LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

DISSERTATIONEN DER LMU

UB

61

CUONG NGOC TRAN

Conflict Detection in Software-Defined Networks

OLMS

Conflict Detection in Software-Defined Networks

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

eingereicht von
Cuong Ngoc Tran
am 11. April 2022

1. Gutachter: PD Dr. Vitalian Danciu
2. Gutachter: Prof. Dr. Wolfgang Hommel
Tag der mündlichen Prüfung: 13. Juni 2022

Cuong Ngoc Tran

Conflict Detection in Software-Defined Networks

Dissertationen der LMU München

Band 61

Conflict Detection in Software-Defined Networks

von
Cuong Ngoc Tran

OLMS

Universitätsbibliothek
Ludwig-Maximilians-Universität München

Eine Publikation in Zusammenarbeit zwischen dem **Georg Olms Verlag** und der **Universitätsbibliothek der LMU München**

Gefördert von der Ludwig-Maximilians-Universität München

Georg Olms Verlag AG
Hagentorwall 7
31134 Hildesheim
https://www.olms.de

Erstveröffentlichung 2023
Zugleich Dissertation der LMU München 2022

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet abrufbar über http://dnb.d-nb.de

Open-Access-Version dieser Publikation verfügbar unter:
http://nbn-resolving.de/urn:nbn:de:bvb:19-303449
https://doi.org/10.5282/edoc.30344

ISBN 978-3-487-16326-0

Contents

List of Figures

List of Tables

Acknowledgement

When your mind's mature,
and independent
in the world of science,
your beloveds, thanks!

It is a process of becoming mature and becoming an independent researcher, therefore the outcome of doctoral research is more than just publications and a dissertation book, it is the researcher himself and his way of thinking. I would like to express my sincere thanks to Vitalian – my Doktorvater – for sowing these seeds and patiently growing them with me throughout the years.

The wholehearted support of Professor Kranzlmüller and the LMU-MNM Team plays the key role to the success of my work. From specialized research problems to daily matters, just "shout out loud" in the team and there will absolutely be helpful echoes. My special thanks go to Professor, Annette, Karl, Nils, Miki, Tobias (Guggemos), Roger, Minh, Dang, Sophia, Matthias, Tobias (Fuchs), Jan, Pascal, Max, Markus, Amir, Daniel, Sergej, Michelle, Korbinian.

This interesting research would not come true without the DAAD scholarship. I am truly grateful for the warm support from DAAD, especially from the nice staffs at Section ST34 (Natalie Bursinski, Christian Strowa…).

The thoughtful feedback and kind encouragements of Professor Wolfgang Hommel sparked confidence in my research during the depressing covid-time. And finally, my disputation could take place with the help of Frau Ulrike Robeck and Professor Matthias Schubert.

In each stage of my life, I was lucky to have nice and attentive teachers, who helped shape my mind and propel me towards new horizons of knowledge. I am greatly indebted to them: thầy Hương, thầy Hoàng, thầy Lim, cô Hồng, thầy Nam, thầy Mẫn, thầy Vũ…

I wish to thank my colleagues at *genua* for their support in the last phase of my dissertation: Simon, Claas, Stefan-Lukas, Alexander, Carsten…The nice policies of the research team facilitated me a lot in this critical phase.

Oma Christine, Opa Reinhold and Tante Stefanie of my daughter My, they are also our family with whom I, my wife and our daughter can share everything.

My parents and my great-aunt "Bà Tự" – the plain farmers, and my aunt "Cô Nụ", always give me the best. No word can convey my gratitude to them.

My wonderful wife – Trang – und our lovely daughter – My – are always beside me through everything. We will continue our way together and forever! ❀

Cảm ơn ba má, bà Tự!
Cảm ơn vợ yêu, con gái yêu!
Cảm ơn ông bà, ba mẹ, anh chị em,
cô chú bác cậu dì, quý thầy cô, bè bạn,
những người luôn hỗ trợ, đồng hành, động viên Cường.
Thành công hôm nay xin sẻ chia cùng mọi người! ❀

Abstract

Traditional networks gain great success but are complex and designed with rigid functions due to the tight bundle of both hardware and software in each network device. The Software-Defined Networking (SDN) architecture facilitates the flexible deployment of these functions by detaching them from network devices to a logically centralized point, the so-called SDN controller, and maintaining a common communication interface between them. While promoting innovation for each side, this architecture induces yet a higher chance of conflicts. Control applications' intents are implemented via the SDN controller as network functions, such as routing, firewalling, load balancing. Different intents imposed concurrently on a network can result in conflicts, triggering unexpected network behaviour. The detection of conflicts in SDN, which is the prerequisite for their comprehensive handling, is the focus of this work.

Our analysis shows that the formal analytical approach is not sufficient in detecting conflicts due to diverse situations arising when operating an SDN. We opt for an experimental approach by determining a parameter space and a methodology to perform experiments through that space. We apply the proposed approach in various network test-beds built on the OpenFlow-based SDN. Each test-bed associates with a network topology, control applications are co-deployed there and traffic is generated between end-points, network behaviour is then observed and analysed, the results are arranged in a dataset of safe cases and potential conflict cases. To cope with the considerable size of the experiment space, we discuss techniques to produce "compact" subspaces and develop a framework to perform experiments in a highly automated manner. Eventually, we have accumulated output from more than 11,700 experiments in our dataset, which cover a number of situations occurring in SDN.

The investigation of the dataset yields a conflict taxonomy composed of various classes organised in three broad types: local, distributed and hidden conflicts. Each conflict class is featured by its unique pattern or property necessary for its identification. Our results augment the existing research in classifying local and distributed conflicts. Hidden conflicts caused by side-effects of control applications' behaviour are completely new. Unlike local and distributed conflicts, they cannot be discerned merely from rules in network devices but require additionally insight of the SDN control mechanics for their recognition.

We introduce the new concept of multi-property set, being a set with multiple properties, and a method to determine the relationship between sets of this kind thanks to a relationship combination operator that we name $\cdot r$ ("dot r"). These are valuable in that they enable the effective comparison of SDN rules without any limitations faced by existing solutions. Moreover, we exploit these findings in building rule graphs to examine possible paths along which packet flows traverse. With the capable means for comparing SDN rules and the rule graph, we present algorithms

to detect conflicts based on their patterns or properties, and develop a conflict detection prototype. The evaluation of the prototype justifies its reliability, indicating the correctness and the realizability of our proposed concepts and methodologies for classifying as well as for detecting conflicts.

Altogether, our work establishes a foundation for further conflict handling efforts in SDN, e. g., conflict resolution and avoidance. In addition, we point out challenges in connection with this work to be explored.

Zusammenfassung

Traditionelle Netze versprechen viel Erfolg, sind jedoch komplex und mit starren Funktionen konzipiert, da die Hardware und Software in jedem Netzgerät eng gebündelt sind. Die Software-basierte-Netz-Architektur (engl. Software-defined Networking, abk. SDN) erleichtert die flexible Bereitstellung von Netzfunktionen, indem sie von Netzgeräten zu einem logisch zentralisierten Punkt, dem sogenannten SDN-Controller, getrennt wird und eine allgemeine Kommunikationsschnittstelle zwischen ihnen aufrechterhält. Dieser Aufbau fördert zwar das Innovationspotential für jede Seite, führt aber auch zu einer höheren Wahrscheinlichkeit von Konflikten. Die Absichten der Steuerungsanwendungen werden über den SDN-Controller im Netz als Netzfunktionen implementiert, wie Routing, Firewalling, Load Balancing. Unterschiedliche Absichten, die dem Netz gleichzeitig auferlegt werden, können zu Konflikten führen und unerwartetes Netzverhalten verursachen. Der Umgang mit Konflikten ist daher unabdingbar. Im Fokus dieser Arbeit steht die Konflikterkennung im SDN, die Voraussetzung für deren umfassende Handhabung ist.

Unsere Analyse zeigt, dass der formalanalytische Ansatz zur Konflikterkennung aufgrund der vielfältigen Situationen, die beim Betrieb eines SDN auftreten, nicht ausreicht. Wir entscheiden uns für den experimentellen Ansatz, indem wir einen Parameterraum und eine Methodik zur Durchführung von Experimenten durch diesen Raum bestimmen. Wir wenden den vorgeschlagenen Ansatz in verschiedenen Netz-Testumgebungen an, die auf OpenFlow-SDN basieren. Jede Testumgebung ist einer Netztopologie zugeordnet, dort werden Steuerungsanwendungen gemeinsam bereitgestellt und Datenverkehr zwischen Endpunkten generiert, das Netzverhalten wird dann beobachtet und analysiert, was die Anordnung der Experimentergebnisse in einem Datensatz mit sicheren Fällen und potenziellen Konfliktfällen erleichtert. Um mit der beträchtlichen Größe des Experimentraums umzugehen, diskutieren wir Techniken, um "kompakte" Unterräume zu erzeugen und entwickeln einen Framework für die hochautomatisierte Durchführung von Experimenten. Schließlich haben wir in unserem Datensatz die Ergebnisse von mehr als 11.700 Experimenten kumuliert, die eine Reihe von Situationen abdecken, die in SDN auftreten.

Die Untersuchung des Datensatzes ermöglicht es uns, eine Konflikttaxonomie zu erstellen, die aus verschiedenen Klassen besteht, die in drei große Gruppen unterteilt sind: lokale, verteilte und versteckte Konflikte. Jede Konfliktklasse zeichnet sich durch ihr einzigartiges Muster oder ihre einzigartige Eigenschaft aus, die für ihre Identifizierung notwendig ist. Versteckte Konflikte, die durch Nebeneffekte des Verhaltens von Steuerungsanwendungen verursacht werden, sind völlig neu. Im Gegensatz zu lokalen und verteilten Konflikten sind sie nicht allein aus Regeln in den Netzgeräten zu erkennen, sondern erfordern zu ihrer Erkennung zusätzlich die Einsicht in die SDN-Steuerungsmechanik. Wir ergänzen die bestehende Forschung zur Klassifikation lokaler und verteilter Konflikte um ein umfassenderes Ergebnis.

Wir stellen das neue Konzept der Multi-Eigenschaft-Menge vor, die eine Menge mit mehreren Eigenschaften ist, und die Methode, um die Beziehung zwischen solchen Mengen mithilfe des Beziehungs-kombinationsoperators, den wir $\cdot r$ ("dot r") nennen, zu bestimmen. Diese sind wertvoll, da sie den effektiven Vergleich von SDN-Regeln ermöglichen, ohne dass Einschränkungen bei den anderen Lösungen entstehen. Ferner nutzen wir diese Erkenntnisse beim Erstellen des Regelgraphen, um mögliche Pfade zu untersuchen, die Paketströme durchlaufen. Mit den fähigen Mitteln zum Vergleichen von SDN-Regeln und dem Regelgraph präsentieren wir Algorithmen, um Konflikte anhand ihrer Muster oder Eigenschaften zu erkennen, und bauen einen Prototyp zur Konflikterkennung. Die Auswertung rechfertigt seine Verlässlichkeit, die die Korrektheit und Realisierbarkeit unserer vorgeschlagenen Konzepte und Methoden zur Klassifikation sowie zur Erkennung von Konflikten anzeigt.

Unsere Arbeit hat eine Grundlage für weitere Forschung zur Handhabung von Konflikten geschaffen, z.B. Konfliktlösung oder -vermeidung. Darüberhinaus weisen wir auf Herausforderungen im Zusammenhang mit dieser Arbeit hin, die es zu erforschen gilt.

1 Introduction

Existing traditional networks have grown rapidly from modest beginnings. Their interconnection making up the giant irreplaceable today's Internet demonstrates an apparent evidence of their success. In general, a traditional network device can be seen as being composed of the three planes: management, control and data [40]. The data plane consists of various ports for receiving and transmitting packets based on its forwarding table residing at the same plane, and switching fabrics to transfer packets from the input buffer to the output buffer. The control plane situating above that represents protocols used to populate forwarding tables in the data plane, e. g., the OSPF, BGP routing protocols in a router. The management plane contains software services useful for network administrators to monitor and configure control functionalities.

The vertical bundle of the control and data planes in each network device makes it capable of functioning autonomously. The control plane of one device communicates with its peers residing in neighboring devices to obtain the information necessary for the data plane to handle network traffic. In many cases, networking can be just as simple as the mere plug-and-play of devices and cables. This very architectural approach of distributed control plane is the main substrate that facilitates the fast growth and massive scalability of the Internet.

However, the mentioned vertical integration also makes the network complex and hard to manage. To make any change to the network, operators need to configure each individual network device separately with low-level and vendor-specific commands. The network management task, to some extent, can be compared to the "mastery of complexity" [69]. In addition, the architecture restricts network innovation to a small arena heavily dependent on a limited number of dominant network device manufacturers. New features for network devices could only be implemented by device vendors, the process often takes months or years from the feature request time until it is available in a new product or new firmware. This is frustrating from the view of network researchers and administrators, and also a big hindrance to network innovation. These limitations push the search for alternatives to traditional networking approach, one of those being Software-Defined Networking.

Experiencing more than two decades of evolution period since the presentation of *active networks* [105] in the early 2000s, the architectural approach that emphasizes the role of software in running networks, so-called Software-Defined Networking (SDN), has now got more traction in the networking community. In SDN, the control and data plane are decoupled from each other and are able to evolve relatively independently, which implicates a higher rate of innovation for each. Although there is no consensus on the definition of SDN, literature [40, 74, 42, 57, 97] commonly characterises its inherent features as follows:

- control functionality is detached from network devices, making them simple packet forwarding elements,
- forwarding decisions are flow-based in contrast with destination-based as in traditional networks,
- control logic is logically centralized in an (external) entity, the so-called SDN controller,
- the network is programmable through control applications running on top of the controller.

The concept of *flow* can be understood as a sequence of packets between a source and a destination, which share the same attributes. The mentioned attributes include fields in a protocol header, such as IP addresses, MAC addresses, status bits, they can also be the ingress port of packets arriving at a network device. A flow is expressed as a rule (or a flow entry) in the rule table of each network device on the path from the source to the destination, and can be represented differently in these devices.

Unanimous consent on SDN architecture has not been achieved though common views exist [38, 42, 91]. The simple form extracted from these commonalities, which underpins our work, is illustrated in Figure 1. The data plane contains SDN devices, also known as SDN switches, being simple forwarding elements without embedded control programs. Network intelligence is removed from them to a logically centralized controller. Packets are handled in these "dumb" switches based on their rule tables, whose rules consist of control information of different layers (e. g., layer 2–4) of the OSI model [47] and associated actions (e. g., forwarding matched packets to a specific port or to the controller, dropping, or modifying some header fields). Standardized interfaces are introduced between the controller and switches for their communication. The controller is a software stack that controls SDN devices. One important function of the controller is to provide the topology service for maintaining the consistent overview of the network. Any change in the network, such as introducing or removing devices, or disconnecting links, should be reflected immediately in the network overview at the controller. The controller changes configurations of network devices based on applications' demands. Network functions, e. g., MAC learning, routing, enforcement of QoS or security, are programmed by control applications, which logically reside

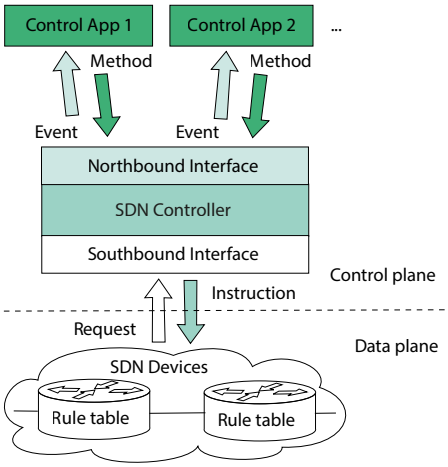


Figure 1: SDN architecture

above the controller. Control applications interact with the controller via its northbound interfaces to modify configurations of SDN devices.

The mechanics of SDN is represented by the arrows in Figure 1. A control application via its *method* asks the controller to deploy new network functions in the data plane, the controller in turn sends *instructions* to the data plane's devices to install new rules in the relevant devices. SDN devices can also send *requests* to the controller for instructions on handling certain traffic types. The controller then generates *events* for control applications, which respond via their *methods*. These *methods* are translated by the controller into *instructions* for the requested devices.

Any innovation brings both opportunities and challenges. SDN facilitates the operations and management of computer networks as its distinguished revolution to the conventional networking approach. By employing the flow concept for granular forwarding decision, an SDN device is able to theoretically function like any network device, such as switch, router, firewall, load balancer, or NAT device. Further, the capability of being programmed renders the implementation of new network features in SDN as simple as writing a program without having to rely much on network vendors, thus promoting network innovation. On the other side of the coin, the SDN architecture reveals appreciable issues. The centralized controller implies a single point of failure. The unavailability of the controller causes the loss of the communication channel between control applications and network devices, possibly making the data plane irresponsive to incoming traffic or behave in an unexpected manner. Another threat is that the compromise of the controller entails the full control of the whole data plane by attackers, which is much worse compared to the threat surface exposed by the distributed control plane in traditional networks. Besides, since control applications are independent from each other, each could ask the controller to realize its intent unbeknown to the others, which likely leads to conflicts in the data plane. These pros and cons of SDN are by all means not an exhaustive list. Nevertheless, they quote a subset of the imperative areas for researching. Our work lays focus on conflicts in the data plane induced by diverse control applications' intents.

1.1 Conflicts in SDN

While the SDN architecture shows many enhancements over that of traditional networks, it introduces a bigger challenge in terms of conflicts among control applications. Network functions are implemented by control applications atop the controller, which are translated by the controller into processing rules to be installed in the data plane's devices. Exemplary network functions include packet filtering, intrusion detection and mitigation, load balancing and so on. These functions are bound to network appliances with fixed positions in a traditional network. Implicitly, the intents associated with these functions are separated. Any issue, observed after traffic is processed by some function, can easily be ascribed to that function and the corresponding appliance. In contrast, SDN control applications can implement these

functions on arbitrary devices; the execution of a function can even be distributed across several devices. The trade-off for flexibility is thus the potential for network errors that cannot be identified easily.

Since each control application intends the best for the network in its own perspective, conflicts are likely when the SDN controller simply accepts requests and deploys them in the network. In practice, the controller may assume coordinating control applications. Still, conflict handling that forms the fundamentals for such a coordinator requires a scrutiny.

1.1.1 A demonstration of conflicts in SDN

Concrete experiments were conducted to illustrate conflicts in SDN [54]. A simple test-bed was constructed consisting of five switches and three end-points as sketched in Figure 2, on which two control applications including an End-point Load Balancer (EpLB) and a Path Load Balancer (PLB) were deployed. EpLB balanced UDP traffic over several targets by modifying the destination network address of relevant packets. PLB distributed traffic addressed to a target on several paths in a round-robin manner. Figure 3a) shows the expected network behaviour when EpLB was deployed alone on *Device 1* to balance traffic over *Target 1* and *Target 2*. The case of deploying only PLB on *Device 1* to balance traffic over three paths through *Devices 2, 3, 4* is illustrated in Figure 3b). Interestingly, the co-deployment of these two applications leads to unexpected results depending on their start order as depicted in Figures 3c and 3d.

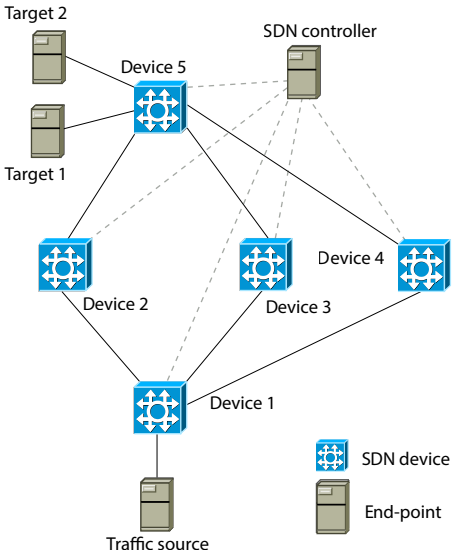


Figure 2: Network topology for experimenting conflicts [54]

Further experiments were carried out with varying order in executing the two load balancers, different targets and changing positions of EpLB in the topology. The outcome was alarming: 15 out of 16 cases were problematic [54]. Preliminary analysis indicates that conflicts depend not only on the overlapping address space but also on execution points of network functions in the topology, and on the deployment order of control applications.

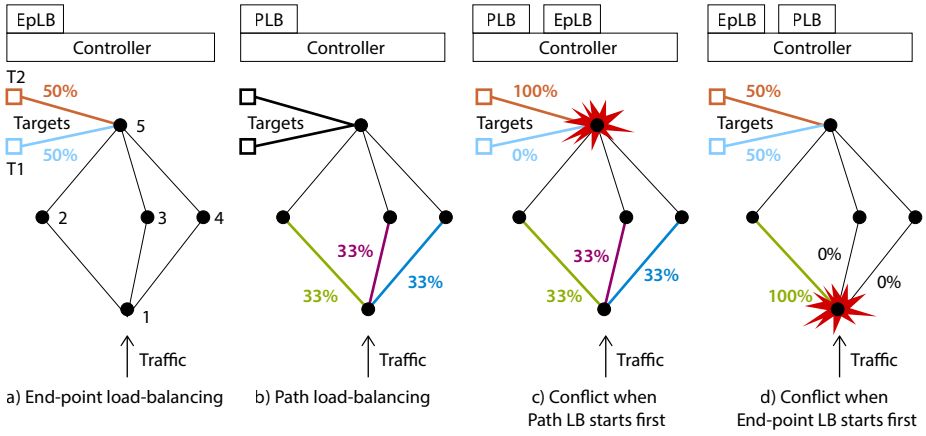


Figure 3: Proof-of-conflict: anomalous network behaviour identified as conflicts [108]

1.1.2 Conflict definition

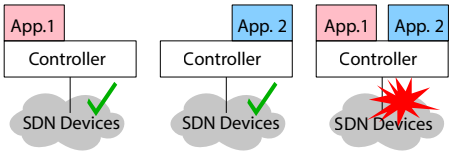


Figure 4: Definition of conflicts

The above conflict example exemplifies the notion of conflicts targeted in this work. We define a conflict as an anomaly caused by interference between different control applications: if the execution of two or more applications together provokes undesired network behaviour while running each of them alone is errorless, there is a conflict. This concept is illustrated in Figure 4.

Our definition of conflict delimits its scope from the concept of *bug*, which refers to a different kind of anomaly pertaining to an application and understood as the term commonly used in programming. We discuss in Chapter 2.2 more on bugs in SDN.

1.2 Research questions, scope and challenges

Conflicts render network behaviour erratic and unreliable, consequently, the control application's behaviour is deviated from its design. This issue is even worse in SDN than in traditional networks as mentioned above. It is vital to handle conflicts, which indicates either to avoid conflicts from occurring, or to resolve them once happening. These two approaches have as a prerequisite *conflict detection*, whose goal is to determine the existence of conflicts in a given set of control applications and among their rules deployed in the network.

A comprehensive study on detection of conflicts in SDN is the focus of this work.

1.2.1 Research questions

This work addresses the following research questions (RQ).

- RQ1: What is a suitable method to examine conflicts in SDN?
- RQ2: How can conflicts between control applications in SDN be classified based on their rules?
- RQ3: Given a set of rules from different control applications that are deployed or to be deployed in an SDN, how many conflicts exist therein according to a provided set of conflict classes? In case there exist conflicts, the following sub-questions need to be answered:
  - Conflict classification: to which conflict class does each of them belong?
  - Conflict localization: which rules are involved?

  The RQ3 and its sub-questions can be illustrated via the interfaces of a conflict detector depicted in Figure 5. The additional input including the control applications' information and the topology description is specified based on our analysis and implementation of the conflict detector in this work, which are elaborated in Chapter 5.

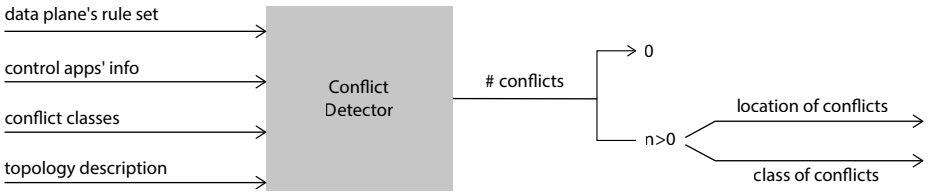


Figure 5: Input and output of the conflict detector

Our survey of research on conflicts in SDN (see Chapter 2) shows that they have been examined mostly based on specific use cases and analytical approach. Every now and then, new kinds of conflicts are discovered. This raises a search for a suitable method of studying conflicts in SDN tøachieve a comprehensive result, leading to RQ1. Once detected, a conflict needs to be classified based on its features to support handling efforts, e. g., conflict resolution, thus RQ2 is posed. The detection of conflicts, being our goal, corresponds to RQ3. Our work answers these RQ sequentially: we first determine a method to examine conflicts, then apply it to obtain a conflict taxonomy of various classes, therefrom we demonstrate how conflicts can be detected.

1.2.2 Scope of this work

We examine conflicts within an SDN controlled by a single controller, the rule sets in the network are installed by different control applications via that controller. We do

not delve into the cooperation between multiple SDN controllers governing different network segments.
We perform the detection of conflicts between control applications, not between management ones. The former concern themselves with the processing and forwarding of packets while the latter involve in monitoring, configuring network devices and their maintenance. Haleplidis et al. provides a detailed discussion on these two aspects of SDN [42].

Conflict detection can be conducted at different levels ranging from rules in the data plane (low level) to intents of control applications (high level). This work focuses on the study of conflicts at the rule level since its thorough examination is still missing, while the conflict exploration at higher levels needs to be founded on the low level ones.

1.2.3 Challenges

This work has encountered a lot of difficulties along its progress. First, we sought an appropriate dataset to experiment conflicts in SDN and to build a conflict detection program, yet there is no existing one. The dataset required for our study needs to portray the condition in which a conflict occurs, including the control applications used, their priority, their start order, the target SDN devices of each application, the tested network topology, the traffic generated in the network, the detected conflict classes and locations.

We find that the formal analytic approach of studying conflict in SDN is not sufficient and adopt the experimental approach supplemented by the analytic one (cf. Chapter 3.1). This entails a wide spectrum of control application types/characteristics to be inspected and experimented. Moreover, as we show in Chapter 3.2.1, the experiment space would explode in an unmanageable magnitude without a proper mechanism to reduce its size. The scarcity of existing applications implemented for a specific SDN platform adds more complexity to the chosen approach.

Conflict handling by itself is a complex classic problem from detection to orchestration, resolution or avoidance. Research on conflicts in SDN is limited at the time of this writing. This is conceivable due to technological limitations as there is no universal SDN realization and even no unanimous agreement on the SDN definition.

1.3 Results

In the course of addressing the research questions posed in Section 1.2.1, we have made contributions to the existing knowledge of conflicts in SDN. We found a suitable method to study conflicts in SDN by combining both analytic and experimental approaches. We establish a conflict taxonomy which is more comprehensive than the hitherto discovered in literature and also includes a completely new conflict branch, namely hidden conflicts. We propose effective methods to detect conflicts in a given

setup based on our conflict taxonomy, their correctness is justified via the realization and evaluation of a conflict detection prototype.

1.3.1 A suitable method to examine conflicts in SDN

Research on conflicts based on specific use cases and the analytical approach as conducted in related work (see Chapter 2.3.3) appears insufficient, new conflict types emerge every now and then. We compare the SDN model with that of distributed computing systems in principle and notice that the software engineering techniques applicable in the latter for verifying and evaluating a program cannot be applied in SDN though similarities between them exist. On the contrary, concrete use cases arising when operating SDN need to be considered as the base for researching conflicts. From our analysis of the SDN model, we infer an experimental parameter space and a general methodology to conduct experiments through that space. Details are presented in Chapter 3.2.

1.3.2 A framework for automating experiments in studying conflicts

We develop a framework that performs experiments and recognises unexpected network behaviour in a highly automated manner. Various experimental subspaces are explored by this framework, covering a large number of settings concerning network topologies, control applications, end-point combinations and other aspects. The results obtained from more than 11,700 experiments are collected in our published dataset[1]. The unexpected behaviour arising during the course of experimenting logged in the dataset is analysed to determine if new conflict types come up. More details are provided in Chapter 3.4.

