calcul des *Double Masks* avec notre algorithme.

En ce qui concerne la partie "distribution" de cette thèse, nous pouvons automatiser tous les processus. Par exemple, nous pouvons nous appuyer sur des techniques d'apprentissage automatique pour déduire les modèles de communication et générer en conséquence les règles de filtrage nécessaires à la distribution par les algorithmes proposés. En outre, notre approche à deux niveaux peut être étendue à plusieurs routeurs FoS chacune pour une application spécifique. Par exemple, les applications de sécurité peuvent s'appuyer sur des routeurs avec des protocoles de sécurité renforcés, tandis que pour les applications de streaming, des routeurs FoS avec moins d'exigences de sécurité sont suffisants. Dans ce cas, les routeurs FiS décideront où transmettre un paquet correspondant en fonction de l'application.

Cette thèse peut être étendue en utilisant l'apprentissage par renforcement (profond) pour distribuer les règles. Ces techniques ont déjà été appliquées à de nombreux problèmes d'optimisation de réseau et pourraient être offrir de bonnes pistes pour le placement et la distribution des règles. Ainsi, en utilisant à la fois des techniques d'inférence pour extraire les politiques de filtrage requises à partir du comportement du réseau et un mécanisme d'apprentissage par renforcement pour appliquer ces politiques, la boucle d'une fonction de filtrage autonome sera complète.