1.3.3 Conflict classification

The analysis of anomalies in the experiments allows the establishment of a conflict taxonomy shown in Figure 6. Three broad categories include local conflicts, distributed conflicts and hidden conflicts, each is sub-categorised in various conflict classes based on their characteristics. Concrete experiments illustrating conflicts are provided in Chapter 3.6 and details on each conflict class with its discernible patterns or properties are presented in Chapter 4.

Local conflicts
These classes of conflicts arise due to the rule overlap local to a switch's rule table. An example is the *generalization* conflict class between two rules i and j which belong

1 https://github.com/mnm-team/sdn-conflicts

to two different control applications and exhibit the pattern: "the priority of rule i is smaller than the priority of rule j, the match space of rule i is a superset of that of rule j, their actions differ". Without the presence of rule j, all traffic matching rule i is handled according to the action of this rule; the presence of rule j renders a subset of traffic matching rule j, thus also matching rule i, to be handled according to the action of rule j, which does not fulfill rule i. As a result, two control applications installing these two rules cannot be satisfied at the same time, which constitutes a conflict.

Local conflicts have been examined extensively in literature, e. g., in the work of Pisharody [87] and Hamed et al. [43]. Our work complements theirs by a systematic analysis and produces a comprehensive result.

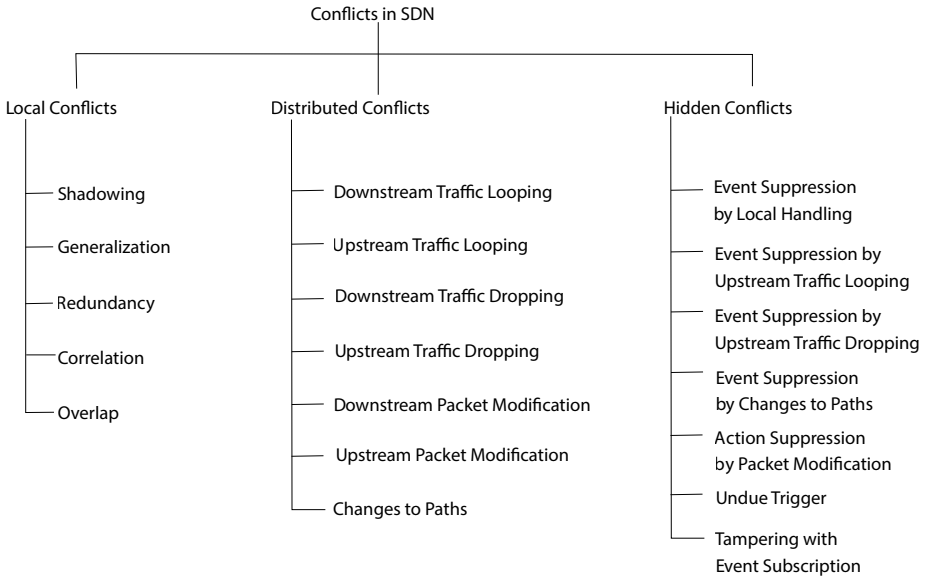


Figure 6: The taxonomy of conflicts in SDN

Distributed conflicts

As opposed to local conflicts, distributed conflicts are caused by rules from different control applications in different rule tables, possibly belonging to different devices across the network. For example, when deploying a control application, the traffic between two end-points A and B is delivered successfully; however, the co-deployment of that control application with another causes that traffic stuck in a loop.

Distributed conflicts have been researched in literature in different forms other than a conflict-related problem, e. g., network invariants [52], security enforcement [88, 95], bugs [50], therefore, the existing results are not suitable for conflict handling efforts. We introduce an effective method to detect distributed conflicts and to facilitate their further handling.

Hidden conflicts

Throughout the experiments, we have discovered a completely new conflict type, which we name *hidden conflicts*. Unlike local and distributed conflicts, hidden conflicts cannot be discerned by analysing alone rules in the data plane but need the insight of the control mechanics from the control plane for their detection.

We give an example of a hidden conflict between two control applications, say App 1 and App 2. App 2, when deployed alone, reacts to the notifications of the traffic flows 1, 2, 3 and 4 by installing the four rules 1, 2, 3, 4 respectively in device 1, this is depicted in the upper box of Figure 7. App 1 in its isolated execution installs rule 1234, which matches also the traffic flows 2, 3, and 4. Consider the case in which rules from App 2 have higher priority and different actions compared to rule 1234 from App 1. The matching scope of rule 1 and rule 1234 is illustrated in Figure 8. In their co-deployment, App 1 installs rule 1 in device 1 upon the arrival of flow 1, App 2 installs rule 1234 also in this device subsequently. When flow 2 comes to device 1, rule 1 does not match it but rule 1234 does, therefore, device 1 does not generate the notification of flow 2 to the control plane. As a result, App 2 does not receive the notification of flow 2 and does not install rule 2 as it did in the isolated execution, its intent is thus not fulfilled, which constitutes a conflict as shown in the lower box of Figure 7. This example is reproduced from experiment 6 in Chapter 3.6.3.

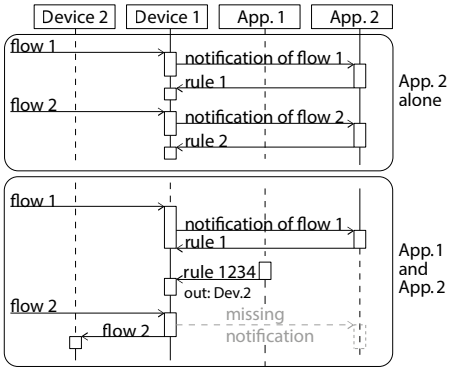


Figure 7: Interactions of an application in isolation and when conflicting with another [25]. For clarity, the controller intermediary has been omitted.

Rule 1 and rule 1234 expose a local conflict belonging to the *generalization* conflict class (see Section *Local conflicts* above), whose effect according to this local conflict class is that the broader rule (rule 1234) defers to the more specific one (rule 1) if the incoming traffic matches both. In the above example, however, the presence of the broader rule deprives the notifications required by a control application to function correctly. The analysis of rules suggests an effect differing from the actual consequence on the control application, which requires knowledge of the control plane's operation for its recognition. In other words, a conflict, as a main effect derived from the rules' relationship in the data plane, can also have its side-effect. We refer to conflicts arising from side-effects as hidden conflicts.

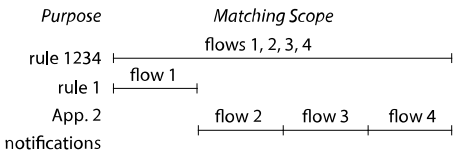


Figure 8: Scope of the rules issued by the applications [25]

To exhaustively examine hidden conflicts, we analyse the operational model of SDN based on its primitives and determine possible disruptions of the control plane's mechanics. Thereby, we are able to establish the causes and effects of different kinds of hidden conflicts and classify them accordingly.

1.3.4 Multi-property set and the relationship combination operator $\cdot r$

SDN rules need to be compared to detect conflicts and attribute each to a conflict class. Specifically, their priority, match fields and actions are collated. The comparison between the priority and actions of SDN rules is intuitive while their match fields' comparison is not that simple. The match fields can contain multiple properties, e. g., source IP address, destination IP address, source TCP port, destination TCP port; they can be expressed differently from rule to rule; their comparison results corresponding to their relationship can be disjoint, equal, subset, superset or intersecting. Due to the rigidity of the existing solutions that hinders their application in comparing general match fields (see Chapter 2.3.4), we introduce the new concept of *multi-property set* and the relationship combination operator $\cdot r$.

A multi-property set is created from the intersection of different single-property sets. For instance, consider a multi-property set of flowers in a garden that have five petals, red color and are scentless, this set contains the flowers with three properties: number of petals, color and scent. It can be constructed by intersecting the three single-property sets: the first set of five-petal flowers, the second set of red-color flowers and the third one of scentless flowers. The match fields of an SDN rule correspond to a multi-property set.

The relationship combination operator $\cdot r$ formulates the relationship between the match fields of two SDN rules based on the relationship of each single-property set associated with each field in the match. The resulting relationship combined by the $\cdot r$ operator is either disjoint, equal, subset, superset, or intersecting.

These concepts are general and can be applicable for any kinds of multi-property sets beyond the scope of our work, e. g., in determining the relationship between the conditions of different Access-Control List (ACL) rules or *IPTables*[2] rules, which is necessary for identifying conflicts or misconfiguration in a given rule set. Their details and how we apply them in comparing SDN rules are presented in Chapter 5.

1.3.5 The algorithms to detect conflicts based on *matchmap, actmap* and *rule graph*

Match fields and actions can be specified distinctly between SDN rules, this complicates their comparison to detect conflicts. For example, rule i of a routing application

2 https://linux.die.net/man/8/iptables

describes only the destination IP address in its match and the output port in its action, while rule j of a load balancing application has the destination IP address and the destination TCP port in its match, its action specifies how it modifies the destination IP address and the output port. We need to put the rule specification on the same scale, i. e., the same format, to be able to apply the concept of multi-property set and the relationship combination operator $\cdot r$ for their automatic comparison. The *matchmap* and *actmap* concepts are introduced for this purpose (see Chapter 5.2). We present additionally the *rule graph* (Chapter 5.4) to examine possible paths along which traffic traverses, this enables the identification of conflicts concerning rules in multiple devices. Our algorithms to detect conflicts based on these means (Chapters 5.5, 5.6 and 5.7) do not suffer from the rigid constraints that those similar in literature do (cf. Chapter 2.3.4).

1.3.6 Conflict detection prototype

Having established the methods to detect conflicts, we implement a conflict detection prototype (Chapter 6.1) based on OpenFlow SDN as a proof-of-concept for this work. The evaluation of the prototype (Chapter 6.2) justifies its soundness and completeness in detecting conflicts, which indicates the realizability, the applicability and the usefulness of our introduced concepts.

1.3.7 List of publications

Throughout the course of our work, we have published the following papers and articles, four of them contribute directly to this dissertation and two are related.

Publications contributing to the dissertation

- CUONG NGOC TRAN and VITALIAN DANCIU: *On Conflict Handling in Software-Defined Networks*. In Proceedings of the 2018 International Conference on Advanced Computing and Applications, pages 50–57, Ho Chi Minh City, Vietnam, 2018. CPS, https://doi.org/10.1109/ACOMP.2018.00016 [110]

  Summary: We demonstrate that the study of conflicts in SDN based on the analytical approach is not sufficient but requires in addition the experimental approach. We present a parameter space for experiments, analyse the challenges of the chosen approach and a methodology to conduct experiments through the space.

  Own contribution: We adopt the same approach to study conflicts in this dissertation, i. e., we analyse the limitations of the analytical approach (cf. Chapter 3.1) and opt for the experimental approach (cf. Chapter 3.2). The results of this paper are extended in the subsequent publication [108].

Other contributors: Vitalian Danciu proposed the SDN operational model and compared it with the model of distributed computing systems, which revealed the obstacles in applying the analytical approach for studying conflicts in SDN. A part of the parameter space was derived from this SDN model.

- CUONG NGOC TRAN and VITALIAN DANCIU: *A General Approach to Conflict Detection in Software-Defined Networks*. SN Computer Science, 1(1):9, Jul 2019, https://doi.org/10.1007/s42979-019-0009-9 [108]

  Summary: This journal article is an extension of the above paper [110]. We improve our analysis of the SDN operational model and introduce a more comprehensive parameter space together with a methodology to iterate through it. Concrete experiments are conducted to demonstrate the feasibility and soundness of the proposed approach.

  Own contribution: The results from this article are fundamentally taken over in this dissertation, including the experimental approach, the parameter space, the methodology for performing experiments (cf. Chapters 3.1 and 3.2), and a part of the conducted experiments (cf. Chapter 3.6). Compared to the article, these aspects are addressed in a more comprehensive manner in the dissertation, e. g., the size of the parameter space is better illustrated in Table 3.2 in Chapter 3.2.1 and more concrete experiments are provided in Chapter 3.6.3.

  Other contributors: In addition to his contributions in the above paper [110], Vitalian Danciu also took charge of refining the content of this article.

- CUONG NGOC TRAN and VITALIAN DANCIU: *Hidden Conflicts in Software-Defined Networks*. In Proceedings of the 2019 International Conference on Advanced Computing and Applications (ACOMP), pages 127–134, Nha Trang, Vietnam, 2019. IEEE, https://doi.org/10.1109/ACOMP.2019.00027 [109]

  Summary: We present hidden conflicts occurring due to the side-effects of rules deployed in the data plane. We analyse the SDN interaction primitives to determine possible causes of hidden conflicts, then perform experiments that support our analysis. In an effort to predict hidden conflicts, we sketch the conflict prediction prototype employing the speculative provocation method.

  Own contribution: Hidden conflicts are a completely new conflict branch in the conflict taxonomy. We adopt the results from this paper as part of this dissertation, which encompasses i) the introduction of hidden conflicts (cf. Section 1.3.3), ii) their analysis and classification (cf. Chapter 4.3) and iii) some experiments illustrating hidden conflicts (cf. Chapter 3.6). This paper is extended in the sub-

sequent publication [25]. The results in the dissertation are more comprehensive compared to those from this paper, e. g., two more hidden conflict classes are presented, namely *event suppression by upstream traffic looping* and *event suppression by upstream traffic dropping* (cf. Chapter 4.3.3).

Other contributors: Vitalian Danciu performed the analysis of the SDN interaction primitives to derive the causes of hidden conflicts and compared this new conflict type to the existing ones.

- VITALIAN DANCIU and CUONG NGOC TRAN: *Side-Effects Causing Hidden Conflicts in Software-Defined Networks*. SN Computer Science, 1(1):278, Aug 2020, https://doi.org/10.1007/s42979-020-00282-0 [25]

  Summary: This article extends the above paper [109] in that we provide more details of the hidden conflict prediction prototype, discuss the characteristics of hidden conflicts, the limitation of the centralized prediction and suggest the integration of the hidden conflict prediction function in control applications.

  Own contribution: Besides the results from the above paper [109], the outcomes from this article are inherited in this dissertation to argue the choice of methods for detecting hidden conflicts (cf. Chapter 5.7). Because of the limitations of the prediction approach, we opt for detecting hidden conflicts based on the input from control applications in this dissertation, instead of predicting hidden conflicts as mentioned in this article.

  Other contributors: In addition to his contributions mentioned in the above paper [109], Vitalian Danciu also pointed out some limitations of the hidden conflict prediction approach and refined the content of this article.

Publications beyond the scope of the dissertation

- CUONG NGOC TRAN and VITALIAN DANCIU: *Privacy-preserving Multicast to Explicit Agnostic Destinations*. In Proceedings of the Eighth International Conference on Advanced Communications and Computation (INFOCOMP 2018), pages 60–65, Barcelona, Spain, 2018. IARIA XPS Press [111].

  Summary: This paper presents MEADcast, a sender-centric multicast protocol that is able to maintain the privacy of the participants in a multicast session and supports gradual deployment with provable increasing efficiency proportional with the number of MEADcast-enabled routers. The recipients are agnostic of each other, thus preserving privacy, and is not required to support the protocol. The sender performs the discovery of MEADcast-enabled routers and transitions from unicast transmission to MEADcast transmission in the presence of MEADcast-

enabled routers. If there is no MEAD-cast enabled router, the sender transmits messages to recipients using the normal unicast scheme.

- VITALIAN DANCIU and CUONG NGOC TRAN: *MEADcast: Explicit Multicast with Privacy Aspects*. International Journal on Advances in Security, 12(1&2):13–28, Aug 2019, https://www.iariajourn-als.org/security/tocv12n12.html [24].

  Summary: This journal article extends the above paper [111]. We provide in addition the detailed design of the MEADcast protocol, illustrate its mechanics through examples, and the required APIs to implement the protocol in practice.

1.4 Methodology and dissertation's structure

The methodology to tackle the conflict detection problem is presented in Figure 9 partitioned into segments representing the dissertation's structure. Specifically, we employ the following measures.

1. Conducting literature review for relevant research (Chapter 2).
   To date, a handful can be listed, e. g., [71, 43, 86], they range from conflicts in distributed systems, in traditional networks and in SDN. Research on conflicts in SDN is limited and requires further investigation.
   Result: existing conflict handling solutions, conflict classes in general and in SDN.

2. Approaches and experiments (Chapter 3).

   - Choosing an appropriate approach to study conflicts in SDN.

     – Analysing the analytical approach. This formal approach turns out to be inadequate in exploring conflicts in SDN and needs to be augmented by the experimental approach.
     – Determining an experiment space with different configurations regarding control applications, network topologies, combinations of end-points.
     – Conceiving a methodology to perform experiments through the experiment space.

     Result: the choice of the experimental approach, a parameter space and a methodology to carry out experiments through that space.

   - Conducting experiments.

     – Selecting an application suite for experiments.
     – Building SDN test-beds.
     – Performing experiments throughout the parameter space using the proposed methodology. Note that, conflicts can only occur in the presence of at least

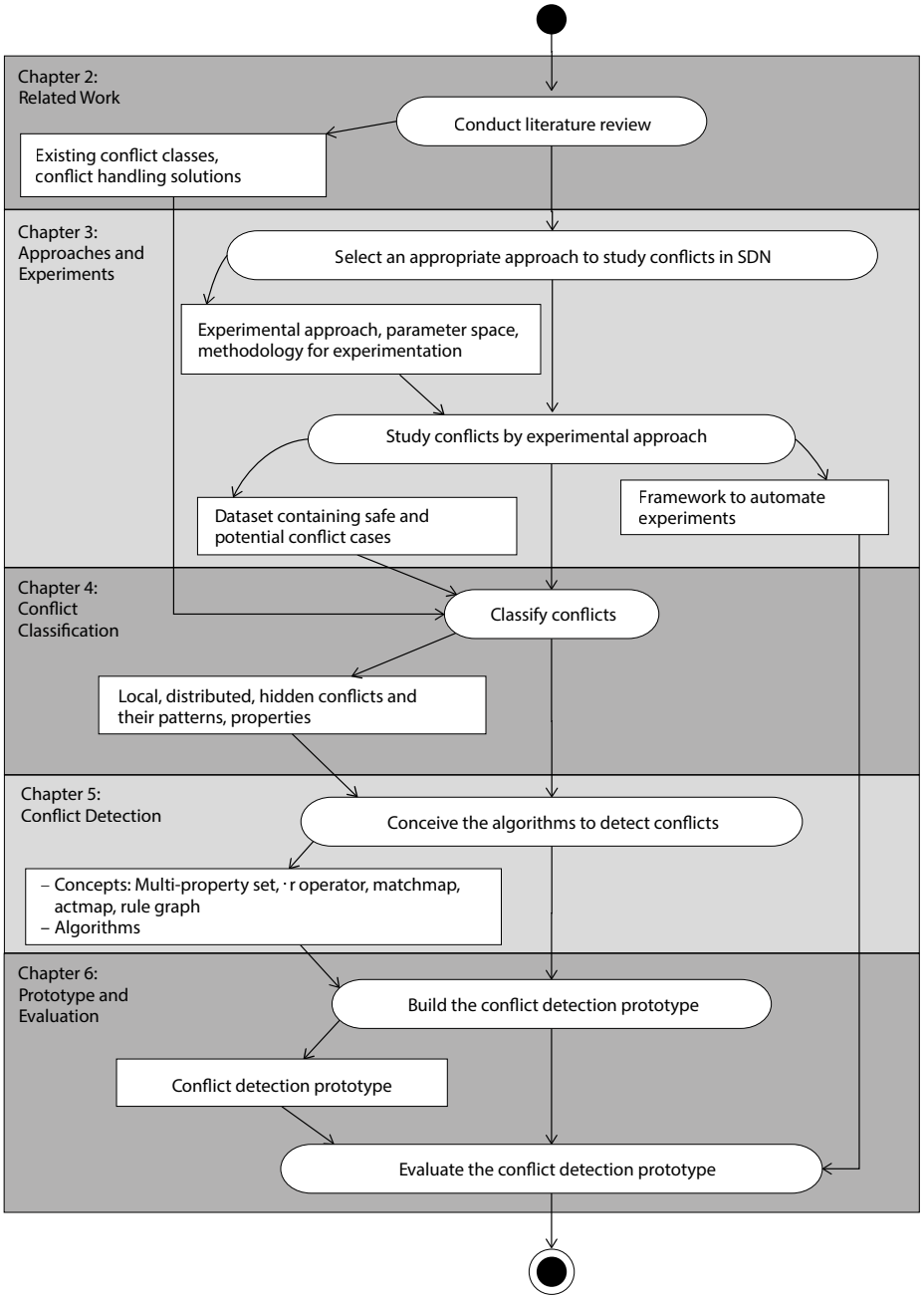


Figure 9: Methodology

two control applications according to our definition (see Definition 1.1.2), applications' bugs (cf. Chapter 2.2) are excluded from the experiment process. In each experiment, the expected network behaviour is compared with the observed network behaviour after deploying the applications together to reason about conflicts. Since the parameter space is immense, a mechanism to reduce its size and to automate the experiments is presented.
– Recording the characteristics of each conflict case, they are reflected by the configuration of the experiment, the way that conflicts happen and the data plane's rule tables. Conflicts can be reasoned therefrom, e. g., a contradicting rule set in an SDN device can be attributed to the configuration of the applications or to the transport protocols used by the end-points.

Result:

– A framework to automate experiments through the parameter space according to the methodology from the previous step.
– A dataset in which each data entry corresponding to an experiment setting is marked as *safe* or as *potential conflict*. The data entry for the case of *potential conflict* logs the condition/environment of conflicts, how and where they occur.

3. Analysing the dataset and classifying conflicts (Chapter 4).
Based on their characteristics, conflicts are divided in different groups, forming a conflict taxonomy. The existing conflict classes and the classification strategies from literature in the first step are also taken into account.
Result: a conflict taxonomy including local, distributed and hidden conflicts, and their patterns, properties.

4. Detecting conflicts (Chapter 5).
Conceiving the methods and algorithms to detect conflicts based on the results from step 3.
Result:

- The introduction of the new concepts that are employed in detecting conflicts: *multi-property set, relationship combination operator* $\cdot r$*, matchmap, actmap* and *rule graph*.
- Algorithms to detect conflicts.

5. Proof of concepts and evaluation (Chapter 6).

- Building a conflict detection prototype based on the concepts and algorithms proposed in step 4.
Result: a conflict detection prototype.

- Evaluating the prototype. The prototype is evaluated based on designed cases and randomly checked cases. In the designed cases, a set of rules are deployed in the chosen network topologies with known conflict types and location, the detection results by the prototype are verified against these designed conflicts. In the randomly checked cases, the prototype is integrated into the framework for automating experiments from step 2, then a massive number of experiments are conducted automatically by the framework with various settings concerning network topologies, control applications, end-point combinations…We select randomly samples and control the detection results of those.

We conclude our work and share our view on future prospects in Chapter 7.

2 Related Work

We delineate salient work through the SDN history that shapes today's SDN architecture, which is a premise for our work. Next, we distinguish the concepts of *bug* and *conflict* to underline our focus. The related research on conflicts in distributed systems management and in traditional networks is then presented. Finally, we analyze the relevant work on conflicts in SDN, mainly from which our work benefits.

2.1 A sketch of the SDN history

The effort for flexible network programmability has been made since years ago. Tennenhouse and Wetherall introduced the concept of *Active Networking* in 1996 [105] as a radical approach to network control. A programming interface (or network API) was envisioned to expose resources (e. g., processing, storage, and packet queues) on individual network nodes, and to support the construction of custom functionality to apply to a subset of packets passing through a node. The idea was further developed in two programming models: capsule [114] and programmable router/switch model [96, 7]. Active Networking did not see widespread deployment possibly due to the lack of a clear path to development.

The next movement witnessed the separation of the control and data planes to ease the network management tasks focusing narrowly on routing and configuration management, such as debugging configuration problems and predicting or controlling routing behaviour. This trend catalyzed the introduction of i) an open interface between the control and data planes, such as the ForCES (Forwarding and Control Element Separation) interface [117] and ii) logically centralized control of the network, proposed in the Routing Control Platform (RCP) [14], SoftRouter [58] architectures and the Path Computation Element (PCE) [30] protocol. Although there was some progress from the industry prototypes and standardization efforts, this trend still did not receive widespread adoption. Dominant equipment vendors had little incentive to promote open data-plane APIs which could facilitate new entrants into the marketplace. Besides, the need to rely on existing routing protocols to control the data plane limits the applications that programmable controllers could support.

The advent of OpenFlow [70] made an important milestone. With experience and lessons learnt from the previous endeavours and the well-timed availability of open APIs from switch chipset vendors like Broadcom, OpenFlow balanced the tension between research and pragmatism in that it facilitates more functions and it is immediately deployable. Still, it needs to be enriched with more applications and use cases. The term "Software Defined Networking" was coined by Kate Greene in 2009 in an article about the OpenFlow project at Stanford University[1].

1 https://www.technologyreview.com/technology/tr10-software-defined-networking/

OpenFlow employs a protocol-dependent mechanism in the data plane to handle packets, in other words, it targets fixed-function switches that recognize a predetermined set of header fields and that process packets using a small set of predefined action. This eventually restricts its innovation capability, e. g., there is no way to support a new protocol containing header fields that are not yet defined by the associated OpenFlow specification. This explains why the numbers of actions and match fields keep increasing with the OpenFlow's evolution: OpenFlow version 1.5 has 44 match fields and 19 actions, while the numbers in OpenFlow version 1.0 were 12 and 10, respectively. Perceiving this limitation, Protocol Oblivious Forwarding (POF) [98, 61] and Programming Protocol-Independent Packet Processors (P4) [12] were introduced with the goal of decoupling network protocols from packet forwarding and make the data plane reconfigurable, protocol-independent, programmable, and future-proof. Specifically, POF uses the *{offset, length}* tuple to form a search key to match a packet and also to encode actions that need to handle protocols e. g., pushing a VLAN header, changing destination IP address; *offset* indicates the skipped bits from the current cursor within the packet and *length* indicates the number of bits that should be included in the key starting from the *offset* position. P4 extends OpenFlow with a parser and table configuration component that is capable of compiling a configuration expressed in a P4 program and (re)configuring an SDN device to instruct it how packets are to be processed, the parser is a finite state machine that walks an incoming byte-stream and extracts headers based on the programmed parsing graph. As both approaches have attracted intensive interest from the Open Networking Foundation (ONF), they are considered in ONF's project on protocol-independent forwarding (PIF)[2]. At the time of this writing, both P4 and POF are still in their early stages of adoption with restricted availability for networking research community. OpenFlow has been widely developed and well supported while also possessing fundamental characteristics similar to P4 and POF for studying conflicts, therefore we choose OpenFlow for our experiments and prototype development. More about the SDN history and influencing projects are presented in [31, 48].

The management aspect involving monitoring, configuring and maintaining network devices tends to be less explored in literature related to SDN. The SDN Research Group (SDNRG) from Internet Research Task Force (IRTF) presents a comprehensive SDN layer architecture taking into account both the control and management concerns [42]. This architecture is compatible with the view of SDN from other standard organizations including International Telecommunication Union (ITU) and Open Networking Foundation (ONF). Well-known management-related work includes SNMP [11], NETCONF [28] with YANG data modeling language [8], OpenFlow Management and Configuration Protocol (OF-Config) [77], Open vSwitch

2 https://www.opennetworking.org/news-and-events/protocol-independent-forwarding/

Database (OVSDB) [83]. Interesting studies in this regard are contributed by the research group of Hommel and Steinke [99, 100, 101]. Our work focuses on conflicts within the control area of SDN involving mainly processing and forwarding packets, and excludes the consideration on the management facet.

2.2 Conflicts and bugs

It is necessary to differentiate between a *bug* and a *conflict*. Early conflict resolution research [71] defined the term *conflict* based on undesired behaviour due to an overlap of policy concerns, i. e., an issue that can only occur in the presence of two or more concerns.

We notice that these two classes of *anomalies* in network behaviour need to be distinguished. If the behaviour can be proven to pertain to a certain application, it is called a *bug*, understood as the term commonly used in programming. Bugs in SDN have been studied in literature on various dimensions, e. g., applying model checking with symbolic execution to debug OpenFlow control applications [15], avoiding programming faults and race conditions for control applications [37].

Some work debugs the data plane agnostic to control applications [45, 44], i. e., it does not care whether the anomalous network behaviour is caused by a control application or by the co-deployment of multiple applications. While the goal of these pieces of work diverts from ours, we could benefit from their findings to pinpoint the problem in the data plane, e. g., to show the traversing path of a packet with any header modifications between two hosts who currently cannot talk to each other, or to detect a loop via packets passing the same switch twice.

As mentioned in Chapter 1, our work focuses on conflicts, which are anomalies caused by the interference between applications.

2.3 State-of-the-art

Conflicts by itself is not a new topic, it has been researched extensively in different fields. We can observe the connection of conflicts in distributed system management, in traditional networks to those in SDN.

2.3.1 Policy conflicts in distributed system management

Distributed system management helps ensure efficient use of resources and provide reliable services to users. Due to its importance, a set of standards (though originally dedicated for Open Systems Interconnection (OSI) management [67]) have been developed to cope with the complexity in managing heterogeneous systems, including: (i) Fault Management to detect, isolate and correct faults, (ii) Accounting Management to record resource usage information and enable service providers to charge for their services, (iii) Configuration Management to control installation and operation

of services within a distributed system, (iv) Performance Management to improve the services provided to users, e. g., better throughput, response time or reliability, or to reduce operating cost, (v) Security Management to support the application of security policies and mechanisms of the system, e. g., access control, encryption facilities.

Policies are usually used as a means of management. They can be represented differently in a hierarchical fashion. At the high level, policy defines the goals of an organization and possibly also the resources to achieve the goals. Low-level policies refined from high-level ones guide the behaviour of resources or infrastructure elements. Danciu sketched a policy refinement hierarchy and its attributes as shown in Figure 10, in which the top level strategic policies are refined into the middle level functional policies, which in turn are refined to the lowest level operational policies [22]. Down along the hierarchy, the business aspects and abstraction level of the policies decrease while their technical aspects and detail level grow. This view of policies was also presented in the work of Wies [115] and Koch [55]. Examples of high-level policy languages are KAoS [112], Virtualization Assurance Language for Isolation and Deployment (VALID) [10] or the Intent Northbound Interfaces introduced by ONF [46], some noticeable low-level policy languages include Ponder [21], REI [49], Policy Description Language (PDL) [62] and eXtensible Access Control Markup Language (XACML) [29].

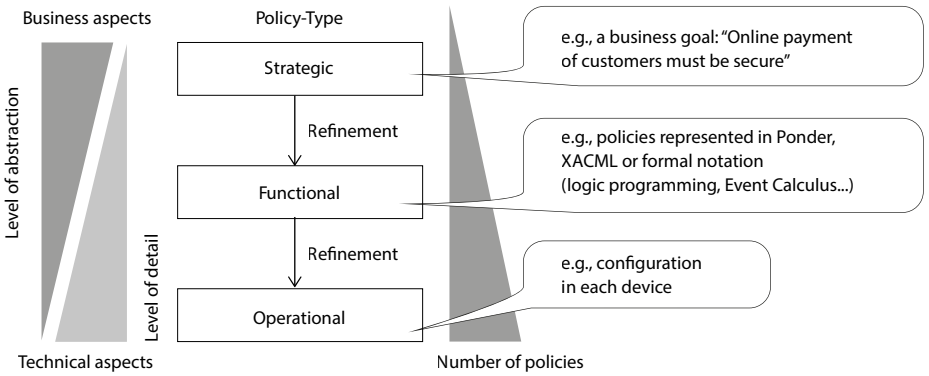


Figure 10: Policy refinement hierarchy [22]

Policy conflicts can occur when different policies have different goals applying to the same or overlapping resources or services in a system. Conflict handling is thus vital in distributed system management.

In order to automate the tasks of policy-based management and conflict handling, policy modelling is necessary. Different working groups presented various views and methods in modelling policies depending on their concerning scope of applications/services. As a result, multiple policy languages are introduced. Pérez et al. [82] provide a comprehensive list of requirements for a certain policy language and a policy framework/architecture. The important features of a policy language include: being unambiguous and verifiable, clear and well-defined semantics, flexibility and ex-

tensibility, interoperability with other languages, support of multiple bindings, being amenable to combining. The policy framework/architecture that transfers, stores and enforces the policies written in a policy language needs to be: (i) well-defined, independent of the particular implementation, (ii) flexible to allow addition of new device types, (iii) capable of interoperating with other framework/architecture, (iv) capable of detecting conflicts between policies, (v) capable of mapping high-level policies to low-level ones, (vi) scalable under an increased system load.

We select the remarkable studies on policy-based management based on the above criteria and discuss them in the following.

Ponder and Event Calculus

Moffet and Sloman [71] described management action policies representing by these attributes: modality (i. e., imperatival and authority), subject, target object, goal which can be refined into actions, and constraint. An exemplary policy is: "Members of Payroll (subject) are permitted (modality) to Read (goal) Payroll Master files (object), from terminals in the Payroll office, between 9AM and 5PM, Monday to Friday (constraint)". A conflict is potential if there exists between two policies the overlap of subjects and/or objects. A conflict taxonomy and the mechanism to resolve conflicts of modality were given. Lupu and Sloman [65, 66] continued this line of research by proposing a method to detect and resolve modality conflicts in an offline manner based on roles and policy precedence. The Ponder policy language [21] proposed by this working group targets security policies for access control implementation also complies to the management action policy template with the mentioned attributes.

Bandara et al. [3] presented a method for transforming specifications of event-driven policies and system behaviour into a formal notation based on Event Calculus [56]. The formal notation in conjunction with abductive reasoning techniques are applied in policy refinement and in performing a priori analysis of policy specifications to identify modality conflicts and some application specific conflicts, such as conflicts of interest and conflicts of duty. This work also facilitates the translation of policy languages like Ponder [21] into the proposed formal notation to ease the conflict handling procedure.

PDL and Logic Programming

PDL [62] is a declarative policy language that follows the "*event causes action if condition*" rule paradigm of active databases. PDL has clearly defined semantics, an architecture for enforcing its policies has also been specified. PDL does not support access control policy and the mechanism to grouping policies. Chomicki et al. introduced a framework employing logic programming for detecting action conflicts between policies represented in PDL and for resolving them [17, 18]. Conflicts are captured as violations of action constraints. The semantics of rules and conflict detection and resolution are defined axiomatically using disjunctive logic programs generated automatically from PDL specifications. Conflicts are resolved by either suppressing the events that could cause conflicts or by overriding the conflicting action.

Rei

The Me-Centric project at HP Labs demanded a language that was able to express management, conversation and behaviour policies, which led to the introduction of the Rei policy language [49]. Rei is described in first order logic format for easy translation from/to RDF, DAML+OIL and OWL. Policy makers create polices specified in Rei for domains consisting of resources. The polices contain actions that can be performed on the resources, properties of users/agents that are allowed to use the resources and the policy objects found by the conditions and the associated actions. Policy object includes rights, obligations and dispensations. Rei allows policies to be combined based on role and group. Rei policies are handled by a *Rei policy engine* residing in a *policy server*, which interacts with a *domain server* to retrieve policies of different domains and inserts them to the policy engine. The domain server keeps the domain memberships of users built from contextual information sources and the location of the policies. The Rei policy engine is queried when an agent requests a certain action on a resource, the engine then checks if the requester has the right by checking its policy objects or if it has been delegated the right. If the agent has the right, its requested action can be enforced.

Rei supports conflict resolution using meta-policy specifications, which are policies about how policies are interpreted and how they can be resolved statically. Conflicts occur if two or more policies have the same action, on the same target but their modalities (right/prohibition, obligation/dispensation) are different. Conflicts are resolved statically by specifying priorities and precedence relations for the policies.

PCIM

IETF Policy Core Information Model (PCIM) [73], extending Common Information Model (CIM) schema[3] from Distributed Management Task Force (DMTF), introduces the classes for defining policy objects that enable application developers, network administrators, and policy administrators to represent policies of different types. In this work, a policy-controlled network can be modelled as a state machine, policies are used to control which state a policy-controlled device should be in or is allowed to be in at a given time. A policy contains a set of policy rules, each consists of a set of conditions and a set of actions. If the set of conditions of a policy rule evaluates to true, the set of actions of this rule is executed. Policy rules can be prioritized, e. g., to express an overall policy that has a general case with a few specific exceptions. In the policy hierarchy, PCIM targets the specifications of lower-level, vendor- and device-independent policies.

The main components of the PCIM model include: Policy Management Tool (PMT), Policy Repository (PR), Policy Decision Point (PDP) and Policy Enforcement Point (PEP).

3 https://www.dmtf.org/standards/cim/cim_schema_v24

- PMT is the tool for the administrator to create, edit, delete policies and monitor the status of the policy-managed environment. Notifications as a result of the system monitoring process are sent to the administrator via PMT.
- PR is the storage for PMT to store the policies manipulated by the administrator and for PDP to put it into use. IETF suggests Lightweight Directory Access Protocol (LDAP) [102] as the access protocol to PR, though other implementations are also available.
- PDP is the point where policy decisions are made. It has the global view of the network area under control. PDP interprets policies from PR into a format that can be understood by PEP and sends them to PEP.
- PEP is the point where the policy decisions are actually enforced. It is a component running on a network/system node executing policies received from PDP. PEP can notify PDP upon encountering unknown situations.

Interactions between PCIM components and possible interfaces are depicted in Figure 11.

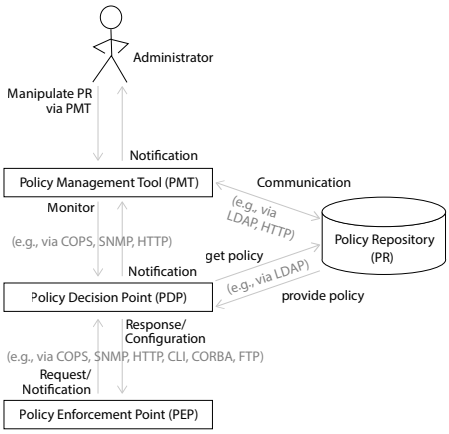


Figure 11: PCIM components and their interactions

As an information model, PCIM is kept generic in nature and must be mapped into a particular data model (e. g., CIM schema, LDAP schema, Management Information Base (MIB) [68]) for implementation. The communication interfaces between its components are therefore flexible up to the data model, Figure 11 gives some examples of those in addition to LDAP, such as, Common Open Policy Service (COPS) [27], Hypertext Transfer Protocol (HTTP) [35], Simple Network Management Protocol (SNMP) [16], Common Object Request Broker Architecture (CORBA)[4], File Transfer Protocol (FTP) [89]. An exemplary application of PCIM for policy-based admission control is presented in a framework in [118].

PCIM framework configures resources via role. The administrator assigns each resource to one or more roles and specifies policies for these roles. The use of role helps maintain the consistent enforcement of policies across the whole network under control and ease the process in which PDP selects policies from PR already grouped on a role basis to apply on a particular set of resources. Moreover, the problem of conflicts between policy rules can be simplified by defining the compatibility between roles, so that, policy rules associated with incompatible roles will not be deployed

4 https://www.omg.org/spec/CORBA/

on the same resource. Still, conflict handling is left to the specific implementation of PCIM.

We could map the SDN architecture to the PCIM model to some extent, a possibility is shown in Figure 12. The (logically) centralized SDN controller corresponds to PDP, it has the global view of the network and translates policies/intents of control programs, which associate with PR, into rules in the rule tables of SDN devices, which represent PEP. It is, however, not a perfect mapping so that the results, including conflict handling, from existing PCIM implementation could be directly applicable in SDN. The underlying assumption of the PCIM model is that policies are stored in a centralized PR and are carefully defined in the predefined format/syntax by an administrator to avoid possible conflicts (though conflicts are inevitable). In SDN, policies are distributed among different control applications without having to comply to any predefined format/syntax, these applications can also come from third parties, making it hard for the administrator to understand their objectives.

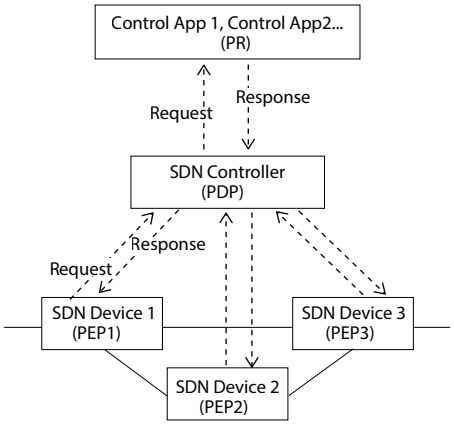


Figure 12: A possible mapping of the SDN architecture to the PCIM model

XACML

The OASIS consortium[5] proposes XACML language [29] specialized in access control policies. XACML is an XML-based language[6], now also supports JavaScript Object Notation (JSON) [13] representation. XACML follows the PDP/PEP approach similar to the PCIM model. It is by itself a policy language, and also a language for describing the access control decision request/response between PEP and PDP. A XACML policy specifies access control rules composed of *subject-target-action-condition*. Policies are written in XACML by an administrator and made available for PDP. Upon receiving a query to access a certain resource, PEP will form a XACML request with relevant attributes (e. g., the requester's attributes, the resource in question, the action) and send it to PDP; PDP then looks at the request and the policies that apply to it, and issues a XACML response to PEP to indicate if access to the resource in question should be granted. XACML reconciles policy conflicts through a collection of combining algorithms, for example, *deny-override, permit-override, first-applicable*.

5 https://www.oasis-open.org/
6 https://www.w3.org/XML/

Summary
These working groups modelled policies and policy conflicts in distributed system management and proposed methods to detect and resolve conflicts. Our work benefits from their concept of conflicts: conflicts can only occur in case of the existence of interference/overlap between two or more different entities, which are SDN control applications in our case. Each control application deploys its rules governing network behaviour. A rule can be seen as a low-level, operational policy that comprises of a subject, an object and a set of actions in the simple form: *subject* performs *action* on *object*, for example, switch S (as subject) forwards out of port O (as action) packet P (as object). The modality (imperatival and authority) and the constraints of policies are not considered at this low-level policy, i. e., the rule, in our work.

Intuitively, two control applications are at odds if their rules contradict each other. In the SDN context, a conflict between two rules occurs if they have the same subjects (the same switch), same or overlapping objects (same packet/flow or overlapping set of packets/flows) and different actions. Figure 13 illustrates a conflict case for two policies: S (subject) performs $A1$ (action 1) on $O1$ (object 1) and S performs $A2$ on $O2$ while $A1 \neq A2$. In fact, we have also showed that if the actions of these two rules are the same or overlapped, conflict is still potential (see Hidden Conflicts in Chapter 4.3). In our work, we model additionally the priority of policies in an explicit manner while it is often implicitly modelled in distributed system management's policies, e. g., it is commonly admitted that the more specific policy has higher precedence over the more general one.

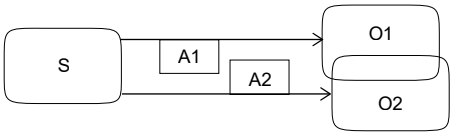


Figure 13: Conflict is potential between two policies with overlapping subjects S (Switch) and objects O (packets/flows)

In the full picture guided by the extensive research in this field, our focus appears on the low-level operational policies. The techniques to detect conflicts specific to SDN at this low level that we have applied are different than the ones devised in this broad field. We inherit therefore chiefly the concept of conflicts and benefit in positioning our work in this general research landscape as well as how it values and could involve.

2.3.2 Policy conflicts in traditional networks

The approaches for policy-based management in distributed systems can generally be applied in networking area, forming the base of Policy-Based Network Management (PBNM) [103]. Some examples include a framework for policy-based admission control focusing on Resource Reservation Protocol (RSVP) [118], RSVP policy control using XACML [106]. The impact of these pieces of work to ours was concluded in the previous Section (2.3.1).

Amongst the research on policy conflicts in traditional networks, the study conducted by the research group of Hamed and Al-Shaer [94, 93, 43] on conflicts in network security policies is most relevant to our work. They examined policy conflicts that occurred due to rule misconfiguration in filtering-based network security, such as in firewalls and IPSec gateways. Similar to policy rules commonly used in a firewall, a policy rule examined in their work is composed of:

```
<order> <protocol> <src_ip> <src_port> <dst_ip> <dst_port> <action>
```

The order of the rule determines its position relative to other filtering rules, which indicates its priority in the rule table; a rule has higher priority than its subsequent rules in the rule table and lower priority compared to its preceding rules. The "5-tuple filter" refers to the common layer 3 and 4 header fields of the OSI [47]: *protocol type, source IP address, source port, destination IP address, destination port*. The action field can be either the *bypass* (the traffic is transmitted unsecured), *discard* (the traffic is dropped) or *protect* (the traffic is securely transmitted, e. g., by employing IPSec) action. IP addresses can be a host or a network address, a wildcard (*) indicates an *any* value of the IP address or the port field.

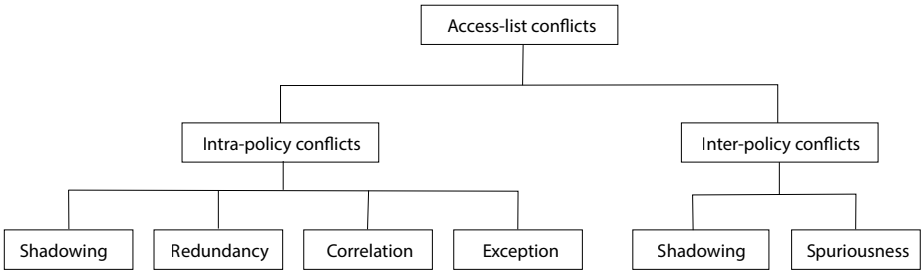


Figure 14: Access-list conflict classes (adapted from [43])

Conflicts between rules are classified into various types (see [43], Figure 3) depending on their interrelationship. The access-list conflicts among those (see Figure 14) also transpire in SDN, which are further divided into the broad types of *intra-policy conflicts* for conflicts occurring within a single device and *inter-policy conflicts* for those between different devices. Downwards in the hierarchy, intra-policy conflicts include the four classes: *shadowing, redundancy, correlation, exception* and inter-policy conflicts contain: *shadowing, spuriousness*.

- Intra-policy shadowing: a rule is shadowed when every packet that could match this rule is matched by some preceding rule having a different action. As a consequence, the shadowed rule will never get effective. Shadowing is considered a critical error as the shadowed rules become obsolete and never take effect, which might cause legitimate traffic to be blocked or illegitimate traffic to be permitted.

The recommended resolution to this type of conflict is to position the general rule after the more specific rule.

- Intra-policy correlation: two rules are correlated if there is some traffic that matches both rules. If they have different actions, then the action performed on the traffic depends on the ordering of the two correlated rules and a conflict occurs in this case. Correlation is considered a conflict warning since the correlated rules imply an action that is not explicitly controlled by the policy. To resolve this conflict, the correlation between rules should be detected and the administrator needs to choose the proper rule order complying with the policy requirements.
- Intra-policy exception (or intra-policy generalization): A rule is an exception of some subsequent rule if their actions are different, and the subsequent rule is more general (it could match all traffic that is matched by the preceding rule). Exception is in general not an error, however, it needs to be identified because the exception rules change the policy semantics, which may be undesirable. This type of conflict is considered a warning that should be highlighted to the administrator.
- Intra-policy redundancy: A rule is redundant if every packet that could match this rule is matched by some other rule having the same action. Redundancy is considered a problem as it increases the size of the rule table and the search time for packet filtering, while not contributing to the policy semantics. The redundant rules should be discovered and removed.
- Inter-policy shadowing: The interrelationship between two rules is similar to that of the intra-policy shadowing conflict but the rules locate in two different devices. An inter-policy shadowing conflict occurs if the rule in the upstream device blocks some traffic that is permitted by the rule in the downstream device. The authors did not mention methods to tackle this kind of conflict.
- Inter-policy spuriousness: In contrast to the inter-policy shadowing conflicts, inter-policy spuriousness conflicts occur if the rule in the upstream device permits some traffic that is blocked by some rule in the downstream device. It is considered a critical conflict because it allows unwanted traffic to flow across the network, increasing the network vulnerability to attacks, e. g., port scanning, denial of service. The authors did not specify mechanisms to handle this conflict type.

This research group also implemented a tool called "Security Policy Advisor" for automatic discovery of the above conflicts. Similar work focusing on generic standalone firewalls and security gateways was conducted by Ferraresi et al. [33]. Studies related to verification of network security policies, e. g., [26, 119, 63] examine the translation and comparison of security rules, e. g., ACL or IPTables[7] rules, some attempt real-time quality. Their proposed methods can be adapted for comparing SDN rules in detecting conflicts, especially when the real-time aspect is desired.

7 https://linux.die.net/man/8/iptables

Summary
In traditional networks, network security rules for packet filtering appear more complex than rules/low-level policies used by other control functions, e. g., routing, load balancing, traffic engineering. Therefore, the view that, policy conflict handling strategies in filtering-based network security can well represent the others, is reasonable. In SDN, a flow rule in the data plane expose the similar structure as the network security rule in that it is comprised of a priority value, various header fields and actions. Techniques and findings in handling conflicts in filtering-based network security can thus be applied in SDN to some extent. Our work takes over, for example, some conflict classes identified by the aforementioned studies. However, we notice that conflict handling in SDN is more arduous as the source of rules is not only from a single administrator or an administrative group, but also from different control applications. This means that studies of conflicts in SDN need to take into account both flow rules in the data plane, and control applications. In addition, the flexibility of SDN in supporting more match fields and action types (e. g., OpenFlow SDN version 1.5.1 [78] specifies 45 match fields and 19 action types) increases the complexity in conflict handling.

2.3.3 Conflicts in SDN

We provide a brief survey on prominent studies related to conflict detection in SDN, shown in Table 2.1. The criteria are selected based on our own concerns driven by the goal of this research, e. g., conflict classification, employed control applications, mechanisms for handling conflicts (including their detection), and the common interests in literature on this topic, e. g., conflict resolution, real-time conflict handling, invariant compliance. Invariants indicate the predefined network properties that must be complied with, e. g., availability of a path to a destination, absence of forwarding loops, enforcement of access control policies, isolation between virtual networks. We observe in the early experiments (in Chapter 1.1.1) that network topologies also influence how conflicts occur, thus this factor is considered in this survey.

FlowChecker [92] is a tool for identifying intra-switch misconfiguration within a single flow table (also rule table) or inter-switch inconsistencies across OpenFlow infrastructure. It encodes the data plane's flow tables configuration in Binary Decision Diagrams and models the network behaviour in a single state machine, then using symbolic model checking and temporal logic to provide the verification interface. This interface can be applied in verifying the network consistency or validating the correctness of the configuration of the data plane against the deployed control application, or in debugging reachability and security problems. VeriFlow [52] is a framework that intercepts the communication channel between the SDN controller and network devices to check for network-wide invariant compliance when new network events emerge. These events comprise forwarding rules being inserted, modified or deleted. VeriFlow is agnostic of the control applications above the controller,

Name	Conflict type	Resolution	Topology awareness	Real-time	Invariants	Mechanism	Target apps
FlowChecker [92]	intra-switch, inter-switch	✗	✓	N/A	✓	Binary Decision Diagrams, symbolic model checking, temporal logic	common control apps
VeriFlow [52]	N/A	✗	✓	✓	✓	equivalence classes, forwarding graphs	app-agnostic, focusing on OpenFlow rules
FortNOX [88] FRESCO [95]	N/A	✓	✗	✓[1]	✗	alias reduced rules	OpenFlow-enabled security apps
HFT [32]	parent-child, inter-sibling	✓	✓	N/A	✗	Hierarchical Flow Tables, Network Information Base	security, traffic engineering
NetPlumber [51]	N/A	✗	✓	✓	✓	header space analysis, incremental check of network updates, plumbing graph	app-agnostic
Corybantic [72]	N/A	✓	✓	N/A	✗	control apps propose high-level objectives, central coordinator resolves conflicts	common control and management apps
Statesman [104]	N/A	✓	✓	✓[2]	✓	three views of network state and dependency model of state variables	management apps, hardware-related
Athens [2]	N/A	✓	✓	N/A	✗	control apps propose high-level objectives, central coordinator resolves conflicts by voting mechanisms	common control and management apps
Ravel [113]	N/A	✓	✓	✓[3]	✓	the whole controller and apps are represented in database languages, conflict handling is based on the view mechanisms of the database and a mediation protocol	common control apps
Brew [86]	redundancy, overlap, shadowing, generalization, correlation, imbrication	✓	✗	✓	✗	flow extraction, flow atomization, Patricia trie lookup, atomic flow comparison	security, focusing on OpenFlow rules
TCDR [20]	inter-transaction, intra-transaction	✓	✗	✓[4]	✗	isolating flow rules of network functions, their deployment is transaction-based, flow rules are compared to detect and resolve conflicts before being deployed	general management apps

[1] (tested up to 1000 flow rules)
[2] (tested up to 100K variables)
[3] (tested on mininet [59])
[4] (tested on mininet)

Table 2.1: A qualitative comparison of research related to conflict detection in SDN

rules passed by the controller to the network are its sole concern. FortNOX [88] is a software extension of the NOX OpenFlow controller [41], which provides role-based authorization and security constraint enforcement when deploying new applications. It is later developed into FRESCO [95], which shares the same scope of concern. The Hierarchical Flow Tables (HFT) framework [32] organizes SDN rules into trees, in which each tree node is independent in choosing the action to execute on a packet. The resolution of conflicts in different parts of the tree is performed by user-defined conflict-resolution operators located at each node of the tree. Its runtime is discussed for future optimization, we note the *real-time* column for HFT thus as "not available". The target application is derived from the policy tree and the actions on traffic supported by HFT, which include admission control, guaranteed minimum bandwidth and "don't care". NetPlumber [51] is a policy checking tool similar to VeriFlow. Its implementation is based on the analysis of the header space of a packet and a *plumbing graph* which captures all possible paths of flows in the network. Real-time capability is achieved by performing incremental check of network updates instead of recomputing all network transformation as its previous work [50]. Corybantic [72] supports modular composition of disparate controller modules (or applications) competing for resources while attempting to maximize the overall value of the controller's decisions. Each module makes proposals of change in the data plane, e. g., turning off a switch or moving a virtual machine; then each module evaluates every current proposal to calculate its gain/cost against that proposal in a common currency, e. g., $ (USD); after that the coordinator, based on the sum value from the previous steps for each proposal, picks the best proposal to deploy. Being at its early development phase, Corybantic leaves some challenging issues including how to make a good proposal for each control module and how to solve the primitive constrained multi-objective optimization problem. Statesman [104] implements a network-state management service that facilitates the independent operation of network management applications. Its design is inspired by version control system (like *git*[8]) which divides the network state into different views for conflict handling. Athens [2] is a revision of the aforementioned Corybantic [72] design combined with some features from Statesman's approach [104] while employing the voting mechanisms to resolve conflicts and to determine the resulting network state. The Ravel system [113] introduces an new paradigm based on SDN, namely database-defined network, it adds an abstract layer above the control plane by means of database, specifically SQL database. Control applications are then characterised by database entries. Conflict handling is accomplished thanks to the database's view mechanisms and a mediation protocol. The Brew framework [86] explores conflicts in SDN security applications, the rule actions are abstracted into *permit* and *deny*. Competing rules, e. g., "forward packet A out of port 1" and "forward packet A out of port 2" seem to be harmo-

8 https://git-scm.com/

nious in Brew, which might need to be addressed by research relying on this. The TCDR approach [20] regards the deployment of a network function as transaction, i. e., either all rules of that function take effect or none of them is enforced in case of conflicts. Its realization is based on FortNOX [88]. Conflicts are arranged into intra-conflict and inter-conflict, each with its own resolution strategy. The evaluation on the mininet emulator [59] shows a comparable performance as its original controller (Floodlight[9]) without the integrated TCDR components.

Summary
Our review of research related to conflicts in SDN communicates a common fact: most studies aim directly at conflict resolution with the trade-off of limiting the application domain of their solutions. The observation that new kinds of conflicts emerge from time to time in various studies advocates the judgement: the comprehensive treatment of conflicts in SDN is unattainable without learning their essence concerning the causes, the classification and the influenced factors. Therefore, we opt for a layered approach focusing first on conflict detection on a broad scale to obtain a more thorough understanding of interference between applications' intents, before taking the step towards further conflict handling (resolution, avoidance, application design).

Most research on conflicts in SDN so far does not concern the sources of rules applied or to be enforced in the data plane. In other words, conflicts arising in the data plane are purely examined between rules present there without a projection to their sources: the control applications. This leads to the omission of important conflict classes that we have discovered (cf. Chapter 4).

2.3.4 Analysis of remarkable research

Our work acquires the existing results regarding conflict classification and detection, while distilling the knowledge learnt from their limitations. We highlight the overlapping while contrasting our contributions in the following.

Conflict classification
The taxonomy of conflicts in network security policies presented by Hamed and Al-Shaer [94, 93, 43] is partly reclaimed in our work. As mentioned in Section 2.3.2 (also in Figure 14, they classify conflicts between access-list rules into *intra-policy* and *inter-policy* categories. The former one refers to conflicts between rules within a device and contains four classes, namely *shadowing, redundancy, correlation* and *exception*; the latter, caused by rules in different devices, consists of two classes: *shadowing* and *spuriousness*. We observe similar conflicts in SDN that can be triggered by local rules

9 https://github.com/floodlight/floodlight

of a single device or by rules distributed in multiple devices, which we name by the more intuitive terms: *local conflicts* and *distributed conflicts*, respectively.

- Local conflicts are organized into *shadowing, redundancy, correlation* and *generalization*, which correspond to the four intra-policy conflict classes. However, we notice another local conflict class with its distinctive pattern, named *overlap*. This kind of conflicts is induced by two rules having intersecting match space and the same actions. Interestingly, *overlap* class is also valid as a subclass of the intra-policy conflicts in network security policies, complementing the taxonomy introduced by Hamed and Al-Shaer.
- Distributed conflicts are divided in a more granular level specific to SDN and our conflict definition, the *upstream* and *downstream* directions of a traffic flow that influence the classification of inter-policy conflicts are applicable for distributed conflicts. The analysis of our experiments' results yields seven distributed conflict classes: *downstream traffic looping, upstream traffic looping, downstream traffic dropping, upstream traffic dropping, downstream packet modification, upstream packet modification* and *changes to paths*.

Pisharody examines conflicts in security policies in SDN [86] based on the existing outcome from Hamed's and Al-Shaer's research. Intra-policy conflicts are arranged into six classes: *shadowing, redundancy, correlation, generalization, overlap* and *imbrication*, in which *imbrication* is new compared to our local conflict classes. *Imbrication* conflicts are featured by rules whose match components specify header fields of different layers that cannot be compared directly. For instance, the match fields of one rule contain only IPv4 addresses, the other rule with match fields contain only MAC addresses. There may exist packets that matched by both rules although their match components expose no overlap at the first glance. Pisharody proposes to reconcile such rules by mapping layer 2 addresses to layer 3 addresses by means of an ARP table lookup, thereby an *imbrication* conflict between rules is transformed to a conflict, if any, belonging to one of the other five local conflict classes. In our work, all rules are represented uniformly before being compared (by employing the *matchmap* and *actmap* concepts, cf. Chapter 5.2), thus we do not consider *imbrication* as a local conflict class. Besides, we find it more suitable to classify a part of *redundancy* conflicts in his work as *overlap*, this is justified in Chapter 4.1.6. Distributed conflicts seem to be evasive in his study.

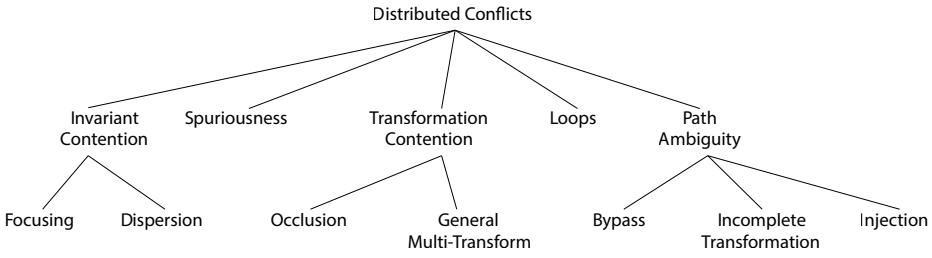


Figure 15: Taxonomy of distributed conflicts established by Reyes (reproduced from [90])

Reyes proposes a taxonomy of distributed conflicts in SDN, which is reproduced in Figure 15. We iterate through each conflict class with arguments regarding its adoption in our work.

- *Invariant contention*: invariants refer to certain criteria that a control application tracks to decide if its policies need to be enforced, e. g., the throughput threshold of some traffic. This conflict class is formulated due to the deviation of the network behaviour between the isolated and the concurrent deployment of control applications, caused by e. g., the traffic throughput at the monitoring point varying in two cases. In spite of the deviated network behaviour, we notice that the control applications' intents are still fulfilled in either case: their rules are enforced if necessary, i. e., only when the invariant threshold is exceeded, and these rules take effect as desired. With reference to our definition of conflicts (see Chapter 1.1.2), we do not incorporate this conflict class and its subclasses (*focusing* and *dispersion*) into our conflict taxonomy.
- *Spuriousness*: this conflict class emanates from the inter-policy conflict class *spuriousness* in Hamed's and Al-Shaer's work mentioned above. An instance of this class is observed when traffic allowed to flow in the network by an application is blocked by another in the downstream direction of that traffic. This conflict class corresponds to our distributed conflict class *policy suppression by downstream traffic dropping*.
- *Occlusion*: is not considered as a distributed conflict class in our work as we define distributed conflicts to occur between rules present on multiple rule tables in the data plane, while the definition of this class in Reyes' work reveals that rules of the impacted application may not be deployed at all. In fact, this class appears to be associated with our hidden conflict class *action suppression by packet modification*.
- *General multi-transform*: this class represents a special case of our distributed conflict class *policy suppression by downstream packet modification* as we observe that conflicts can arise even when traffic is transformed only once.
- *Loops*: this class corresponds to our distributed conflict classes *policy suppression by downstream/upstream traffic looping*.
- *Path ambiguity*: this class and its subclasses (*incomplete transformation, injection, bypass*) are associated with our distributed conflict class *policy suppression by changes to paths*. Conflicts of the subclasses *incomplete transformation* and *injection* appear to have the same cause: traffic is transformed in transit on the forward direction from the source to the destination, but fails to be re-transformed properly on the backward direction, leading to the communication failure between the involved end-points. We consider these situations to be special cases of the other subclass *bypass*, and group them in the same distributed conflict class.

Rule comparison
We need to compare SDN rules to detect conflicts. One of the objectives is to determine the relationship between the two sets associated with the match fields of two given rules, which can be either *disjoint, equal, subset, superset* or *intersecting*. As the match fields of SDN rules can contain multiple individual fields, e. g., *source IPv4 address, destination IPv4 address, IP protocol, source TCP port, destination TCP port*, the direct application of the existing set theory to derive their relationship is not simple, especially in case we wish to generalize the comparison of sets containing multiple fields and automating this task in a computer program. In this regard, the method proposed in Hamed's and Al-Shaer's work for comparing network security policies [93], reapplied in Pisharody's study for comparing SDN rules [86], is most relevant. Yet, it is rigid in that the match fields of the policy/rule must be specified in the format of the below five-element tuple so that the comparison method can be performed:

```
<protocol> <src_ip> <src_port> <dst_ip> <dst_port>
```

The arising rigidity is conceivable as their research focuses on security rules/policies only but not generic ones. With that limitation taken into consideration, we introduce a general method to derive the relationship between sets containing multiple fields, based on the new concept *multi-property set* and the relationship combination operator $\cdot r$ (see Chapter 5.1). The two new concepts *matchmap* and *actmap* are then presented (in Chapter 5.2) to apply this method for comparing SDN rules whose match fields do not need to be restricted to the five-element tuple as mentioned above.

Conflict detection
Reyes [90] extends our conflict detection prototype (cf. Chapter 6.1) for identifying conflicts. The detection results concerning *general multi-transform* conflicts in his work are reused in evaluating our prototype (cf. Chapter 6.2). Existing network debugging and verification tools such as VeriFlow [52], HSA and its Hassel library [50], NetPlumber [51], APKeep [119] can identify traffic looping and dropping, the graphs representing the traffic paths for verifying invariants in these tools (e. g., *forwarding graph* in Veriflow and APKeep, *propagation graph* in HSA and *plumbing graph* in Netplumber) are built in a similar manner as our *rule graph* for detecting distributed conflicts (cf. Chapter 5.4). However, due to the disparate goals, these graphs are not meant for detecting conflicts between control applications as ours and thus cannot be employed directly in our case. Our *rule graph* is constructed with richer features encoded in edge attributes to facilitate the reasoning about distributed conflicts. An interesting topic on optimising the performance of our prototype can take advantages of the strategies learnt from their work, e. g., by using the efficient *trie* data

structure [116] for searching rules, or partitioning the rule graph in multiple subgraphs to parallelize the reasoning of conflicts and aggregating the interim outcomes to obtain the overall results.

3 Approaches and Experiments

The detection of conflicts in SDN within the scope of our work exhibits similarities to the correctness verification of a distributed program in distributed computing systems. Such verification can be undertaken by using common methods in software engineering domain, e. g., model checking techniques. We attempt to employ the same approach by developing a simple SDN model to analyse conflicts. We notice, however, that it is insufficient. The study needs to be complemented by experiments to cover situations particular to the mechanics of SDN. Based on the model's analysis, we derive a parameter space and a general methodology to conduct experiments as well as to detect conflicts through that space.

We perform a massive number (over 11,700) of experiments by applying the proposed methodology over various parts of the parameter space, most of them are carried out automatically with the help of our framework. The framework automates experiments in the chosen parameter subspaces, identifies potential conflicts and creates a dataset for further examination of conflicts. The proposed methodology is clarified via concrete experiments with a set of control applications covering different criteria. We present nine experiments selected from both manual and automatic ones, which supports the formulation of conflict classes and the decisions in realizing the conflict detection prototype in subsequent chapters. The extraction of conflict patterns or properties demands human intervention, we present a strategy to reduce manual effort while still being able to maintain the high level of effectiveness for this process.

3.1 Considering the analytical approach

We consider the analytical approach in detecting conflicts by modelling SDN and perform analysis thereon. The comparison with distributed computing systems reveals yet that particular characteristics of SDN hinder this approach. We come to the conclusion of employing the experimental approach as a result.

3.1.1 SDN model

We sketch a simple SDN network for our analysis shown in Figure 16. Two end-points are connected via a network containing three SDN switches, each switch has one or more rules organized in a rule table. Two applications deploy their rules in the switches via the controller. Packets are sent by an end-point into the network and handled by rules in switches. A switch can escalate notifications to the controller to request for instructions on handling a certain packet, the notifications are passed to the applications, which can react by issuing rules to be installed in the switch, or do nothing. A rule in a switch can modify its matched packet, send it out of a port to

the next rule in another switch or to the controller, or drop that packet. Rules can have different priority which decides which rule dominates in case two or more rules in the same rule table match the same packet.

Traffic exchanged between end-points can be of various profiles and transport types, e. g., a VoIP phone call can use constant bit rate (CBR) transported by UDP. They can communicate in unicast or multicast channel. Applications can be deployed simultaneously or in different chronological order. Each has one or more configurations, which reflect the change in the application's behaviour but not its intent. For instance, an End-point Load Balancer may balance traffic per packet or per session while still maintaining its intent of balancing traffic on its replicas. An application can install its rule in one or more switches with different priorities.

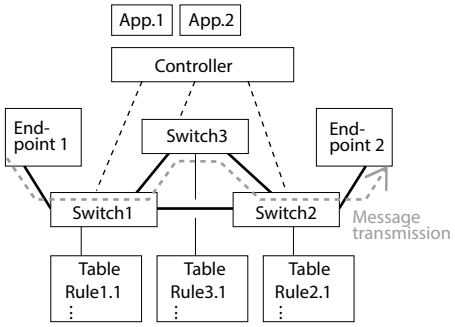


Figure 16: Common SDN elements [108]

We regard a sequence of rules in different SDN switches that handle a message from its reception in the network to its consumption as a program. The message in transit associates with the program state. The program is correct if the message is processed as intended by the application. For example, if the application's intent is to deliver certain traffic from two end-points, the corresponding program is correct when it passed the traffic between these two end-points successfully. A program is incorrect otherwise, i. e., some anomalies occur. An anomaly becomes a conflict if it is caused by rules from different sources according to our definition of conflict in Chapter 1.1.2.

Concept	Distributed computing systems	Software-defined network
operation	instruction	rule action
input	data read from outside the program	messages incoming to the network
output	data delivered outside the program	messages delivered to network edges
environment	operating system	controller
program	sequence of instructions	set of rules in switch tables
state	content of variables	messages in transit

Table 3.1: Comparison of concepts for computer programs and our SDN model [108]

3.1.2 Analogy to distributed computing systems

We notice the similarities between the SDN model and the distributed computing systems running distributed programs. The analogy is summarized in Table 3.1. A distributed program consists of a sequence of instructions, which corresponds to a

sequence of rules in an SDN program. Input and output of the SDN program are messages coming to the network and delivered to end-points. The controller together with applications correspond to the operating system in which the distributed program is running.

Verification of correctness
A program can be verified against its specification, e. g., by using model checker to traverse all possible values of its variables. Provided that the specification and the model of the program are correctly formulated, a program is verified to be correct if its variables do not show any illegal value during the checking.

Obstacles to analytical approach
The ability of control applications to install or remove rules in a switch makes up a major hindrance against the approach of applying in SDN the verification technique commonly used in the software engineering domain. The reason is that it is alike the (hypothetic) operating system being able to modify the code of a program while it is running, which would lead to changing the program to a new one. Therefore, any verification that has been made on the original program needs to be carried out again. The assumption of an isolated environment for running a program that enables its verification does not hold in SDN.

3.1.3 Conclusion

We conclude that the plain analytical approach is not suitable for studying conflicts in SDN. The situations in which an SDN program is changed by control applications need to be examined via experiments, which provide the base for the analysis and for the generalization of conflicts in SDN. As a result, we opt for the experimental approach.

3.2 Experimental approach

In essence, our approach is to perform various experiments and observe conflicts therein. The conflicts are analysed and generalized to conflict classes together with their patterns and characteristics, which enable their detection. This general approach is portrayed in this section via a parameter space and a methodology to perform experiments on that space.

3.2.1 Parameter space

The analysis of the SDN model (Section 3.1.1) allows us to derive exhaustively a parameter space of eight dimensions shown in Figure 17 to perform experiments. Each box surrounding the ticks on some axes denotes multiple values of the same group, e. g., there are various end-point combinations for the 1:1 communication, or different designed topologies. The axes of the parameter space are described in the following.

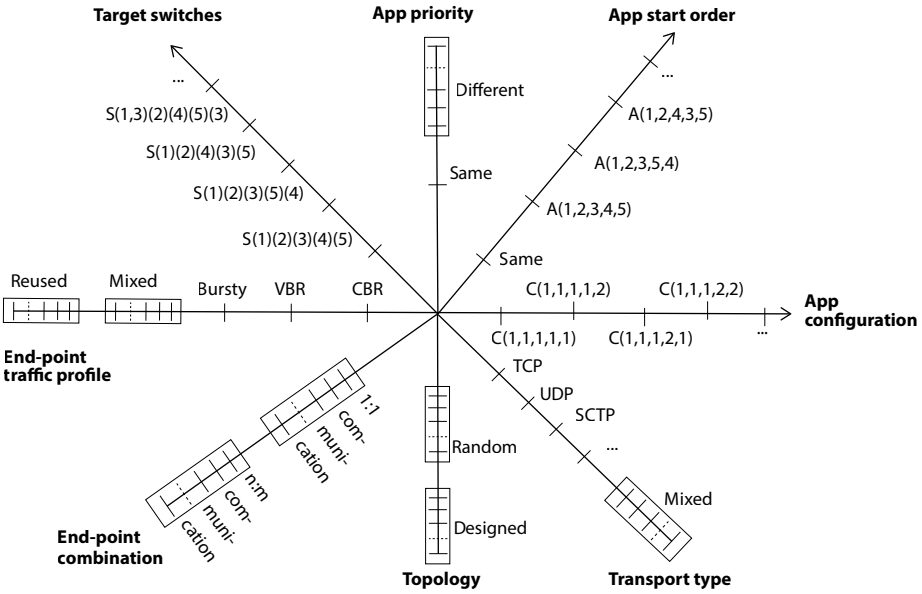


Figure 17: Parameter space [108]

Topology The early demonstration of conflicts in Chapter 1.1.1 shows that network topologies influence the occurrence of conflicts. Topologies can be in designed or randomized structure.

Transport type Transport protocols behave with varying levels of robustness regarding network anomalies. Common transport protocols and their combinations are considered to ensure that the observation of conflicts (or the lack thereof) is not dependent on the properties of the transport layer.

App. configuration The configuration of an SDN application defines its behaviour regarding the introduction or removal of rules in the network, or the decisions taken upon escalated messages. For instance, firewall: stateful or stateless; end-point load balancer: packet-based or flow-based; path load balancer: balancing over first set of links or over second set; bandwidth-related QoS app.: guaranteeing minimums of various bandwidth thresholds, etc. A mark of *C(1,1,1,2,2)* means that applications #1, #2 and #3 use their first configuration while applications #4 and #5 use their second configuration.

App. start order SDN applications may be started and allowed to place their rules in different chronological orders. They can also be started simultaneously (same order). The axis contains all combinations of start orders within a set of applications. The start order of applications is represented as a tuple *A(first, second, …)*.

App. priority Applications may install their rules with the same or different priority. The rule priority has local effect within the rule table of an SDN switch and helps decide which rule is used in case multiple rules can match an incoming packet. We

allow all combinations of priority in a chosen set of applications for experiments.
Target switches An SDN application may realize its function by placing rules on one or more switches. To ensure a total coverage of cases, we introduce all combinations of switches relevant to each application on this axis. The exemplary notation S(1,3)(2)(4)(5)(3) means that application #1 interacts with switches 1 and 3, application #2 with switch 2, application #3 with switch 4, application #4 with switch 5 and application #5 with switch 3.
Traffic profile The traffic profile between end-points might differ depending on the OSI Layer-7 application generating traffic in the data plane. We include distinct traffic profiles as well as their combinations for experiments. The *reused* mark indicates traffic profiles from sources other than our own traffic generators, e. g., from published traces.
End-point combination Communication between end-points can take place in a unicast (1:1) or multicast (n:m) fashion. We include all combinations of end-points, structured by unicast/multicast.

Size of the parameter space
We assume the following figures in the general case:

- t different topologies
- x transport protocols and their combinations
- a applications, each has maximal c configurations
- s switches in each topology
- p traffic profiles
- e end-points get involved in each test

Consequently, there are

- t points on the *Topology* axis,
- x points on the *Transport type* axis,
- $A = \sum_{i=2}^{a} c^i$ points on the *App. configuration* axis, as we vary the number of applications from 2 to a,
- $O = \sum_{j=2}^{a} \binom{a}{j} \times j!$ points on the *App. start order* axis,
- $P = \sum_{k=2}^{a} (k^k - k + 1)$ values on the *App. priority* axis,
- $S = \sum_{l=2}^{a} (2^s - 1)^l$ points on the *Target switches* axis, as one application can be deployed on $1..s$ switches,
- p points on the *End-point traffic profile* axis,
- $C = \sum_{m=2}^{e} \binom{e}{m} \cdot m!$ points on the *End-point combination* axis.

We give the total number of experiments as

$$\Omega = t{\cdot}x{\cdot}A{\cdot}O{\cdot}P{\cdot}S{\cdot}p{\cdot}C = t{\cdot}x{\cdot}\sum_{i=2}^{a} c^i{\cdot}\sum_{j=2}^{a} \binom{a}{j}{\cdot}j!{\cdot}\sum_{k=2}^{a}(k^k{-}k{+}1){\cdot}\sum_{l=2}^{a}(2^s{-}1)^l{\cdot}p{\cdot}\sum_{m=2}^{e} \binom{e}{m}{\cdot}m!$$

Applying the above formula to a case where
$t = 5, x = 2, a = 5, c = 2, s = 10, p = 5, e = 5$,
the number of experiments would be:
$\Omega = 1172083162379999772672000000 \approx 1.17 \cdot 10^{27}$

# switches	Ω (approximate value)
10	$1.17 \cdot 10^{27}$
11	$3.76 \cdot 10^{28}$
50	10^{87}
100	10^{162}

Table 3.2: Growth of the number of experiments by the number of switches

Table 3.2 shows the growth of the experiment space with reference to the number of switches, assuming that the other dimensions' size is fixed.
The addition of one switch into the topology increases the number by an order of magnitude, to $\Omega \approx 3.76 \cdot 10^{28}$. More realistic scenarios increase the size of the space to a degree beyond any hope of a direct experimental approach: for 50 switches, $\Omega \approx 10^{87}$ and for 100 switches, $\Omega \approx 10^{162}$. Obviously, mechanisms to reduce the size of the space and a methodology suitable to perform highly automated experiments are necessary.

Reducing the size of the space
We can exploit the specific property of the control applications selected for experiments to reduce the size of the *target switches* axis. For example, a firewall should not be deployed on all devices but on those at the network boundary, an end-point load balancer should place its rules on devices belonging to the paths connecting its target end-points.

To serve the goal of studying conflicts, we can choose pragmatically points in the parameter space where conflicts are potential, i. e., those with control applications having common concerns. For instance, conflicts are more likely between two applications interested in TCP traffic and having rules deployed on the same switch.

It is worthwhile to choose "compact" subspaces from the large space to conduct experiments. A "compact" subspace contains only valid points and none of them is redundant. A point is invalid if the association between marks on different dimensions fails, for instance, a mark of C(1,1,1) on the *App Configuration* axis cannot be associated with a mark of A(1,2) on the *App Start Order* axis as the former is applicable for three applications while the latter applies for two applications only. A redundant point repeats the same characteristics of another already-processed point. Consider a case in which two points having the same values on all axes of the parameter space except for the *priority*, the mark on the *priority* axis of the first point is {2, 2} and the second is {3, 3}. Although these two marks are different, they reflect the same property: the two applications in concern are deployed with the same priority, which renders one of the mentioned points redundant. By only choosing valid and "concise" points for experiments, we can obtain a more meaningful dataset for studying conflicts. We provide more details on generating "compact" subspaces with respect to application-related dimensions in Section 3.4.3.

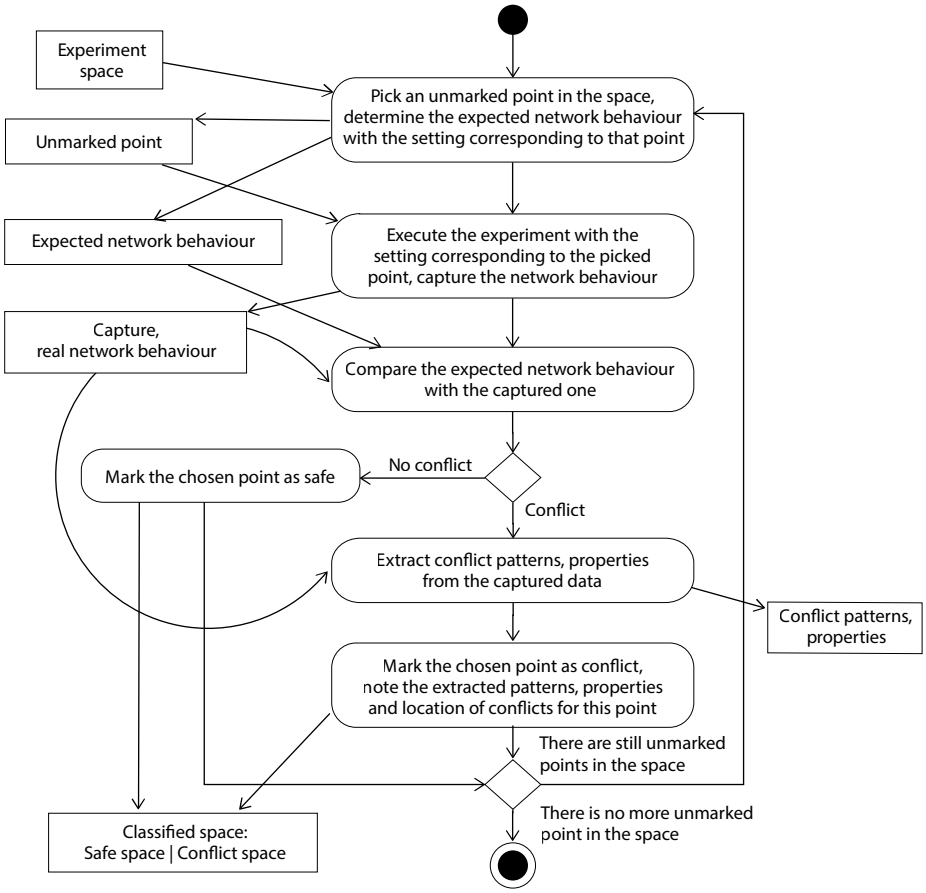


Figure 18: Methodology for exploring conflicts [108]

3.2.2 Methodology

In principle, experiments are conducted through each point in the parameter space (Figure 17) to study conflicts. The analysis of conflict cases allows the extraction of patterns and properties of conflicts, from which conflict classes are formulated. In practice, we can choose different subspaces to perform experiments. Figure 18 sketches the steps of our methodology.

1. A point in the space is selected as the starting point. The expected network behaviour of each control application in that point is determined by deploying it in isolation and capturing all appropriate data, e. g., packet captures for each involving port of end-points, SDN switches, rule tables of SDN switches, link throughput. These captures are then analysed to extract only the data germane to the set of control applications involved in that point. The applications are assumed to be reasonably correct and reliable, which means their (re)actions are always

consistent and deterministic against network events, e. g., incoming of a packet, violation of a predefined bandwidth threshold. We notice that within the scope of our work, we do not consider network faults or changes to topologies during the observation phase, i. e., we do not simulate the failure or the introduction of network devices.

Example: An End-point Load Balancer with configuration 1 is deployed on switch 1 and a Path Load Balancer with configuration 1 on switch 1. End-points 1 and 2 transmit UDP with a constant bit rate (CBR) traffic profile on a topology of 10 switches. When deployed in isolation, the Path Load Balancer (PLB) splits a flow into equal sub-flows over different paths, and the End-point Load Balancer (EpLB) balances incoming requests over a list of end-points. The traffic throughput on the concerned paths and the number of communication sessions at each end-point are logged for the case of PLB and EpLB, respectively. The rule tables of the switches where these balancers install rules are also recorded.

2. The experiment is carried out. The chosen applications with the associated configurations are deployed on the network in the specified order, the end-points with specified traffic profile transmit the transport protocol described in the selected point. The network behaviour is captured, including, e. g., traffic throughput of each switch port, rule tables, the number of communication sessions at each end-point and the state of these sessions (successful or failed).
3. The experiment's outcome is compared with the expected outcome by matching the relevant data items from the capture of each application to the data gathered when running that application in isolation. This entails performing different types of comparison, depending on the application type. For instance, to observe the effect of the End-point Load Balancer, the statistics data at the traffic source and the target end-points are of interest. For the Path Load Balancer, the number of packets/flows on the target paths should be considered, other information is out of concern.

   If the behaviour matches, the point is judged as "safe" and the next experiment associated with a new point is executed. If the result suggests a conflict due to different behaviour, an analysis step follows.
4. The captured data are analysed to ascertain the presence of conflicts and to classify them. If enough conflicts of a class have been observed, its patterns and/or properties are formulated. A pattern constitutes a generalisation of a certain combination of rules (and state) that lead to a conflict of a specific class. Thus, a pattern can be viewed as a "signature" of a conflict class. The properties/patterns of a conflict class enable the classification and detection of conflicts belonging to that class, as presented in Chapters 4 and 5.
5. The point is marked as "safe" or "conflict". The procedure is repeated if unmarked points exist in the space.

The dataset created from the classified space and the collection of conflict patterns, properties are valuable for the handling of conflicts. In this work, we make use of them for the classification and the detection of conflicts. In general, they are also beneficial for further conflict handling efforts, e. g., conflict resolution and avoidance.

3.3 Explored subspaces

A point in the parameter space is a tuple of values from all dimensions of the parameter space. Each point encodes the setting of an experiment. According to the way we perform experiments, each subspace can be associated with the points generated in a chosen network topology. Roughly speaking, for each network topology, we create a corresponding test-bed containing virtual machines functioning as SDN devices (switches) and end-points specified in that topology, and an SDN controller. Control applications are then deployed with various settings concerning their configurations, start order, priority and target switches. Next, traffic is generated between different combinations of end-points. Results are logged in a dataset and anomalous cases are analysed to extract conflict features. These steps are explicated in Sections 3.4 and 3.6.

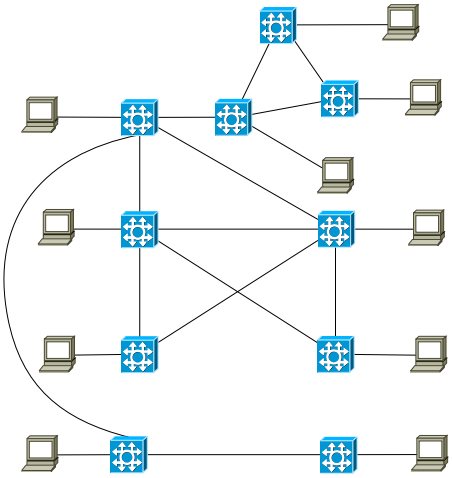


Figure 19: A designed topology with 10 switches and 10 end-points

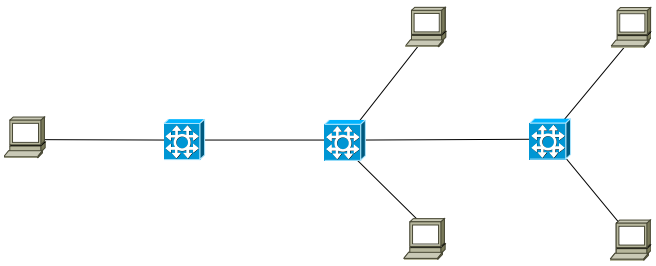


Figure 20: Simple linear topology

We have performed experiments on six designed and six random network topologies, four of them are shown in Figures 19, 20, 21 and 22. Control applications are diverse in functions and thus in how they control the network, including routing, firewalling, traffic engineering (a complete list of control applications is described in Section 3.5.2). The actions of their rules include dropping packets, forwarding packets out of a set

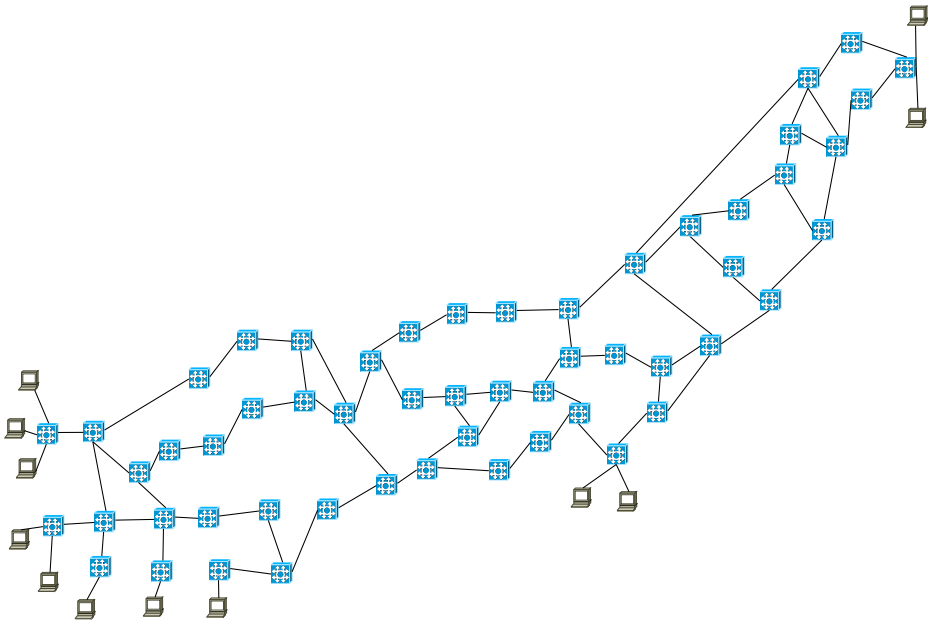


Figure 21: A designed topology simulating the core backbone of the Nippon Telegraph and Telephone (NTT) network [1] in Japan (55 switches, 12 end-points, reproduced from Reyes' work [90])

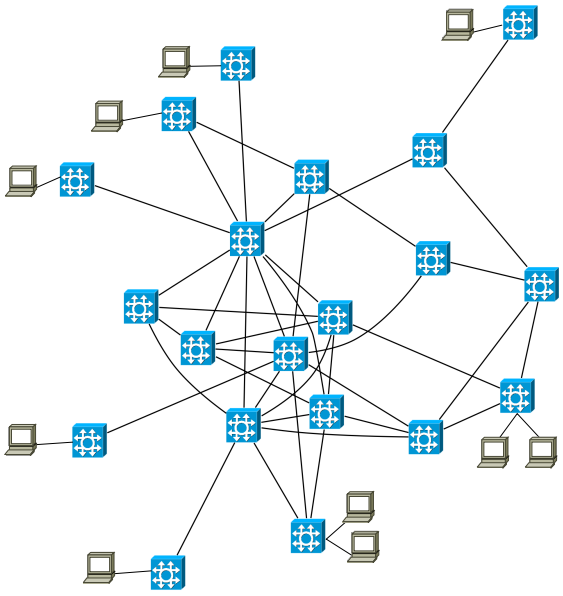


Figure 22: A random topology generated with the Barabasi-Albert model [4] (20 switches, 10 end-points, reproduced from Reyes' work [90])

of ports, modifying packet headers, or a combination of those. They are deployed in different priority and start order, their target switches also vary. TCP and UDP traffic is generated between end-points mostly in unicast communication scheme with VBR or CBR traffic profiles. Table 3.3 highlights the important features of the explored subspaces.

Category	Value	Note
# Topologies	12	6 designed topologies, 6 random topologies, # end-points ranges from 4 to 21, # switches from 3 to 55
# Applications	14	containing fundamental functions (e. g., topology discovery, ARP cache, NDP cache) and applications involved directly in conflict study, e. g., End-point Load Balancer, Path Load Balancer, Firewall, Path Enforcer…(see Section 3.5.2 for the full list of applications)
App. configuration	$1 \rightarrow 5$	each app. has at least 1 configuration, at most 5
App. start order	same and different	at least two apps. are co-deployed in an experiment, at most 5
App. priority	same and different	the co-deployment of 2 apps. yields 3 combinations of priority, there are 541 combinations for 5 apps (see Section 3.4.3)
Target switches	$1 \rightarrow$ all	each app. can have one target switch or more, or even deploy its rules on all switches, e. g., the Shortest Path First app
Ep. Traffic Profile	CBR and VBR	*netcat* and *iperf* programs are used to generate TCP/UDP traffic
EP. Combination	unicast, multicast	multicast traffic is generated for the MEADcast app. in IPv6, all other apps. are active on IPv4 unicast traffic
Transport type	TCP, UDP	–
# Experiments	11,772	8796 experiments expose no conflict, 2976 experiments show potential conflicts (these experiments are conducted automatically, the manual experiments are not counted)

Table 3.3: Information of the explored subspaces for conflicts

Our published results[1] contain 11,772 experiments conducted automatically by our framework (see Section 3.4), among those 2976 experiments ($\approx$ 25.3%) exhibit unexpected output that need to be analysed to conclude possible conflicts, these results are visualized in Figure 23. The unexpected output is determined by comparing the network behaviour in isolated deployment and in co-deployment of control applications. Conflicts are potential in experiments yielding the unexpected output, their existence can only be confirmed via the manual analysis, or with the employment of a conflict detection program. A smaller number of experiments have been carried out manually for the stepwise examination of conflicts, they are not counted here since their results are not collected in an organized manner as in the automatic execution

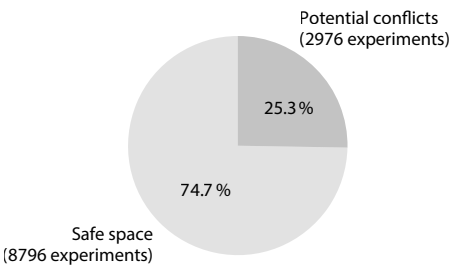


Figure 23: The proportion of the safe space and the space containing potential conflicts in the total number of 11,772 experiments conducted automatically by our framework

1 https://github.com/mnm-team/sdn-conflicts

with the framework. In the next section, we introduce a framework that enables the massive execution of experiments and the collection of the dataset for conflict study.

3.4 A framework for automating experiments

The extremely large size of the parameter space for experiments figured out in Section 3.2.1 urges the automation in their execution besides the need to reduce its size. We describe in this section our approach to achieve these goals and implement a framework therefrom.

3.4.1 Generating SDN test-beds

We extend the existing work on creating virtual machine infrastructure by Danciu et al. [23] to generate our SDN test-bed in a highly automatic manner. The network topology, i. e., the connections between SDN switches and end-points, is described in a file followed a predefined format, this is input in a script to generate the SDN test-bed. The test-bed contains a control machine, which is a virtual machine built on the KVM software [53]. Inside the control machine, multiple virtual machines are created based on the Xen software [5], one of them is the SDN controller, the others are SDN switches and end-points connected to each other according to the network topology chosen for experiments. Each test-bed thus corresponds to a network topology, from which a number of subspaces of the parameter space are constructed and a sequence of experiments are carried out. The use of virtual machine for each device facilitates the customization and monitoring of each individual device for conflict examination.

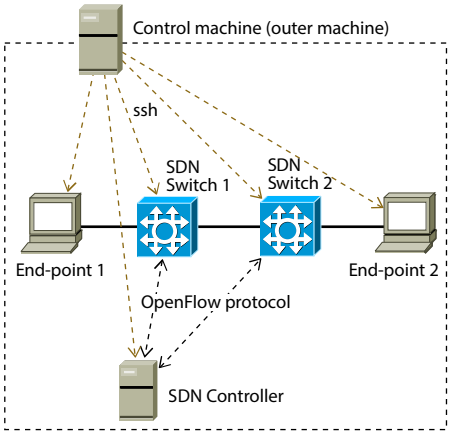


Figure 24: Illustration of a simple test-bed for a network containing two end-points and two SDN devices

A simple test-bed containing a controller, two switches and two end-points is shown in Figure 24. We describe each component of the test-bed in the following.

Control machine (outer machine)
This virtual machine is built on the KVM [53] software and is also referred to as the outer machine. All other test-bed components are packed within this outer machine so that their activities are isolated from the outside networks, which helps avoid any possible interference from outside that could otherwise impact the experiment process in an uncontrolled manner. The control machine communicates with other

virtual machines via the *ssh* channel. It coordinates, assigns and schedules the jobs to be executed in the SDN controller, SDN switches and end-points according to the settings of the parameter space, then collects data from these machines and performs analysis to detect anomalous cases, and finally creates the dataset for subsequent tasks, e. g., analysis of conflicts for their classification, evaluation of the conflict detection prototype.

Example Consider the setting for Experiment 1 (Section 3.6.3) described in Figure 27a, which we outline here for the sake of readability. In this experiment, two applications End-point Load Balancer (EpLB) and Shortest Path First (SPF) are deployed with configuration 1 each, in the same starting order, and the same priority of 1, EpLB places rules in switch 7 while SPF interacts with all switches, the end-points use constant bit rate (CBR) transported via TCP, PC1 and PC2 are traffic sources, PC3 and PC4 are traffic sinks. The test-bed is generated according to the topology *topo1*. In essence, the control machine needs to run a script that performs the following tasks via the *ssh* channel.

1. Preparing a clean environment for experiments, e. g., stopping all controller-related processes that possibly were executed earlier in other experiments, cleaning the ARP cache in the end-points and rule tables in the switches, removing trace files created by other experiments.
2. Starting the EpLB and SPF control applications in the controller simultaneously, each with configuration 1, priority 1. Specifying the target switch for EpLB as 7, and SPF as all.
3. After the control applications were started successfully in the controller, running the pre-installed software in PC3 and PC4 to create the server processes, then executing the software in PC1 and PC2 to create the client processes that establish connections to the server processes. The pre-installed software has to generate TCP traffic after the CBR traffic profile.
4. After the communication between end-points were finished (or after a timeout period), creating the logs or trace data in the end-points and in the switches, e. g., the state (being successful or failed) of the communication between the end-points, rule tables in each switch, throughput logged at each port.
5. Analysing the collected data to detect anomalous cases, storing the result in the dataset together with the collected data, and the settings of the experiments (which corresponds to a point in the parameter space).
6. Returning the clean environment for subsequent experiments as in step 1.

All of the above steps are performed automatically in our framework.

SDN Controller

SDN Controller contains all control applications. Each control application has its global and local configurations. The global configuration specifies the priority, the

configuration number (according to the *app. config* axis in the parameter space), and the target switches of the involved applications. The global configuration can be rewritten by the control machine. The local configuration is specific to the control application and cannot be changed by the outer control machine, e. g., for EpLB, the first local configuration can be to balance requests over some set of servers, the second configuration over another set of servers, for PLB, the first local configuration is to balance traffic if the throughput exceed the first threshold value, the second local configuration corresponds to the second threshold value. The control application reads these configuration files and behaves accordingly.

SDN Switch
Also known as SDN device. The Open vSwitch software [84] with the enabled OpenFlow features is employed to make a virtual machine become an OpenFlow switch. Script files are put in each switch that can be called by the control machine to measure throughput of the switch's interfaces during experiments, or to dump the rule tables of the switch. These results are collected by the control machine.

End-point
The software to generate traffic is pre-installed in the end-points. According to the value of the End-point traffic profile, transport type and end-point combination, the control machine will run the appropriate software in the end-points to generate the desired traffic. The communication state is logged in the end-points, e. g., how many (UDP/TCP) sessions were established, and with which end-points, how many were successful or failed. The control machine collects these results for further analysis.

3.4.2 Encoding experimental subspaces

Each test-bed corresponds to a network topology, it is alike that we choose a mark on the topology axis of the parameter space for the first step before selecting the values of the other axes. Hence, after the test-bed is generated, we need to choose different subspaces derived from the network topology by varying the other dimensions of the parameter space.

The tactics for reducing the parameter space's size mentioned in Section 3.2.1 are applied in choosing the experimental space. The control applications and their settings are chosen in such a way that conflicts are highly potential, i. e., they share some common concerns in at least one relevant dimension, for instance, all of them are interested in TCP traffic, they place rules in the switches that probably belong to the traffic paths between end-points according to our estimation from the network topology. Listing 3.1 shows an exemplary encoding of a subspace.

```
#
1  plb eplb spf  # app. name, e.g., Path Load Balancer, End-point Load Balancer
2  2 2 1         # the number of configurations of each app. in line 1
3  tcp udp       # transport layer
4  7 4:7:all     # target switches
5  cbr           # end-point traffic profiles
6  10            # the number of switches in the network topology
7  10            # the number of end-points in the network topology
8  1 2:3 4       # end-point combinations, sources : targets
9  topo1         # name of the tested network topology
```

Listing 3.1: Encoding of an experimental subspace

1. The 1. line names the three control applications *PLB*, *EpLB* and *SPF routing* that take part in the experiments.
2. The 2. line specifies the configuration of each application in the 1. line. In this case, *PLB* has 2 different configurations, *EPLB* and *SPF routing* each has only 1 configuration.
3. The 3. line lists the transport type of the traffic to be generated by the end-points, which is TCP and UDP in this example.
4. The 4. line shows the target switches on which each control application will be deployed, delimited by the colons (":"). *PLB* is deployed on switches 7 and 4, *EPLB* on switch 7 and *SPF routing* targets all switches.
5. The 5. line specifies the traffic profile to be generated by the end-points.
6. The 6. line shows the total number of switches in the network topology. The information about the number of switches and end-points (shown in the next line) is useful in preparing a clean test-bed for performing each experiment. For example, if there are n switches in the topology, their names according to our naming scheme will be $sw_1, sw_2, \ldots sw_n$, the control machine can communicate with these switches by their name via *ssh* to clean their rule tables when needed.
7. The 7. line shows the total number of end-points in the network topology.
8. The 8. line indicates the combinations of end-point communication. In this case, end-points 1 and 2 are the traffic sources, end-points 3 and 4 are the traffic sinks. The communication combinations are: 1–3, 1–4, 2–3, 2–4.
9. The 9. line specifies the name of the topology used for this experiment round. As mentioned above, each test-bed corresponds to a fixed network topology and we cannot change the network topology within a test-bed. To perform experiments on a new topology, a new test-bed needs to be generated from its topology's specification.

Note that a subspace as specified in this listing contains a set of points, each point is a setting in the parameter space from which a single experiment is performed. Each subspace is thus associated with a sequence of experiments. In the above encoding, lines 6, 7 and 9 specify the information related to the network topology and thus are fixed in the test-bed. We can change the other lines to create new subspaces

associated with the current topology, e. g., specifying other control applications in line 1, changing the target switches of the control applications in line 4, or the end-point combination in line 8.

This encoding facilitates the analysis of the dataset obtained after finishing the experiments and the re-execution of an experiment point in the parameter space to reproduce the outcome for examining an individual case if necessary. Next, we explain how the points of the subspace, i. e., the experiment settings are generated from the encoding of the subspace.

3.4.3 Generating compact values for dimensions related to control applications from a subspace's encoding

We observe that the values for dimensions related to control applications, e. g., *app. config, app. start order* and *app. priority* can be succinctly generated based on the list of control applications involved in experiments and the number of configurations of each. Our method of producing the configuration values for these dimensions ensures that all generated points for experiments are valid and none of them is redundant.

Choosing control applications from the application list

Each application is determined by its index in the application list. For instance, from line 1 of Listing 3.1, the index of *PLB* is 0, *EpLB* 1 and *SPF* 2. The combinations stemming from these three applications indicated by their indices are shown in Listing 3.2.

```
#
1    0 1
2    0 2
3    1 2
4    0 1 2
```

Listing 3.2: Apps combinations

For n control applications, experiments are conducted for different cases from deploying two applications, then three applications, and so on until n applications are tested. The permutation of the indices in each application combination (e. g., each line in Listing 3.2) results in the associated values for the *app. start order* dimension. In our test, we deploy all control application simultaneously to facilitate the automation of the experiments. In manual deployment for analysis in depth, we can deploy control applications in different order as intended to observe conflicts. The different order entails other concerns, e. g., when should the second application be started, when come the third, how long should the interval between their start time be. The choice and the result seem to depend much on the characteristics of the control applications, the schedule and the type of traffic generated in the data plane.

Configuration generation

The configuration combinations are derived from each application combination (i. e., each line in Listing 3.2). The number of configurations of each application is specified in the encoding of the experimental subspace. For example, line 1 of Listing 3.2 *0 1* refers to a combination of *PLB and EpLB*, each has two configurations according to line 2 of Listing 3.1, say *config. 1 and confg. 2*, we can infer thus 4 configurations for these application combinations as shown in Listing 3.3.

```
#
1    1 1
2    1 2
3    2 1
4    2 2
```

Listing 3.3: Configuration combinations for *PLB* and *EpLB*

Priority generation

Priority implicates the rule order generated by different applications. The order pattern should be unique to eliminate redundant setups and hence, redundant experiments. For 2 applications, the priority combinations are:

```
#
1    2 2
2    2 3
3    3 2
4    3 3        # duplicated with 2 2, two apps. having the same priority
```

Listing 3.4: Priority combinations for two apps

Note that the application priority starts at 2, since the priority of 0 is reserved for the table-miss entry and the priority of 1 for basic functions, e. g., ARP cache, ARP/ICMP switching.

The interesting fact is that the number of essential priority combinations is not intuitively derivable from the number of applications involved. One can mistakenly infer that there are 4 priority combinations for 2 applications, 27 for 3 applications and so on. In fact, it is 3 for 2 applications because the combination of *3 3* is a duplicate pattern of *2 2* as noted in Listing 3.4, there are 13 combinations for 3 applications shown in Listing 3.5, in which the priority relationships are denoted by the relationship between *a, b, c* in the *pattern* column, the duplicate combinations are pointed out. For 4 control applications, the total number of combinations is 75, it is 541 for 5 and 4863 for 6 applications. The method for generating the priority combinations for n applications is portrayed in Appendix 7.

```
#          Main combi.    Pattern          Duplicated combi.
1          2 2 2          # a=b,a=c,b=c    3 3 3    4 4 4
2          2 2 3          # a=b,a<c,b<c    2 2 4    3 3 4
3          2 3 2          # a<b,a=c,b>c    2 4 2    3 4 3
4          2 3 3          # a<b,a<c,b=c    2 4 4    3 4 4
5          2 3 4          # a<b,a<c,b<c
6          2 4 3          # a<b,a<c,b>c
7          3 2 2          # a>b,a>c,b=c    4 2 2    4 3 3
8          3 2 3          # a>b,a=c,b<c    4 2 4    4 3 4
9          3 2 4          # a>b,a<c,b<c
10         3 3 2          # a=b,a>c,b>c    4 4 2    4 4 3
11         3 4 2          # a<b,a>c,b>c
12         4 2 3          # a>b,a>c,b<c
13         4 3 2          # a>b,a>c,b>c
```

Listing 3.5: Priority combinations for three apps

3.4.4 End-point related dimensions

The *end-point combination* dimension is specified in the encoding of the experimental subspace (e. g., line 8 in Listing 3.1). As mentioned earlier in Section 3.5.2, we do not deploy the *MEADcast* application in the automatic massive test but only in manual run, which is the sole control application that influences the end-points' multicast communication. The other control applications target normal unicast traffic, thus the subspace encoding reflects only unicast communication between end-points. Traffic generation software is pre-installed in all end-points, then depending on the values of the *end-point traffic profile, transport type* dimensions, the appropriate software is called from the control machine to generate the desired traffic.

Example Lines 3, 5, 8 in Listing 3.1 specify the end-points 1 and 2 to communicate with the end-points 3 and 4 via TCP and UDP using the constant bit rate (CBR) encoding. The control script from the control machine starts in end-points 3 and 4 the server processes at some certain port using the *nc* (netcat) software, then starts in end-points 1 and 2 the client processes to communicate with both end-points 3 and 4, the input and the options of the *nc* command allows the TCP and UDP traffic to be generated in the CBR encoding.

3.4.5 Expected and observed network behaviour

Our definition of conflicts in Chapter 1 implies the comparison between the expected and observed network behaviour in detecting conflicts. The expected network behaviour is obtained from the isolated deployment of each control application. In contrast, the observed network behaviour is procured from the co-deployment of different control applications.

Logged data for analyzing network behaviour
There are various kinds of data in each experiment that could be logged for analysis, e. g., traffic dumps for each interface of each network device, the number of packets being dropped/forwarded by an interface, the speed and latency of the file transmission or communication between end-points. Storing all traffic dumps would be useful in most cases but requires large storage space. In comparing the expected and real network behaviour to determine the presence of conflicts, we find the below information necessary and sufficient.
Interface throughput over time The real-time traffic dumps are used to calculate the traffic throughput over time of each interface of a network device without having to store them. The throughput data are important in ensuring the correct outcome of control applications that control the network bandwidth, e. g., Path Load Balancer, bandwidth-related Traffic Engineering.
Communication state between end-points While deploying control applications in isolation, the communication between end-points must be successful except for those applications that explicitly drop some kind of traffic, e. g., firewall. The co-deployment of control applications resulting in unsuccessful communication between end-points would show the sign of anomalies that need further analysis. The log data of communication state between end-points are also useful in ensuring the correct behaviour expected by the control applications that redirects end-points communication, e. g., EpLB.
Rule tables The rule table can be analysed to ensure that the rules of a control application are effective at its interested points (switches) in the network. There may be the case in which the communication between end-points is successful in co-deployment of the control applications while the rules of one of them are not present at its target switch, although those rules are shown in the isolated deployment. The correct network behaviour in this case deserves extra analysis to determine whether it is correct by chance and if conflicts exist.

Comparison of expected and real network behaviour
Each control application is deployed in isolation with all of its configurations and settings specified in the encoding of the subspace, e. g., in Listing 3.1. The network behaviour in co-deployment of control applications is compared with that of each application based on the logged data mentioned above. The deviated results from the comparison indicate the presence of anomalies that suggests further examination to detect and classify conflicts.

We choose the metric among the logged data to compare network behaviour based on the intent of the control applications participating in the co-deployment, while rule tables are always compared. For instance, EpLB does not concern about the interface throughput but the communication state between end-points, e. g., if the communication is successful, how many sessions are handled by a certain end-point; in contrast, PLB is not interested in how communication sessions are balanced at end-

points but the throughput of each interface of its target switches. After determining the metric according to the control applications involved, we perform the comparison as follows:

- If the metric includes the interface throughput, the maximal value that was logged is used. We allow a difference within a threshold, e. g., if we choose a threshold of 5 Mbps and the maximal throughput of an interface in the isolated deployment of the control application is 60 Mbps while it is 64 Mbps in the co-deployment, then no problem is raised regarding this metric.
- If the metric includes the communication state between end-points, we compare the state of the communication sessions reported at end-points, e. g., via the exit code of the program used for generating traffic, and the number of sessions handled by each end-point acting as a server in the communication.
- In terms of rule tables, we compare the number of rules installed by the control application in concern at its target switches. We differentiate rules from different control applications based on the *cookie* values that we assign them. The presence of a rule in a switch indicates two possibilities: i) the control application installs that rule independent of the traffic coming to that switch, or ii) the control application reacts to the arrival of certain traffic in the switch by installing that rule. In both cases, if there exists no disruption, the number of rules of an application should stay the same in the isolated deployment as well as in the co-deployment.
- Some control applications, e. g., firewall, can drop traffic, which could fail the communication sessions between end-points that violate its policy. If none of these applications takes part in the co-deployment but there exists failure in the communication between end-points, an alert of conflict is raised.

3.4.6 Dataset

Experiments' results are logged in a dataset, that is organized in such a way that each individual experiment can be conveniently replayed to reproduce the logged results. We exploit the dataset to study features of conflicts to classify them (Chapter 4).

Dataset structure

We record the result of each experiment in terms of conflicts, its settings and rule tables of network devices in a dataset. Listing 3.6 illustrates the file structure of the dataset.

```
|-- 200820_152207
|   |-- all_config
|   |-- conflict.txt
|   |-- sw1_flowdump.tar.gz
|   |-- sw2_flowdump.tar.gz
|   `-- ...
|-- 201109_003301
|   |-- all_config
```

```
|   |-- conflict.txt
|   |-- sw1_flowdump.tar.gz
|   |-- sw2_flowdump.tar.gz
|   `-- ...
|-- ...
|   |-- ...
`-- massive
    |-- arpcache.py
    |-- eplb_config_local1
    |-- eplb_config_local2
    |-- eplb.py
    |-- spf.py
    `-- ...
```

Listing 3.6: An excerpt of the file structure of an exemplary dataset

We refer to all experiments associated with a subspace (e. g., the subspace specified in Listing 3.1) as an experiment round, these have their results stored in the same directory named after the time point in the format *Year-Month-Day_Hour-Minute-Second* (e. g., 200820_152207) when the round started. The time point serves as the identifier (or shortly *id*) of the experiment round, which is required if we need to reproduce and verify the results of that round. In an experiment round, each control application is first deployed in isolation with each of its configuration, then they are co-deployed according to the settings of each point in the subspace. All settings of these points are written in a configuration file, named *all_config*, in the format: *app_name:app_config:app_priority:target switch*. Since control applications are always simultaneously co-deployed in this framework, we omit the value of the *app start order* dimension in the configuration file. An excerpt of an exemplary *all_config* file is shown in Listing 3.7. We can observe that it begins by the encoding of the subspace similar to Listing 3.1 (lines 1–9), line 11 is the *id* of the experiment round. There are 79 points in total in the subspace shown in this *all_config* file, which means 79 individual experiments for the co-deployment of control applications need to be conducted (likewise, there are 76 points in the subspace described in Listing 3.1). The way we store the settings in the *all_config* file allows the precise and handy reproduction of each experiment point or the whole experiment round.

```
#
1       eplb pplb plb
2       1 4 1
3       tcp udp
4       7:3 4:7 5 6
5       cbr
6       10
7       10
8       1 2 8 9:3 4 5 6
9       topo1
10
11      200820_152207
12      point 1
```

```
13      eplb:1:2:7
14      pplb:1:2:3 4
15      point 2
16      eplb:1:2:7
17      pplb:1:3:3 4
18      point 3
19      ...
        point 78
        eplb:1:4:7
        pplb:4:2:3 4
        plb:1:3:7 5 6
        point 79
        eplb:1:4:7
        pplb:4:3:3 4
        plb:1:2:7 5 6
```

Listing 3.7: An excerpt of an exemplary *all_config* file

Deviated network behaviour between the isolated and the co-deployment of control applications occurring during the experiment round is logged in a file named *conflict.txt*, an exemplary excerpt is shown in Listing 3.8. The logged data mentioned in Section 3.4.5 are used for the comparison. The first line of the *conflict.txt* file is the *id* of the experiment round. Points that do not show up in the *conflict.txt* file correspond to the safe settings, e. g., points 4, 5, 6 in the above example.

```
#
1       200820_152207
2       error with nc, point = 1
3       nc from pc2 to pc3(UNKNOWN) [192.168.1.3] 3423 (?) : Connection timed out
4       error with nc, point = 1
5       nc from pc9 to pc3(UNKNOWN) [192.168.1.3] 3493 (?) : Connection timed out
6       error with iperf, point = 1, iperf from pc2 to pc3
7       [  3] WARNING: did not receive ack of last datagram after 10 tries.
8       error with iperf, point = 1, iperf from pc9 to pc3
9       [  3] WARNING: did not receive ack of last datagram after 10 tries.
10      error with nc, point = 2
11      nc from pc2 to pc3(UNKNOWN) [192.168.1.3] 3423 (?) : Connection timed out
12      error with iperf, point = 2, iperf from pc2 to pc3
13      [  3] WARNING: did not receive ack of last datagram after 10 tries.
14      conflict, bw difference=5.94, point=2,sw5, eth3
15      conflict, bw difference=5.86, point=3,sw5, eth3
16      error with nc, point = 7
17      nc from pc2 to pc3(UNKNOWN) [192.168.1.3] 3423 (?) : Connection timed out
18      error with nc, point = 7
19      nc from pc9 to pc3(UNKNOWN) [192.168.1.3] 3493 (nut) : Connection timed
    ↪ out
20      ...
```

Listing 3.8: An excerpt of an exemplary *conflict.txt* file

After each experiment finished, the rule tables of all network devices are stored. In our framework, we use the *ovs-ofctl dump-flows* command for this purpose and we name the rule tables obtained by this command as *flow dump*. After the whole ex-

periment round finished, all of these rule tables are compressed into a tarball[2] file and stored in the dataset under the directory named by the *experiment round's id*. The content of a tarball is illustrated in Listing 3.9, the name of each flow dump is composed of the prefix *sw<i>_flowdump_* and the suffix which is either the point number in the experiment subspace or the control application's name and its configuration number in case it is deployed in isolation.

```
#
1       sw1_flowdump_1
2       sw1_flowdump_2
3       sw1_flowdump_3
4       sw1_flowdump_4
5       ...
6       sw1_flowdump_eplb_1
7       sw1_flowdump_eplb_2
8       sw1_flowdump_plb_1
9       ...
```

Listing 3.9: An excerpt of an exemplary tarball of *sw1_flowdump.tar.gz*

An excerpt of the raw content of a flow dump in the tarball is shown in Listing 3.10. These flow dumps are crucial in analyzing conflicts.

The *massive* directory in Listing 3.6 contains the source code of all control applications used for the experiments. We include them in the dataset so that any third party can repeat the experiments and reproduce the results when necessary.

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=77.728s, table=0, n_packets=147, n_bytes=8820, priority
      ↪ =65535,dl_dst=01:80:c2:00:00:00,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x400, duration=69.193s, table=0, n_packets=11, n_bytes=753, idle_timeout
      ↪ =1800, priority=2,tcp,in_port=2,nw_src=192.168.1.1,nw_dst=192.168.1.3,
      ↪ tp_src=2313,tp_dst=3413 actions=output:3
 cookie=0x400, duration=62.161s, table=0, n_packets=1, n_bytes=74, idle_timeout
      ↪ =1800, priority=2,tcp,in_port=2,nw_src=192.168.1.2,nw_dst=192.168.1.3,
      ↪ tp_src=2323,tp_dst=3423 actions=set_field:00:16:3e:11:11:04->eth_dst,
      ↪ set_field:192.168.1.4->ip_dst,output:3
 cookie=0x700, duration=69.156s, table=0, n_packets=0, n_bytes=0, idle_timeout
      ↪ =1800, priority=2,tcp,nw_src=192.168.1.3,nw_dst=192.168.1.1,tp_src=3413,
      ↪ tp_dst=2313 actions=output:2
 cookie=0x700, duration=65.646s, table=0, n_packets=0, n_bytes=0, idle_timeout
      ↪ =1800, priority=2,tcp,nw_src=192.168.1.4,nw_dst=192.168.1.1,tp_src=3414,
      ↪ tp_dst=2314 actions=output:2
 ...
```

Listing 3.10: An excerpt of an exemplary flow dump

2 https://wiki.debian.org/TarBall

Retrieving information of an experiment from the dataset
The organization of the dataset allows determining the setting (corresponding to a point in the parameter space) of an experiment and retrieving the rule tables of network devices produced by that experiment.
Example Lines 10–14 of the file *conflict.txt* in Listing 3.8 report that point 2 of the subspace experienced a problem related to the communication between PC2 and PC3 and the traffic throughput difference of the interface *eth3* of switch 5 was higher than the chosen threshold (we specified the threshold of 5 Mbps for our framework). The first line of this file specifies the *id*, being *200820_152207*, of the experiment round to which the experiment corresponding to point 2 belongs, the directory containing the file *conflict.txt* is named after this *id* (see also Listing 3.6). In the file *all_config* under this directory, lines 15–17 (see Listing 3.7) specify the setting of this point: EpLB with configuration 1, priority 2, interacts with switch 7 and PPLB with configuration 1, priority 3, interacts with switches 3 and 4. In this case, our framework deploys additionally the SPF control application (see Section 3.5.2) with the priority of 1 to provide the routing function since it is not specified in this point of the subspace. Lines 3, 5 and 8 of *all_config* state that the end-points 1, 2, 8, 9 transmit constant bit rate via TCP and UDP to the end-points 3, 4, 5, 6; we use the tools *nc* and *iperf* in our framework to generate such traffic. We can retrieve the *flow dumps* of the target switches of the control applications in point 2, i. e., switch 7 for EpLB and switches 3, 4 for PPLB by extracting the tarballs *sw7_flowdump.tar.gz, sw3_flowdump.tar.gz, sw4_flowdump.tar.gz* to get the relevant files *sw7_flowdump_2, sw3_flowdump_2, sw4_flowdump_2*. There we also have the rule tables of the isolated deployment in the files *sw7_flowdump_eplb_1, sw3_flowdump_pplb_1, sw4_flowdump_pplb_1*. With the setting inferred, we can replay the experiment to inspect step by step the occurring conflicts when necessary.

3.4.7 (Re)Production of the test-bed

Having explained the principles in building test-beds for experimenting conflicts, we delineate the technical aspects useful for its (re)production in this section. The test-bed in Figure 25 is reused for illustration. All programs mentioned in this section are available in our published codebase[3].

Environmental settings
In a normal setup, the whole test-bed is bundled in a big virtual machine (VM) to avoid any possible interference from other (production) network traffic. This big virtual machine is named *outer control machine*, run in a host machine. We recommend the KVM hypervisor [53] in the host to run the outer control machine and the Xen hypervisor [5] for the inner machines (including the SDN controller, switches and end-points). Reyes presents two alternative approaches in which the outer machine

3 https://github.com/mnm-team/sdn-conflicts

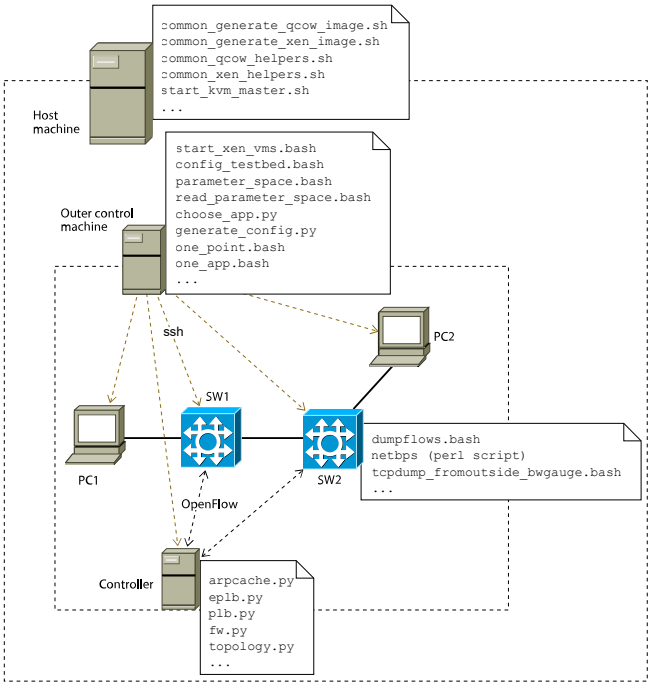


Figure 25: Illustration of a simple test-bed with the associated programs in each class of machines

is created with the VirtualBox hypervisor[4] or directly with Xen [90]. In the latter case, the host machine has Xen installed and plays the role of the outer machine. One can choose a suitable setting depending on her preference and the computing environment being used.

Network topology and specification files

Each test-bed corresponds to a network topology. The creation of a test-bed from a network topology specification is based on the work of Danciu et al. [23], in which the specification files for each machine must be prepared beforehand. Reyes provides a convenient python script to generate these specification files from a simpler specification containing the machines' names and links between them. For the topology in Figure 25, an excerpt of the specification looks like:

```
model.testbed.switches.add("sw1")
model.testbed.switches.add("sw2")
model.testbed.hosts.add("pc1")
model.testbed.hosts.add("pc2")
switch_edges = [["sw1","sw2"]]
```

4 https://www.virtualbox.org/

```
host_edges = [["sw1","pc1"], ["sw2","pc2"]]
edges = host_edges + switch_edges
```

Listing 3.11: An excerpt of the specification of machines and links for generating the test-bed in Figure 25

One can find the sample python script in our published codebase (file *topogen/autogen/sample_topology.py*). The controller is added by default and thus is not specified in the script. Some excerpts of the specification files for *Controller, PC1, SW1* and *outer machine* after executing the python script are shown below.

```
XEN_IMAGE_NAME="controller"
XEN_TEMPLATE_IMAGE="templates/controller/controller.img"
XEN_BRIDGES="(${XEN_IMAGE_NAME}_vif0,br_man)"
XEN_AUTOCONF="${XEN_IMAGE_NAME}_vif0"
```

Listing 3.12: An excerpt of the specification file *xen_controller.spec* for *controller* in the test-bed in Figure 25

```
XEN_IMAGE_NAME="pc1"
XEN_TEMPLATE_IMAGE="templates/pc/pc.img"
XEN_BRIDGES="(${XEN_IMAGE_NAME}_vif0,br_man) (${XEN_IMAGE_NAME}_vif1,br_s1p1)"
XEN_AUTOCONF="${XEN_IMAGE_NAME}_vif0"
```

Listing 3.13: An excerpt of the specification file *xen_pc1.spec* for *PC1* in the test-bed in Figure 25

```
XEN_IMAGE_NAME="sw1"
XEN_TEMPLATE_IMAGE="templates/sw/sw.img"
XEN_BRIDGES="(${XEN_IMAGE_NAME}_vif0,br_man) (${XEN_IMAGE_NAME}_vif1,br_s1p1) (${
    ↪ XEN_IMAGE_NAME}_vif2,br_s1s2)"
XEN_AUTOCONF="${XEN_IMAGE_NAME}_vif0"
```

Listing 3.14: An excerpt of the specification file *xen_sw1.spec* for *SW1* in the test-bed in Figure 25

```
QCOW_IMAGE_NAME="simple_testbed"
QCOW_TEMPLATE_IMAGE="templates/outer.qcow"
XEN_SPEC_LIST="xen_controller.spec,xen_pc1.spec,xen_pc2.spec,xen_sw1.spec,xen_sw2.
    ↪ spec"
```

Listing 3.15: An excerpt of the specification file *qcow.spec* for the outer control machine in the test-bed in Figure 25

We can observe from these specification files that a template image is required for each kind of machine (lines starting with *XEN_TEMPLATE_IMAGE*), we prepared four template images for the controller, hosts, switches and the outer control machine for this purpose. Each template image contains existing programs as shown in Figure 25, they are necessary for configuring the test-bed, for conducting experiments and creating log files to identify anomalies in an automatic manner. All machines are connected to the management bridge *br_man* (line starting with *XEN_BRIDGES*), the outer machine has access to the inner ones via this bridge. Another interface of *PC1* connects to the bridge *br_s1p1*, to which one of the interfaces of *SW1* also connects, thus there is a connection between *PC1* to SW1 via the bridge *br_s1p1*. The other connections are established in the same manner.

Once the specification files for all machines are available, a command is called to create the outer control machine together with all Xen virtual machines inside:

```
bash common_generate_qcow_image.sh qcow.spec testbed.qcow
```

qcow.spec is the specification file for the outer machine shown in Listing 3.15 and *testbed.qcow* is the target *qemu* disk image[5]. This *qemu* disk image can be modified via the *qemu-nbd* and *mount* tools, e. g., to configure the IP address for *ssh* when the machine is brought up.

The test-bed can be started from the host machine with the command:

```
bash start_kvm_master.sh testbed 88
```

in which *testbed* is the name of the qcow image just created (*testbed.qcow*), the number *88* specifies the last byte for the MAC address of this new outer control machine, this number is specific to our own environment to distinguish test-beds created on the same host machine. After the outer machine is brought up, it automatically invokes the scripts *start_xen_vms.bash, config_testbed.bash* (shipped with its template image) to start all the Xen virtual machines inside and to configure the test-bed (e. g., configure IP addresses for these machines, set up OpenFlow switches, bring up their interfaces).

It is to be noted that the main purpose of the above steps is to create and bring up the test-bed, bundled in the *qcow* image of the outer control machine. One can copy and run this *qcow* image in another host machine with KVM hypervisor, or convert it into other formats to run with the preferred hypervisors, e. g., VirtualBox, VMWare Workstation Player[6].

At this point, the test-bed is completely established, one can proceed to perform experiments therein.

Conducting experiments and collecting the dataset

Once the test-bed is up, we can access the outer control machine to adapt the parameter space encoded in the file *parameter_space.bash*. The structure of this file was explained in Section 3.4.2. Then, all experiments corresponding to the encoding in this file can simply be performed by executing the script:

```
bash read_parameter_space.bash
```

The tasks performed by this script are as follows.

1. It parses the file *parameter_space.bash* for necessary information about the subspace encoded therein, e. g., which control applications take part in the experi-

5 https://linux.die.net/man/1/qemu-img

6 https://www.vmware.com/products/workstation-player.html

ments, how many configurations each has, their target switches, how many end-points and switches there are in the test-bed, the end-point communication combination.
2. It prepares the clean environment for experiments by removing the existing log files and terminating any running experiment-related processes in the controller, all switches and end-points.
3. Each control application is first executed alone with each of its configuration. The types of traffic (TCP/UDP) specified in the file *parameter_space.bash* are generated between all combinations of end-points derived from this file. Log files are created in each end-point and switch, which manifest the expected network behaviour. These log files hold the information concerning the throughput of each interface of each switch, the communication state between end-points, and the rule tables of each switch.
4. Control applications are then co-deployed with different combinations of their configurations and priority. Log files created at each end-point and switch during this step are compared with the relevant ones obtained previously in the isolated deployment of each control application, deviations indicating potential conflicts are recorded in the file *conflict.txt*, all settings are written into the file *all_config*.
5. After all experiments derived from the file *parameter_space.bash* are performed, the dataset is updated and the clean environment for subsequent experiment rounds is returned.

All of the above steps are carried out automatically by the script *read_parameter_space.bash*. The number of experiments varies depending on the number of control applications and their configurations specified in *parameter_space.bash*, e. g., there are 76 individual experiments for the encoding in Listing 3.1.

The command to execute the control applications in the controller in steps 3 and 4 is the following:

```
ssh -n controller "sh -c 'ryu-manager --observe-links $applist &' > /dev/null
    ↪ 2>&1"
```

$applist variable contains the list of control applications, e. g., *plb.py, eplb.py*. In Chapter 6, we introduce the conflict detection prototype as a control application, named *detector.py*. The evaluation of the prototype in the test-bed can be automated by simply appending the prototype program *detector.py* in the *$applist* variable in the above command.

Replaying an experiment

One may replay a particular experiment from an experiment round to inspect the occurring problems logged in the dataset. We provide the two scripts *one_app.bash* and *one_point.bash* for this purpose. As mentioned in Section 3.4.6, each experiment round (containing a set of experiments associated with the space specified in the

file *parameter_space.bash*) is assigned a unique ID in the dataset represented by the time point in the format *Year-Month-Day_Hour-Minute-Second* (e. g., 200820_152207) when the round started (see also Listing 3.6). Experiment settings are encoded as points in the dataset, which are used in the files *all_config* and *conflict.txt* (see Listings 3.7 and 3.8). The setting associated with a point can be restored by:

```
bash one_point.bash <experiment round ID> <point number>
```

The command also deploys all control applications in that point together, we can then generate traffic in the data plane, capture packets, inspect switches' rule tables…to examine the reported problems in detail.

In a point, control applications are co-deployed in some combination of their configurations and priority. One may want to examine also how each of them is deployed in isolation to compare the expected and observed network behaviour and derive conflicts. The script *one_app.bash* is useful in this case:

```
bash one_app.bash <app_name> <config> <experiment round ID>
```

Testing a new control application

A new control application can be added to the test-bed for experimenting. The following steps need to be fulfilled.

- Its *python* program (we use the ryu SDN framework implemented in the *python* programming language as the controller) needs to be copied to the controller machine at the location specified by the *APPDIR* variable of the file *collectdata_config.bash* in the outer machine (which is */root/massive/* by default), its local configurations in the files named *appname_config_local_<i>* where $i \geq 1$ must also be placed there.
- The file *parameter_space.bash* needs to be updated with the presence of this new control application, its number of configurations and target switches.
- The file *read_parameter_space.bash* needs to be updated with the comparison between the expected and observed network behaviour to detect symptoms of conflicts and report them in the log files. The comparison implemented for the existing control applications (e. g., End-point Load Balancer) in this file can be used as a guideline.

Then, experiments associated with the adapted parameter space containing the new control application can be carried out by invoking the script:

```
bash read_parameter_space.bash
```

3.5 SDN control applications

Control applications play a crucial part in our conflict researching. We observe that some properties of control applications may affect the way conflicts happen. We describe these properties in this section, which are referred here and there later in this book. A set of control applications follows.

3.5.1 Properties of SDN control applications

A control application can be deployed within the controller or separate, it can install rules in an active or passive manner, and can maintain its state reflecting what it has done.

Controller built-in vs. separate
SDN control applications can be implemented as controller built-in or they can be separated from the controller and use the controller northbound interface (or Application Programming Interface (API)) to control the network. The common controller northbound APIs are Representational State Transfer (REST) APIs [36], applications using REST APIs are referred to in this work as REST-based.

Controller built-in applications are quicker in response to an event or in installing rules in the data plane while those separate from the controller induce more latency and their responsiveness is highly dependent on the network state. Due to the high latency in communicating with the controller, implementing separate applications after the event/method model mentioned in Chapter 1 is not recommended though it is possible by employing certain technique, e. g., Websocket [34] to maintain the two-direction connection between the controller and a separate application and to avoid the expensive polling problem pointed out in [64]. However, separate applications can support the user/administrator better in interacting flexibly and remotely with the controller to actively control the network. Besides, they can be stopped or started independently from the controller whereas the controller built-in applications in some cases or in specific implementation can only be started or stopped together with the controller.

Regarding our experiments' setup, controller built-in applications can well replace the separate ones for the same goals while providing higher level of reliability. Therefore, most of our chosen control applications for experiments are controller built-in ones.

Active vs. passive
A control application is classified as *active* if it behaves based on monitoring information from the data plane that it actively detects (e. g., via symmetric request/response messages using the controller's method/event mechanism) or if its behaviour is independent from events from the controller, e. g., it monitors the network and intercepts if necessary. A *passive* application only reacts upon receiving events generated by the controller.

Intuitively, active applications can better avoid conflicts than passive ones since they deploy rules only when necessary based on their analysis of the current data plane's situation. For studying conflicts, different combinations of both active and passive applications are of interest.

Stateful vs. stateless (with reference to packet-in events)
There are situations that a switch receives at a time an avalanche of packets of the same packet flow (i. e., having the same OSI L2–4 header fields) that it has to ask the controller for instructions on forwarding; the switch sends many packets of this flow to the controller which then generates a mass of packet-in events to control applications. Control applications that treat packet-in events in a stateless manner may result in network behaviour different from those stateful. Take, for example, a stateless End-point Load Balancer (EPLB) application that balances incoming requests to a virtual server in a round-robin fashion on different target replicas, this EPLB receives multiple packet-in events of the same packet flow at a time, it generates corresponding rules to each of them: rules directing them to the first replica, then rules to the second replica and so on until it has handled all packet-in events of that flow. The resulting effect, which is generally not expected, is that rules of the same match fields and different actions are installed repeatedly and then immediately replaced/overridden by another rules. In general, this is not the behaviour that the application developer desires and appears to be likely a bug. The common practice is to handle packets of the same flow consistently and the control application should be stateful in terms of handling packet-in events, i. e., it should log the packet-in events that it has handled already and ignore the subsequent packet-in events having the same concerning packet headers. In our experiments, the control applications are always stateful in handling packet-in events.

3.5.2 Control applications for experiments

We describe the control applications that we employ for our experiments. Most of them rely on the fundamental functions implemented as controller built-in including topology discovery, ARP cache and NDP cache.

To avoid confusion, it is to be noted that in the SDN paradigm, a control application can place its rules in any target switch, these rules can be assigned with predefined priority while matching packets coming to the switch containing them. For passive control applications, we also mention their target traffic in their description, being the content of packet-in events to be handled by them.

Table 3.4 outlines the control applications that we employed according to their characteristics. Since they are always stateful in handling packet-in events, we do not mention this feature in the table.

Fundamental functions

Topology discovery: to discover ports, links and switches, their addition or removal. If a port, a link or a switch is removed, the existing rules related to it will also be deleted in the corresponding network devices. The controller sends Link Layer Discovery Protocol messages (LLDP) [19] regularly to probe the aliveness of each switch, port or link and updates the network topology accordingly (Volkan Yazici provides a good explanation of the network discovery mechanism in SDN in his blog[7]). We encode the network topology as a directed graph with self loops and parallel edges, which facilitates the other network functions, such as routing, load balancing.

Apps	Active	Passive	Controller built-in	REST-based	Target traffic
Topology Discovery	✓		✓		LLDP
ARP cache	✓	✓	✓		ARP
NDP cache	✓	✓	✓		IPv6, ICMPv6
SPF		✓	✓		ARP, ICMP, TCP, UDP
EpLB		✓	✓		TCP, UDP
PLB	✓		✓		TCP, UDP
PPLB4S		✓	✓		TCP, UDP
PPLB4D		✓	✓		TCP, UDP
Firewall	✓		✓	✓	TCP, UDP
TE	✓	✓	✓	✓	TCP, UDP
PE		✓	✓		TCP, UDP
pHS		✓	✓		TCP, UDP
aHS	✓		✓		TCP, UDP
MEADcast		✓	✓		MEADcast traffic (UDP over IPv6)

Table 3.4: Classification of control applications. SPF: Shortest Path First, EpLB: End-point Load Balancer, PLB: Path Load Balancer, PPLB4S: Source-based Passive Path Load Balancer, PPLB4D: Destination-based Passive Path Load Balancer, TE: Traffic Engineering, PE: Path Enforcer, pHS: Passive Host Shadowing, aHS: active Host Shadowing.

ARP cache: to learn and cache the MAC and IPv4 address mapping based on the *ARP REQUEST* and *ARP REPLY* messages of end-points. Its operating mechanism is similar to the address resolution proxy found in [60]. The ARP cache helps eliminate the need of the spanning tree protocol in network topologies containing loops and reduce undesirable traffic noise, e. g., broadcast traffic caused by ARP REQUEST/REPLY while examining conflicts.

NDP cache: to learn and cache the MAC and IPv6 address mapping of end-points based on the *Neighbor Solicitation* and *Neighbor Advertisement* messages as specified in the Neighbor Discovery Protocol (NDP) [75].

7 https://vlkan.com/blog/post/2013/08/06/sdn-discovery/

Shortest Path First Routing (SPF)
Target traffic: ARP/ICMP/TCP/UDP.
This controller built-in application provides the basic routing function for ARP, ICMP and TCP/UDP traffic based on the shortest path first algorithm. It takes into account the *in_port* of the packet coming into a switch and route the packet out of that switch on the port other than the *in_port* to avoid traffic loop. SPF extends the fundamental functions (topology discovery and ARP cache) and reacts *passively* upon receiving packet-in events. It can be executed as one of the main control applications for examining conflicts or as a background one for the others. In the latter case, SPF does not deploy rules interfering the control applications that it supports.
SPF targets all switches in the network.

End-point Load Balancer (EpLB)
Target traffic: TCP/UDP traffic from a given IP/MAC address to be used as virtual server IP and a list of replicas IP/MAC addresses.
The Session-based End-point Load Balancer (SBEpLB or EpLB) is realized as a controller built-in application. It is interested in only packet-in events for TCP/UDP traffic. Each session corresponds to a tuple of *<IP source, IP destination, TCP or UDP protocol, TCP/UDP source port and destination port>*. When a client establishes connection to a proxy server, EpLB will direct this connection to one of the replicas in a round-robin manner. Once a replica is assigned for the connection from the client, this TCP/UDP session will be maintained until it ends, which means a half-done session will never be assigned to another replica.
After choosing the replica for a new session, EpLB consults the global network topology (provided by the fundamental functions) to decide the output port for the current session to reach the replica. To some extent, it performs the routing function for TCP/UDP traffic on its responsible network devices (the balancing points).

Path Load Balancing (PLB)
The controller built-in application PLB balances traffic over multiple paths to reduce/avoid congestion in the network. PLB actively collects and analyzes regularly statistic data of its balancing points (one or a couple of switches) to calculate the throughput of ports/flows and shifts some traffic flows on lower-load paths. It uses the fundamental functions (ARP cache and topology discovery) to choose the paths for balancing.

Source-based Passive Path Load Balancer (PPLB4S)
Target traffic: TCP/UDP traffic from a given list of IP/MAC addresses (to be used as traffic source).

A use case for PPLB4S: a set of servers are broadcasting some live services and there are multiple clients connecting to these servers to download the content. There may be heavy traffic on the direction from servers to clients. Hence, traffic originating

from these servers, upon reaching the balancing points, will be directed on different paths starting from the balancing point, if possible, to the clients. This helps reduce the possible high load on a particular path.

PPLB4S functions as a passive controller built-in application and installs path flows from balancing points to destinations, i. e., it installs rules on all switches along the path that it has determined.

Destination-based Passive Path Load Balancer (PPLB4D)
This application is similar to the above PPLB4S, but as the name indicates, it bases itself on the destination addresses.

Target traffic: TCP/UDP traffic from a given list of IP/MAC addresses (to be used as traffic destination).

PPLB4D functions as a passive controller built-in application and installs path flows from balancing points to destinations, i. e., it installs rules on all switches along the path that it has determined.

Firewall (FW)
This control application is implemented in both controller built-in and REST-based versions. Common firewall behaviour is specified in [39], it is expressed typically as a predefined static set of rules to be deployed and does not reflect dynamic reactions to network states. We extend the basic firewall functions with the capability of actively monitoring its target switches and carrying out certain operations. Specifically, the firewall periodically monitors the target switches based on their flows' and ports' information. The top-load flow of a port will be dropped temporarily if the port throughput surpasses a predefined threshold.

Traffic Engineering (TE)
This control application is implemented in both controller built-in and REST-based versions. It directs traffic on specific paths according to predefined policies. It can behave in either active or passive manner. Traffic engineering practice related to traffic shaping [9], traffic throttling or rate limiting has not yet realized due to the limited support for OpenFlow's *meter* feature by Open vSwitch (our SDN test-bed is built with Open vSwitch version 2.6.2).
Path enforcer (PE) implemented by Reyes [90], is a specific case of the Traffic Engineering application. This controller built-in passive application installs rules to direct traffic on chosen paths, for example, some traffic class needs to be forwarded on a more reliable path than the default shortest path.

Passive Host Shadowing (pHS)
Passive Host Shadowing, implemented by Reyes [90] as a controller built-in passive control application, adds rules to redirect traffic to an alternative destination by modifying the corresponding fields of packets. It is useful in ensuring the ser-

vice availability when a server needs to be maintained and a shadowed server takes responsibility. The exertion of the shadowed server allows the service continuity without any change from end users or DNS solutions.

Active Host Shadowing (aHS)
We implement a variant of Reyes' Host Shadowing control application that behaves in an active manner, named active Host Shadowing. It is a controller built-in application that places its rules in the network without passively reacting upon receiving packet-in requests. Its use-cases stay the same as its passive counterpart.

MEADCast – Multicast in SDN
Privacy-Preserving Multicast to Explicit Agnostic Destinations – short for MEADcast [24] – is a protocol that supports the smooth transition from massive unicast to sender-centric multicast to reduce traffic volume in the network while hiding the identity of recipients from each other. MEADcast is deployed as a controller built-in application and reacts to MEADcast-labeled traffic, which includes special UDP messages transmitted over IPv6. Details of its implementation and deployment in SDN is presented by Minh Nguyen in his thesis [76].

Our observation via manual deployment of MEADcast with the other control applications reveals no conflicts due to their non-overlapping concerns: one targeting IPv6 and the others IPv4, therefore MEADcast was not involved in our massive test (Section 3.4).

3.6 Selected experiments illustrating the methodology

We present nine experiments selected from the whole set of our experiments (more than 11,700 individual experiments, see Section 3.3) to illustrate the proposed methodology. They also support the conflict analysis and classification in Chapter 4, and assist our decision in implementing the conflict detection prototype in Chapter 6. The first two show conflicts arising from contradicting rules within a rule table. The results from the third and fourth experiments indicate that conflicts can occur by the combination of rules in different devices. Another type of conflict appears in the fifth and sixth experiments as side-effects of clashing rules in the same rule table, the seventh experiment shows no side-effect but only conflicts featured by main effects of contradicting rules. We illustrate side-effect conflicts in the eighth and ninth experiments caused by rules scattered in different devices.

3.6.1 Experimental environment

Experiments are deployed on the topology named *topo1* shown in Figure 26. The test-bed is built based on virtual machines as described in Section 3.4.1. We employ the Ryu SDN framework [107] for the SDN controller and OpenFlow 1.3 as the controller southbound API. Open vSwitch [84] with OpenFlow support is used for SDN

switches. Traffic among end-points is generated by common tools: *iperf*[8], *netcat*[9] and *ping*.

3.6.2 Applications' configurations for experiments

We employ the control applications Shortest Path First (SPF), End-point Load Balancer (EpLB), Path Load Balancer (PLB), Traffic Engineering (TE), Destination-based Passive Path Load Balancer (PPLB4D) and Firewall (FW) described in Section 3.5. They are executed concurrently in different combinations, depending on each experiment. Their configurations are described in the following.

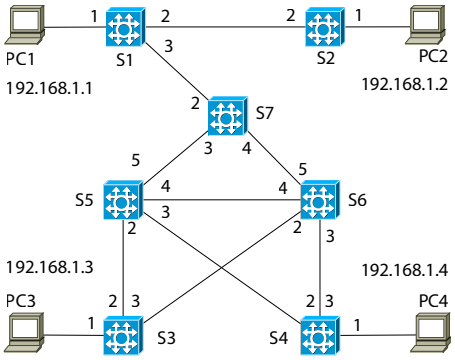


Figure 26: Topology *topo1* for the experiments. The numbers surrounding a switch indicate its port names.

Shortest Path First (SPF)

The SPF application uses the topology information provided by the controller to realize the shortest path first routing function for all common kinds of traffic: ARP, ICMP, TCP, UDP. It is configured to deploy rules in two manners for IP traffic:

- *SPF1:* the rule's match fields include: source IP address, destination IP address, IP protocol number[10]
- *SPF2:* the rule's match fields include only the destination IP address.

End-point Load Balancer (EpLB)

The session-based end-point load balancer balances TCP/UDP traffic among configurable replicas. To change the target replica transparently to the sender, EpLB modifies specific fields (e. g. destination MAC address, destination IP address) of packets at its target switches. This operation is implemented by installing rules with the *set_field* action in the corresponding OpenFlow SDN devices.

EpLB deploys its rules on switch S7 to balance UDP/TCP sessions between PC3 and PC4. It operates in two configurations:

- *EpLB1:* always forwarding traffic from PC1 to PC4 and from PC2 to PC3.
- *EpLB2:* the first incoming session destined to PC3 will be sent to PC3, the second session to PC3 will be changed to PC4 by rewriting the destination information of the relevant traffic, the third will come to PC3 and so on.

8 https://iperf.fr/
9 https://man.openbsd.org/nc.1
10 https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml

The balancing operation is transparent to end users in that the traffic in response from PC4 to the original source will be changed to appear as if it was sent from PC3.

Path Load Balancer (PLB)
This application is deployed with only one configuration (PLB1) on switch S7. Load on the link S7–S5 and S7–S6 is monitored. If this load exceeds a pre-defined threshold, this application divides the load on different paths to the destination based on the session information in a round-robin manner, otherwise it does nothing. For instance, if the traffic volume on the link S7–S6 is above the threshold while the link S7–S5 is underutilised, the largest flow on this link is directed to the link S7–S5 by rewriting the output port of the corresponding rule(s) in the rule table of switch S7.

Traffic Engineering (TE)
TE is executed in three different manners.

- *TE1:* It directs all TCP and UDP traffic targeting PC4 with the destination port of 5001 on the link S6–S5 by installing a flow entry on switch S6 to direct this traffic out of port 4.
- *TE2:* It redirects all traffic of the same port, destination, e. g. all traffic to the web server on port 80, on a dedicated path which is supposed to be more secure and reliable. In our experiment, all TCP and UDP traffic to PC3 with the destination port being 5001 will be sent through the link S7–S6 by installing a flow entry on switch S7 to direct all traffic of this kind out of its port 4.
- *TE3:* It sends all TCP traffic to PC3 out of port 3 of switch S7 on the link S7–S5 and all TCP traffic to PC4 out of port 4 of switch S7 on the link S7–S6.

Destination-based Passive Path Load Balancing (PPLB4D)
This control application is deployed in two configurations on switch S5.

- *PPLB4D1:* It directs alternately TCP and UDP sessions to PC4 on different paths, e. g., the first session to PC4 is sent on the path S5–S3–S6–S4, the second on S5–S4 and the third on S5–S6–S4.
- *PPLB4D2:* It directs alternately TCP and UDP sessions to PC3 on different paths, e. g., the first session to PC3 is sent on the path S5–S3, the second on S5–S4–S6–S3 and the third on S5–S6–S3.

Firewall (FW)
The Firewall is deployed with one configuration (FW1) on switch S3 to drop all TCP and UDP traffic having the destination port of 5001 targeting PC4.

3.6.3 Experiments

For each experiment, we show its settings as a point in the parameter space established in Chapter 3.2.1, the expected and the observed (real) network behaviour, and the rule tables of the switches that contain rules causing conflicts. Some axes share the common value, specifically, all experiments use the same topology, being *topo1*, PC1 and PC2 communicate with PC3 and PC4 using the constant bit rate traffic profile.

We use the *cookie* value of a rule to determine the control application installing that rule. Information irrelevant to our analysis is removed while displaying rules, including *duration, table, n_packets, n_bytes*. To improve clarity, we annotate the action of each rule governing TCP/UDP traffic in a rule table together with the next device, i. e., a switch or an end-point, that receives packets matched by that rule.

Experiment 1: SPF1 and EpLB1
(Figure 27 and Table 3.5) In this experiment, SPF1 and EpLB1 are executed with the same priority of 1. Only the controller management rules are present in the rule table of switch S7 in the beginning, they include rule 1 used for the topology discovery service and rule 8, being the table-miss flow entry which forwards packets unmatched by other rules in the same rule table to the controller. Rule generation happens on first traffic both for EpLB1 and SPF1. After establishing the communication by *nc* between PC1 and PC3, the rule table of switch 7 contains the rules shown in Table 3.5.

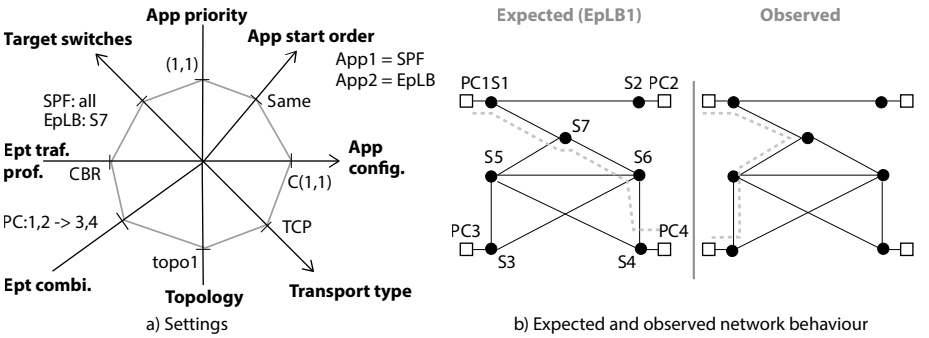


Figure 27: Experiment 1: settings, expected and observed network behaviour

Conflict observation The resulting combination of rules prevents the second application from being effective. Layer 4 traffic from PC1 to PC3 is handled by PC3 instead of PC4, as intended by EpLB.
Conflict analysis The conflict can be identified by comparing rules 6 and 7 highlighted in Table 3.5. From the order of the rules installed, rule 7 of EpLB is never used to match incoming traffic since it is "covered" by rule 6 of the SPF application. According to the OpenFlow 1.3 standard, a packet is matched by only one rule in a rule table and the subsequent ones will be ignored.

#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	Action
1	Ctrl	65535	–	01:80:c2:00:00:00	LLDP				Ctrl
2	SPF1	1	00:16:3e:11:11:01	00:16:3e:11:11:03	ARP				output:3
3	SPF1	1	00:16:3e:11:11:03	00:16:3e:11:11:01	ARP				output:2
4	SPF1	1	00:16:3e:11:11:04	00:16:3e:11:11:01	ARP				output:2
5	SPF1	1	00:16:3e:11:11:01	00:16:3e:11:11:04	ARP				output:3
6	SPF1	1	–	–	–	192.168.1.1	192.168.1.3	TCP	output:3 (to S5)
7	EpLB1	1	00:16:3e:11:11:01	00:16:3e:11:11:03	–	192.168.1.1	192.168.1.3	TCP	SF:eth_dst=00:16:3e:11:11:04, SF:ip_dst=192.168.1.4, output:4 (to S6)
8	Ctrl	0							Ctrl

Table 3.5: Experiment 1: switch S7's rule table after the first TCP session. SF stands for *set_field*.

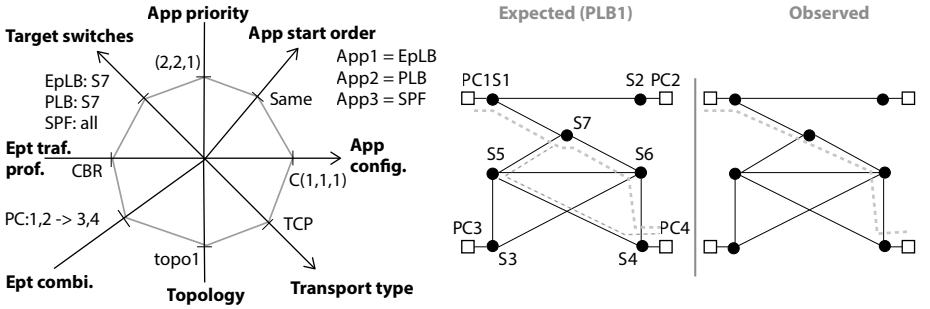


Figure 28: Experiment 2: settings, expected and observed network behaviour

Experiment 2: EpLB1 and PLB1

(Figure 28 and Table 3.6) In this experiment, SPF1 provides the base routing function and has lower priority while EpLB1 and PLB1 run with the same higher priority. The throughput threshold for PLB1 is set to 5 Mb/s. The rule table of switch S7 has only rules 1, 12 (controller management rules) in the beginning. Rule generation happens on first traffic both for EpLB1 and PLB1.

Initially, the network behaviour corresponds to expectation, in that traffic transferred from PC1 to PC3 is handled by PC4. When the throughput observed on the link S7–S6 exceeds the threshold, PLB deploys its rules.

Conflict observation The combination of rules in this experiments prevents PLB from being effective. The flow above threshold still has its traffic being forwarded on the link S7–S6 although PLB intends to move it to the link S7–S5.

Conflict analysis We find that the rules 2 and 4 highlighted in switch 7's rule table (Table 3.6) are at odds, and again the observed effect is a function of rule order.

Experiment 3: PPLB4D1 and TE1

(Figure 29 and Table 3.7) SPF1 provides the basic routing function with the priority of 1, PPLB4D1 installs its rules with the priority of 2 on switch S5, TE1 with the priority of 2 on switch S6. In the beginning, the rule tables of switches S5 and S3 have only the

#	App	Prio	in_port	L2:Src	Dst	Type	L3:Src	Dst	Prot	Action
1	Ctrl	65535		–	01:80:c2:00:00:00	LLDP				Ctrl
2	EpLB1	2	–	00:16:3e:11:11:01	00:16:3e:11:11:03	–	192.168.1.1	192.168.1.3	TCP	set_field:eth_dst=00:16:3e:11:11:04, set_field:ip_dst=192.168.1.4, output:4 (to S6)
3	EpLB1	2	–	00:16:3e:11:11:04	00:16:3e:11:11:01	–	192.168.1.4	192.168.1.1	TCP	set_field:eth_src=00:16:3e:11:11:03, set_field:ip_src=192.168.1.3, output:2 (to S1)
4	PLB1	2	2	00:16:3e:11:11:01	00:16:3e:11:11:03	–	192.168.1.1	192.168.1.3	TCP	set_field:eth_dst=00:16:3e:11:11:04, set_field:ip_dst=192.168.1.4, output:3 (to S5)
5	SPF1	1	–	00:16:3e:11:11:01	00:16:3e:11:11:03	ARP				output:3
6	SPF1	1	–	00:16:3e:11:11:03	00:16:3e:11:11:01	ARP				output:2
7	SPF1	1	–	00:16:3e:11:11:04	00:16:3e:11:11:01	ARP				output:2
8	SPF1	1	–	00:16:3e:11:11:01	00:16:3e:11:11:04	ARP				output:3
9	SPF1	1	–	–	–	–	192.168.1.1	192.168.1.3	TCP	output:3 (to S5)
10	SPF1	1	–	–	–	–	192.168.1.4	192.168.1.1	TCP	output:2 (to S1)
11	SPF1	1	–	–	–	–	192.168.1.1	192.168.1.4	TCP	output:3 (to S5)
12	Ctrl	0								Ctrl

Table 3.6: Experiment 2: switch S7's rule table

	#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action	
	1	Ctrl	65535	–	01:80:c2:00:00:00	LLDP						Ctrl	
Switch	2	PPLB4D1	2	–	–	–	192.168.1.1	192.168.1.4	TCP	48350	5001	output:2	(to S3)
S5	3	SPF1	1	00:16:3e:11:11:01	00:16:3e:11:11:04	ARP						output:3	
	4	SPF1	1	00:16:3e:11:11:04	00:16:3e:11:11:01	ARP						output:5	
	5	Ctrl	0									Ctrl	

	#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action	
Switch	1	Ctrl	65535	–	01:80:c2:00:00:00	LLDP						Ctrl	
S3	2	SPF1	1	–	–	–	192.168.1.1	192.168.1.4	TCP	–	–	output:3	(to S6)
	3	Ctrl	0									Ctrl	

	#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action	
Switch	1	Ctrl	65535	–	01:80:c2:00:00:00	LLDP						Ctrl	
S6	2	TE1	2	–	–	–	–	192.168.1.4	TCP	–	5001	output:4	(to S5)
	3	TE1	2	–	–	–	–	192.168.1.4	UDP	–	5001	output:4	(to S5)
	4	Ctrl	0									Ctrl	

Table 3.7: Experiment 3: rule tables of switches S5, S3, S6

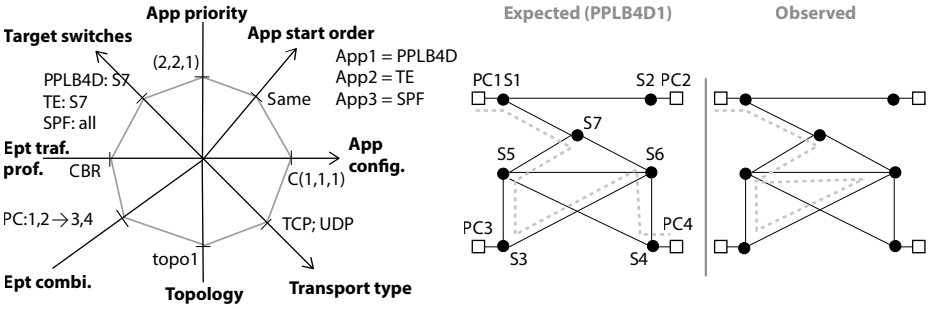


Figure 29: Experiment 3: settings, expected and observed network behaviour

first and the last rules (controller management rules), while all rules in the rule table of switch S6 are populated in the meantime. Other rules are installed in switches S5 and S3 on the first TCP traffic from PC1 to PC4.

Conflict observation PC1 and PC2 fail to communicate with PC4 on some TCP and UDP connections as their traffic is caught in a loop among switches S5–S3–S6–S5.

Conflict analysis We analyse the rule tables of these switches and see that the effective rules for the generated traffic forward it in a loop. These rules are highlighted in Table 3.7. Specifically, rule 2 in switch S5 installed by PPLB4D sends the traffic to switch S3, where rule 2 of SPF directs it to switch S6, rule 2 installed by TE in switch S6 forwards it back to switch S5, the traffic is handled again by rule 2 in switch S5, making a traffic loop.

Experiment 4: PPLB4D1 and FW1

(Figure 30 and Table 3.8) We also use SPF1 to provide the routing function with the priority of 1, lower than that of the two main control applications in test: PPLB4D1 and FW1. In the beginning, switch S5 has only rules 1 and 9 (controller management rules), while the rule table of switch S3 is completely populated with its four rules. Other rules are installed in switch S5 when PC1 and PC2 communicate with PC4 respectively.

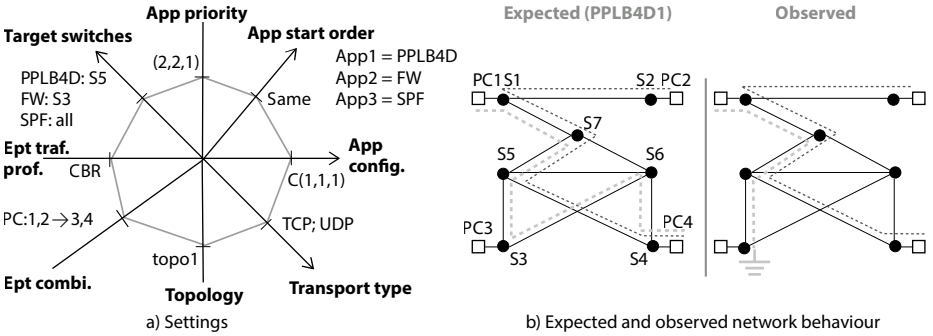


Figure 30: Experiment 4: settings, expected and observed network behaviour

	#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action	
	1	Ctrl	65535	–	01:80:c2:00:00:00	LLDP						Ctrl	
	2	PPLB4D1	2	–	–	–	192.168.1.1	192.168.1.4	TCP	48354	5001	output:2	(to S3)
	3	PPLB4D1	2	–	–	–	192.168.1.2	192.168.1.4	TCP	51094	5001	output:3	(to S4)
Switch	4	SPF1	1	00:16:3e:11:11:01	00:16:3e:11:11:04	ARP						output:3	
S5	5	SPF1	1	00:16:3e:11:11:04	00:16:3e:11:11:01	ARP						output:5	
	6	SPF1	1	00:16:3e:11:11:04	00:16:3e:11:11:02	ARP						output:5	
	7	SPF1	1	00:16:3e:11:11:02	00:16:3e:11:11:04	ARP						output:3	
	8	SPF1	1	–	–	–	192.168.1.4	192.168.1.2	TCP	–	–	output:5	(to S7)
	9	Ctrl	0									Ctrl	

	#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action
Switch	1	Ctrl	65535	–	01:80:c2:00:00:00	LLDP						Ctrl
S3	2	FW1	2	–	–	–	–	192.168.1.4	TCP	–	5001	drop
	3	FW1	2	–	–	–	–	192.168.1.4	UDP	–	5001	drop
	4	Ctrl	0									Ctrl

Table 3.8: Experiment 4: rule tables of switches S5, S3

Conflict observation Some TCP and UDP sessions starting from PC1 and PC2 to PC4 fail as their traffic gets dropped at switch S3.

Conflict analysis We find that the combination of rules in switches S5 and S3 that handled the matched TCP or UDP traffic involves in causing the issue. A case is observed from Table 3.8: rule 2 in switch S5 installed by PPLB4D sends its matched traffic to switch S3, the traffic is dropped by rule 2 there.

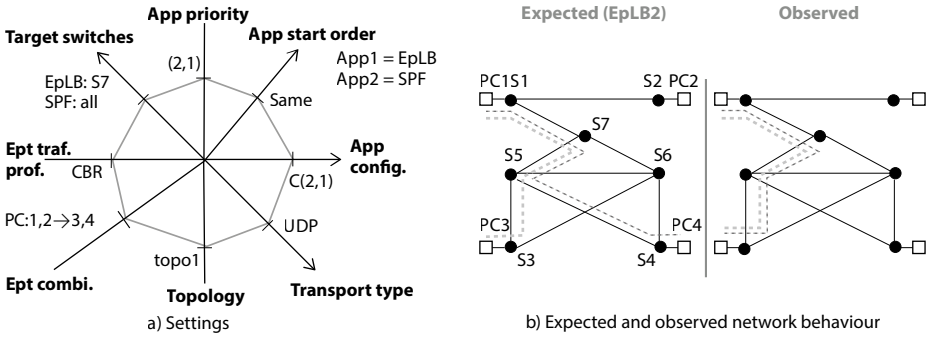


Figure 31: Experiment 5: settings, expected and observed network behaviour

#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action
1	Ctrl	65535	–	01:80:c2:00:00:00	LLDP						Ctrl
2	EpLB2	2	–	–	–	192.168.1.1	192.168.1.3	UDP	48834	5001	output:3 (to S5)
3	EpLB2	2	–	–	–	192.168.1.3	192.168.1.1	UDP	5001	48834	output:2 (to S1)
4	SPF1	1	00:16:3e:11:11:01	00:16:3e:11:11:03	ARP						output:3
5	SPF1	1	00:16:3e:11:11:03	00:16:3e:11:11:01	ARP						output:2
6	SPF1	1	–	–	–	192.168.1.1	192.168.1.3	UDP	–	–	output:3 (to S5)
7	Ctrl	0									Ctrl

Table 3.9: Experiment 5: switch S7's rule table after the first UDP session

Experiment 5: EpLB2 and SPF1

(Figure 31 and Table 3.9) In this experiment, PC1 and PC2 send traffic to PC3, PC4 acts as a replica of PC3. The rule table of switch S7 has only rules 1, 7 (controller management rules) in the beginning. Rule generation happens on first traffic both for EpLB2 and SPF1.

Conflict observation Further UDP sessions from PC1 to PC3 can not be balanced as expected. All of the next UDP sessions from PC1 always come to PC3 while they were meant to be alternately handled by PC3 and PC4.

Conflict analysis The problem can be identified by comparing rules 2 and 6 highlighted in Table 3.9 and analysing the operation mechanism of the control applications. EpLB features a UDP session by additional information of layer 4 source and destination ports as reflected in rule 2. It is supposed to install new rules to handle further UDP traffic from PC1 to PC3 having different combination of layer 4 source-destination ports when being triggered by the corresponding packet-in events for this kind of traffic from the controller. However, since rule 6 matched the mentioned

incoming traffic already, no packet-in event is generated. We can conclude in this case that the installation of rules in the data plane exposes two effects: i) the main effect observed in the data plane with the respective conflict between rules, and ii) the side-effect influencing the control plane's mechanics and thus cancelling the involved control application's intention. Specifically, the presence of rule 6 has the side-effect of suppressing the packet-in events required by EpLB for its proper function.

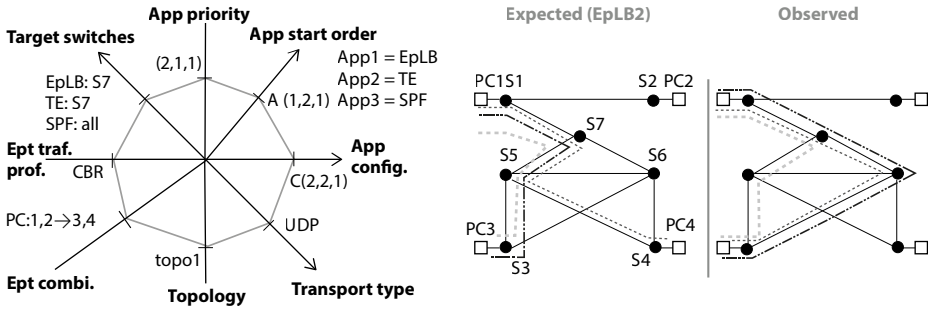


Figure 32: Experiment 6: settings, expected and observed network behaviour

#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action
1	Ctrl	65535	–	01:80:c2:00:00:00	LLDP						Ctrl
2	EpLB2	2	–	–	–	192.168.1.1	192.168.1.3	UDP	38643	5001	output:3 (to S5)
3	EpLB2	2	–	–	–	192.168.1.3	192.168.1.1	UDP	5001	38643	output:2 (to S1)
4	SPF1	1	00:16:3e:11:11:01	00:16:3e:11:11:03	ARP						output:3
5	SPF1	1	00:16:3e:11:11:03	00:16:3e:11:11:01	ARP						output:2
6	TE2	1	–	–	–	–	192.168.1.3	TCP	–	5001	output:4 (to S6)
7	TE2	1	–	–	–	–	192.168.1.3	UDP	–	5001	output:4 (to S6)
8	Ctrl	0									Ctrl

Table 3.10: Experiment 6: switch S7's rule table after the first UDP session and deploying TE1's rules

Experiment 6: EpLB2 and TE2

(Figure 32 and Table 3.10) SPF1 is modified to work in concert with EpLB2 and TE2 in this experiment, its rules will be overwritten or not deployed at all where EpLB2's or TE1's rules are active. EpLB2 balances sessions between PC3 and PC4 where PC4 acts as a replica of PC3. TE2 installs static rules on switch S7 to direct all TCP and UDP traffic having the specified destination port (5001 in this case) to PC3 out of port 4 of switch S7. The rule table of switch S7 has only rules 1, 8 (controller management rules) in the beginning of the experiment. Rule generation happens on first traffic both for EpLB2 and SPF1. In the role of an administrator, we install TE2 rules later via REST API. This experiment shows the importance of the application deployment order.

Conflict observation Similar to the previous experiment, EpLB2 is completely disabled for subsequent UDP sessions having the destination port of 5001 after the TE2' rules become effective.

Conflict analysis Rules 2 and 7 are identified to be responsible for the problem and are highlighted in Table 3.10. Since rule 7 is more general in that it matches only the destination IP address and the destination UDP port, further UDP sessions with these fields will be handled by this rule and no packet-in event is generated that would otherwise ensure the proper function of EpLB2. Again, the presence of rule 7 has the side-effect inducing this conflict.

#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action
1	Ctrl	65535	–	01:80:c2:00:00:00	LLDP						Ctrl
2	TE3	2	–	–	–	–	192.168.1.3	TCP	–	–	output:3 (to S5)
3	TE3	2	–	–	–	–	192.168.1.4	TCP	–	–	output:4 (to S6)
4	SPF2	1	00:16:3e:11:11:01	00:16:3e:11:11:03	ARP						output:3
5	SPF2	1	00:16:3e:11:11:03	00:16:3e:11:11:01	ARP						output:2
6	SPF2	1	00:16:3e:11:11:04	00:16:3e:11:11:01	ARP						output:2
7	SPF2	1	00:16:3e:11:11:01	00:16:3e:11:11:04	ARP						output:3
8	SPF2	1	–	–	–	–	192.168.1.3	–	–	–	output:3 (to S5)
9	SPF2	1	–	–	–	–	192.168.1.1	–	–	–	output:2 (to S1)
10	SPF2	1	–	–	–	–	192.168.1.4	–	–	–	output:3 (to S5)
11	Ctrl	0									Ctrl

Table 3.11: Experiment 7: switch S7's rule table after establishing TCP sessions from PC1 to PC3 and PC4 and deploying TE3's rules

Experiment 7: TE3 and SPF2

(Figure 33 and Table 3.11) This experiment shows that side-effects do not happen at all when the application with more specific rules does not operate on the basis of packet-in events.

The rule table of switch S7 has only rules 1, 11 (controller management rules) in the beginning. Rule generation happens on first traffic by SPF2. TE3's rules are installed subsequently.

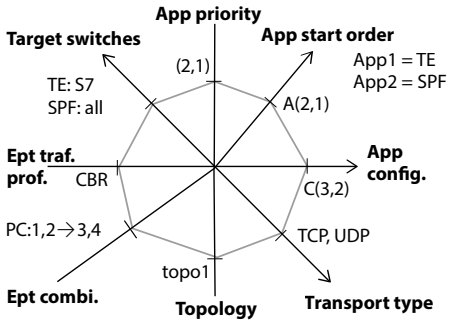


Figure 33: Experiment 7: settings

Observation and analysis Rules 2 and 8 follow the *overlap* conflict pattern (see Chapter 4.1.5), which features a similar relationship between two rules as in the *generalization* pattern but their actions are the same. Rules 3 and 10 exhibit the *generalization* conflict pattern. The network behaves as expected for the main effect and there is no side-effect at all: all TCP traffic to PC3 and PC4 is forwarded according to rules 2 and 3, other traffic, e. g. UDP, ICMP is controlled by SPF2's rules.

Experiment 8: TE2 and PPLB4D2

(Figure 34 and Table 3.12) SPF1 is modified to work in concert with PPLB4D2 and TE2 in this experiment, its rules is overwritten or not deployed at all where PPLB4D2's or

	#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action	
Switch S7	1	Ctrl	65535	–	01:80:c2:00:00:00	LLDP						Ctrl	
	2	TE2	2	–	–	–	–	192.168.1.3	TCP	–	5001	output:4	(to S6)
	3	TE2	2	–	–	–	–	192.168.1.3	UDP	–	5001	output:4	(to S6)
	4	SPF1	1	00:16:3e:11:11:03	00:16:3e:11:11:01	ARP						output:2	
	5	SPF1	1	00:16:3e:11:11:01	00:16:3e:11:11:03	ARP						output:3	
	6	SPF1	1	00:16:3e:11:11:03	00:16:3e:11:11:02	ARP						output:2	
	7	SPF1	1	00:16:3e:11:11:02	00:16:3e:11:11:03	ARP						output:3	
	8	SPF1	1	–	–	–	192.168.1.3	192.168.1.1	TCP	–	–	output:2	(to S1)
	9	SPF1	1	–	–	–	192.168.1.3	192.168.1.2	TCP	–	–	output:2	(to S1)
	10	Ctrl	0									Ctrl	

	#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action	
Switch S6	1	Ctrl	65535	–	01:80:c2:00:00:00	LLDP						Ctrl	
	2	SPF1	1	–	–	–	192.168.1.1	192.168.1.3	TCP	–	–	output:2	(to S3)
	3	SPF1	1	–	–	–	192.168.1.2	192.168.1.3	TCP	–	–	output:2	(to S3)
	4	Ctrl	0									Ctrl	

	#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action	
Switch S3	1	Ctrl	65535	–	01:80:c2:00:00:00	LLDP						Ctrl	
	2	SPF1	1		00:16:3e:11:11:03	ARP						output:1	
	3	SPF1	1	00:16:3e:11:11:03	00:16:3e:11:11:01	ARP						output:2	
	4	SPF1	1	00:16:3e:11:11:03	00:16:3e:11:11:02	ARP						output:2	
	5	SPF1	1	–	–	–	–	192.168.1.3	TCP	–	–	output:1	(to PC3)
	6	SPF1	1	–	–	–	192.168.1.3	192.168.1.1	TCP	–	–	output:2	(to S5)
	7	SPF1	1	–	–	–	192.168.1.3	192.168.1.2	TCP	–	–	output:2	(to S5)
	8	Ctrl	0									Ctrl	

	#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action	
Switch S5	1	Ctrl	65535	–	01:80:c2:00:00:00	LLDP						Ctrl	
	2	SPF1	1	00:16:3e:11:11:03	00:16:3e:11:11:01	ARP						output:5	
	2	SPF1	1	00:16:3e:11:11:01	00:16:3e:11:11:03	ARP						output:2	
	4	SPF1	1	00:16:3e:11:11:03	00:16:3e:11:11:02	ARP						output:5	
	5	SPF1	1	00:16:3e:11:11:02	00:16:3e:11:11:03	ARP						output:2	
	6	SPF1	1	–	–	–	192.168.1.3	192.168.1.1	TCP	–	–	output:5	(to S7)
	7	SPF1	1	–	–	–	192.168.1.3	192.168.1.2	TCP	–	–	output:5	(to S7)
	8	Ctrl	0									Ctrl	

Table 3.12: Experiment 8: rule tables of switches S7, S6, S3, S5 after establishing TCP sessions from PC1 and PC2 to PC3 and deploying TE2's rules

TE2's rules are active. TE2 installs static rules on switch S7 to direct all TCP and UDP traffic having the destination port 5001 destined to PC3 out of port 4 of this switch. PPLB4D2 monitors packet-in events triggered by switch S5 for TCP/UDP sessions destined to PC3, and installs rules to balance these sessions on different paths. The rule table of each switch has only the first and the last rules (controller management rules) in the beginning, except that switch S7 is also populated with TE2's rules. Rule generation happens on TCP traffic from PC1 and PC2 to PC3 for SPF1.

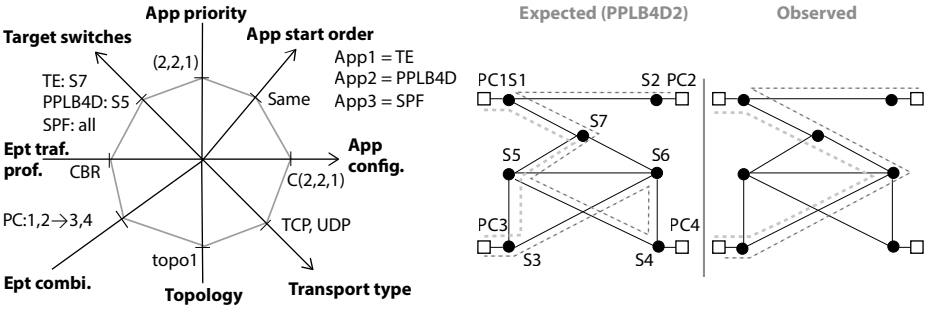


Figure 34: Experiment 8: settings, expected and observed network behaviour

Conflict observation TCP sessions from PC1 and PC2 to PC3 are not balanced by PPLB4D2 in its co-deployment with TE2 as they were in the isolated execution of PPLB4D2.

Conflict analysis These TCP sessions are redirected around the listening point of PPLB4D2, being switch S5, therefore this control application does not receive its interested packet-in events and does not react to balance these sessions. The responsible rules are highlighted in Table 3.12. The mentioned sessions are handled by rule 2 of TE2 at switch S7, which forwards them to switch S6 and they are matched by rules 2 and 3 there, these sessions are then sent to switch S3 and reached PC3 afterwards. As a result, they do not cross switch S5 at all as expected by PPLB4D2.

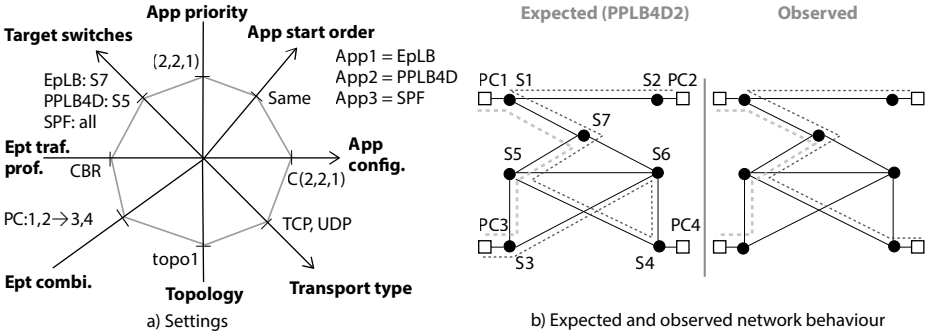


Figure 35: Experiment 9: settings, expected and observed network behaviour

Experiment 9: EpLB2 and PPLB4D2
(Figure 35 and Table 3.13) SPF1 is modified to work in concert with EpLB2 and PPLB4D2 in this experiment, its rules are suppressed where EpLB2's or PPLB4D2's rules are active. EpLB2 registers for packet-in events originated from switch S7 for TCP/UDP sessions destined to PC3. It reacts by installing rules in switch S7 to balance these sessions between PC3 and PC4, the relevant fields in each packet are rewritten (using the OpenFlow's *set_field* function) where necessary. PPLB4D2 monitors packet-in events sent by switch S5 for TCP/UDP sessions destined to PC3, and installs rules to balance these sessions on different paths. The rule table of each switch has only the first and the last rules (controller management rules) in the beginning. Rule generation happens on TCP traffic from PC1 and PC2 to PC3 for all control applications.
Conflict observation Some TCP/UDP sessions from PC1 and PC2 to PC3 are not balanced by PPLB4D2 in its co-deployment with EpLB2 as they were in the isolated execution of PPLB4D2.
Conflict analysis The traffic of these TCP/UDP sessions is modified and the associated packet-in events are not interested by PPLB4D2 anymore. This case is illustrated by the highlighted rules in Table 3.13: the TCP traffic handled by rule 3 in switch S7 is modified to have its new destination as PC4, this traffic is sent to switch S5, PPLB4D2 receives the corresponding packet-in event but is not interested in and ignores it. Consequently, this session is handled by rule 10 in switch S5 installed by SPF1.

3.6.4 Deriving conflict patterns and properties

The classification of conflicts facilitates their handling. Conflicts having common features can be grouped in a class. The common features for some conflicts can be identified by a pattern, while some cannot be portrayed by a pattern but only by their properties. For demonstration purposes, we show how a conflict pattern is extracted from the first and second experiments' results, and discuss the properties of the conflicts occurring in the other experiments. The results from all above experiments are referred in Chapters 4 and 5 for our analysis in reasoning about and classifying conflicts, and in justifying the implementation decision of the conflict detection prototype.

Extraction of conflict patterns
Table 3.14 collects the manually detected contradicting rules from the first and second experiments and generalizes them into more abstract footprints of conflict including *priority*, *match* and *action*. These conflict footprints are referred to as *correlation* in the classification of network security policy conflicts in traditional networks presented in [43].

	#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action
	1	Ctrl	65535	–	01:80:c2:00:00:00	LLDP						Ctrl
	2	EpLB2	2	–	–	–	192.168.1.1	192.168.1.3	TCP	50816	5001	output:3 (to S5)
	3	EpLB2	2	–	–	–	192.168.1.2	192.168.1.3	UDP	53756	5001	set_field:eth_dst=00:16:3e:11:11:04, set_field:ip_dst=192.168.1.4, output:3 (to S5)
SW	4	EpLB2	2	–	–	–	192.168.1.3	192.168.1.1	TCP	5001	50816	output:2 (to S1)
S7	5	EpLB2	2	–	–	–	192.168.1.4	192.168.1.2	UDP	5001	53756	set_field:eth_src=00:16:3e:11:11:03, set_field:ip_src=192.168.1.3, output:2 (to S1)
	6	SPF1	1	00:16:3e:11:11:03	00:16:3e:11:11:01	ARP						output:2
	7	SPF1	1	00:16:3e:11:11:01	00:16:3e:11:11:03	ARP						output:3
	8	SPF1	1	00:16:3e:11:11:02	00:16:3e:11:11:03	ARP						output:3
	9	SPF1	1	00:16:3e:11:11:03	00:16:3e:11:11:02	ARP						output:2
	10	SPF1	1	00:16:3e:11:11:04	00:16:3e:11:11:02	ARP						output:2
	11	SPF1	1	00:16:3e:11:11:02	00:16:3e:11:11:04	ARP						output:3
	12	Ctrl	0									Ctrl

	#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action
	1	Ctrl	65535	–	01:80:c2:00:00:00	LLDP						Ctrl
	2	PPLB4D2	2	–	–	–	192.168.1.1	192.168.1.3	TCP	50816	5001	output:2 (to S3)
	3	SPF1	1	00:16:3e:11:11:03	00:16:3e:11:11:01	ARP						output:5
Switch	4	SPF1	1	00:16:3e:11:11:01	00:16:3e:11:11:03	ARP						output:2
S5	5	SPF1	1	00:16:3e:11:11:02	00:16:3e:11:11:03	ARP						output:2
	6	SPF1	1	00:16:3e:11:11:03	00:16:3e:11:11:02	ARP						output:5
	7	SPF1	1	00:16:3e:11:11:04	00:16:3e:11:11:02	ARP						output:5
	8	SPF1	1	00:16:3e:11:11:02	00:16:3e:11:11:04	ARP						output:3
	9	SPF1	1	–	–	–	192.168.1.3	192.168.1.1	TCP	–	–	output:5 (to S7)
	10	SPF1	1	–	–	–	192.168.1.2	192.168.1.4	TCP	–	–	output:3 (to S4)
	11	SPF1	1	–	–	–	192.168.1.4	192.168.1.2	TCP	–	–	output:5 (to S7)
	12	SPF1	0									Ctrl

Table 3.13: Experiment 9: rule tables of switches S7, S5 after establishing TCP sessions from PC1 and PC2 to PC3

Exp. #	Field	Rule i value	Rule j value	Relation
1	priority	1	1	equality
2		2	2	
1	match	tcp,L3_src=192.168.1.1, L3_dst=192.168.1.3	tcp,L2_src=00:16:3e:11:11:01, L2_dst=00:16:3e:11:11:03, L3_src=192.168.1.1, L3_dst=192.168.1.3	overlap
2		tcp,L2_src=00:16:3e:11:11:01, L2_dst=00:16:3e:11:11:03, L3_src=192.168.1.1, L3_dst=192.168.1.3	tcp,in_port=2, L2_src=00:16:3e:11:11:01, L2_dst=00:16:3e:11:11:03, L3_src=192.168.1.1, L3_dst=192.168.1.3	
1	action	output:3	set_field, output:4	inequality
2		set_field, output:4	set_field, output:3	

Table 3.14: Identifying a conflict pattern by the relations between field values [108]

We observe that within a pair of rules, their priority is equal, the space represented by their match fields has an overlap and their actions differ. Thus the conflict pattern for rules i and j can be formulated as:
$Correlation : priority_i = priority_j, match_i \cap match_j \neq \emptyset, action_i \neq action_j$
In general, we can derive other conflict classes by generalizing the structure of the detected conflict patterns. We apply this method to infer a complete list of local conflicts presented in Chapter 4.1.

Featuring conflicts by properties

The conflicts occurring in experiments 3 and 4 show no pattern of contradicting rules in a single rule table but come from the combination of rules in different devices. We refer to this class of conflicts as *distributed conflicts*. They can be derived from rules that together handle a common packet flow in an unexpected manner, e. g., forwarding it in a loop or dropping it. We provide more details of distributed conflicts in Chapter 4.2.

The effects suggested by the mutual contradicting rules in experiments 5 and 6, as we show in Chapter 4.1, differ from the actual consequence that the victim control application suffers. To be more specific, rules 2 and 6 in Table 3.9 of experiment 5 exhibit the conflict of *overlap* class, whose effect should be harmless, rules 2 and 7 in Table 3.10 expose a *generalization* conflict with the supposed effect to be the general rule (rule 7) would defer to the more specific one (rule 2) for the common traffic. The real effect was that the control applications of higher priority function incorrectly due to the lack of packet-in events. It can be seen from experiment 7 that the same patterns show but this kind of conflict does not occur. Hence, the pattern from these contradicting rules cannot represent this kind of conflicts. In experiments 5 and 6, the conflicts are contingent on the mechanic at the control plane, specifically the packet-in event. This class of conflicts is featured by the mechanic of the control applications that operate in response to this kind of event, not a static conflict pattern. We name

this kind of conflicts as *hidden conflicts* and detail them in Chapter 4.3. Experiments 8 and 9 exhibit other types of hidden conflicts that occur due to the rule combination in different devices.

3.7 Extracting conflict patterns and properties

The determination of a pattern or the unique properties of a conflict type to formulate a conflict class, which is a step in our chosen methodology for studying conflicts (see Section 3.2.2), requires human decisions. To cope with the considerable number of results in the dataset, we combine the extraction of conflict patterns and properties with the development of a conflict detection prototype (see Chapter 6) and its evaluation against the dataset in a repetitive manner as shown in the methodology in Figure 36.

1. We select a set of unanalysed conflict cases logged in the dataset, they correspond to a set of points in a subspace that is reported to induce conflicts, which are recorded in the file *conflict.txt* of an experiment round (see Section 3.4.6).
2. We analyse and extract conflict patterns, properties from these cases manually. If a pattern or a group of properties are observed more than once, we formulate a conflict class therefrom. An example of extracting a conflict pattern is shown in Section 3.6.4. Furthermore, we attempt to derive new conflict classes from the caught patterns, properties if possible. For example, from the pattern of the *correlation* conflict class formulated in Section 3.6.4, which features the relationship between two rules i and j based on the tuple composed of their priority, match and action as:
   $Correlation : priority_i = priority_j, match_i \cap match_j \neq \emptyset, action_i \neq action_j$,
   we analyse the patterns containing this tuple but with different relationship between each of its components, one of the new conflict classes can be formulated as:
   $Redundancy : priority_i \leq priority_j, match_i \subseteq match_j, action_i = action_j$
   The effect associated with each conflict class is drawn, e. g., the effect of the *redundancy* class is harmless. More details on the results of this step are presented in Chapter 4.
   The points associated with these conflict cases in the dataset are marked as "analysed" and will not be processed again in subsequent rounds of the experiment loop. If no new conflict class is found in the chosen set of unanalysed conflict cases, the process repeats from step 1.
3. We build the conflict detection prototype based on the newly discovered conflict classes and the existing ones obtained from literature in the first round of the experiment loop, or integrate the new conflict classes into the prototype from the second round. The concepts, the algorithms to detect conflicts and the implementation of the prototype are elaborated in Chapter 5.

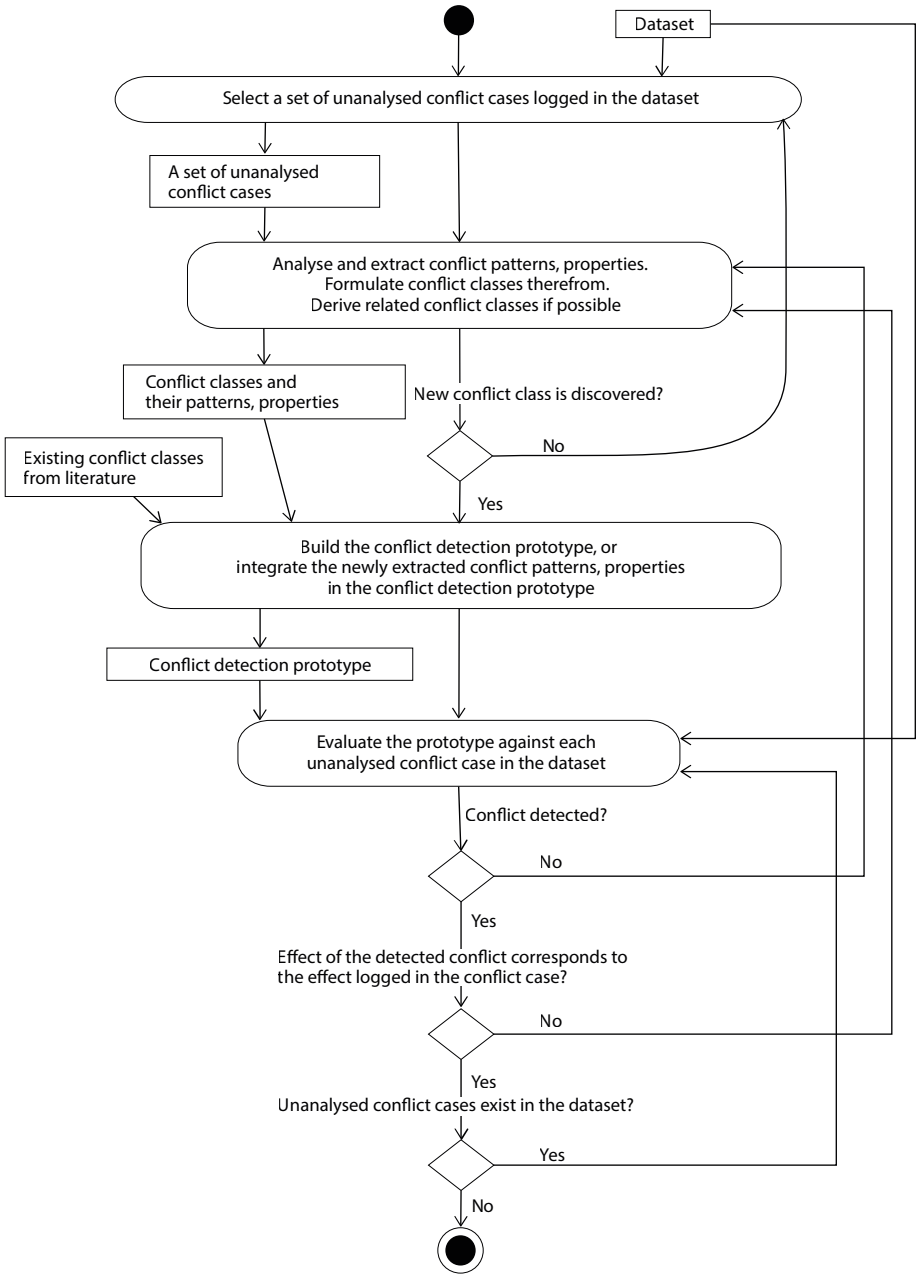


Figure 36: Methodology to extract conflict patterns, properties from the dataset

4. We evaluate the prototype against each unanalysed conflict case in the dataset.

    a) If the prototype detects no conflict, although that case is logged as potential conflicts in the dataset, this indicates the possible presence of a new conflict class. The process is brought back to step 2, where we analyse this case manually to determine if conflicts exist and extract their patterns or properties.
    b) The process is also brought back to step 2 for manual analysis and extraction of conflict patterns, properties, in case the prototype detects conflicts but their effect according to the classification in step 2 does not correspond to the effect of the conflict case logged in the dataset. For instance, a conflict detected by the prototype belongs to the *redundancy* class, whose effect is harmless according to the result in step 2; however, the issue logged in file *conflict.txt* for that case is that the throughput difference of a switch interface was higher than the preset threshold, the effects do not correspond and this case needs to be examined manually. If the log reports unsuccessful communications between end-points while no control application in the co-deployment intends to drop traffic (e. g., firewall), that case should be analysed manually, i. e., the process also repeats at step 2. Due to the large number of such cases, we only select a limited amount therefrom for manual investigation. This incompleteness, however, is offset by the analysis to derive new conflict classes based on the existing ones described in step 2.
    c) Otherwise, the prototype is evaluated against the next unanalysed conflict case in the dataset until all cases have been processed.

The added outcome of this loop is therefore the gradual enhancement of the conflict detection prototype. Besides this incremental evaluation of the prototype, we also evaluate it in designed and random scenarios presented in Chapter 6.

3.8 Conclusion

We have presented an SDN model and applied the formal analytical approach for researching conflicts. Our findings reveal that this approach exposes the inherent shortcomings due to particular situations arising when operating an SDN. We opt for the experimental approach to overcome these limitations, in which we introduce a methodology and a parameter space for experiments. The large experimental space size could not be addressed manually, we implement a framework for automating experiments consequently. The framework facilitates the creation of test-beds, each corresponds to a network topology. It allows the specification of different parameter (sub)spaces, from which experimental settings are derived and experiments are performed automatically. The expected and observed network behaviour are compared to determine potential conflicts based on communication states between end-points, throughput of switches' interfaces and their rule tables. All settings and experiments'

results are logged in a dataset. The framework also supports the reproduction of a single experiment in the dataset, which aids the stepwise inspection of conflicts in that experiment. Our published results[11] consists of 11,772 experiments conducted automatically by this framework, among those conflicts are potential in 2,976 experiments.

We describe a set of control applications and nine concrete experiments to illustrate the proposed methodology as well as the conflicts encountered. These conflicts can manifest within a single switch, or on multiple switches. Some conflicts occur due to the side-effects of rules that influence the SDN's control mechanic. They provide guidelines for the classification of conflicts in the next chapter. The extraction of conflict patterns or properties from the dataset for their classification requires human intervention. We propound a methodology to reduce the manual effort while still maintaining the high accuracy, in which the conflict detection prototype is incrementally improved with the newly identified conflict patterns/properties and then is further applied in the extraction process.

11 https://github.com/mnm-team/sdn-conflicts

4 Conflict Classification

We increment the existing research by a more comprehensive classification of *local conflicts*, and the examination of *distributed conflicts* with more focus on contradicting rules among various control applications. Notably, we discovered a completely new type of anomalies occurring due to side-effects of rule installation, which we name *hidden conflicts*. While the local and distributed conflicts can be determined based on the presence of rules in the data plane, knowledge from the control plane is required for the identification of hidden conflicts. In this chapter, we classify conflicts in SDN and show the patterns of local conflict classes as well as the properties of distributed and hidden conflicts that enable their detection. For hidden conflicts, as a new type, we present in addition our approach to examine and to classify them.

We align the conflict classes in a taxonomy shown in Figure 37. These classes together with their patterns or properties are derived by our analysis of the experiments' results and are complete with reference to the experimental subspaces described in Chapter 3.

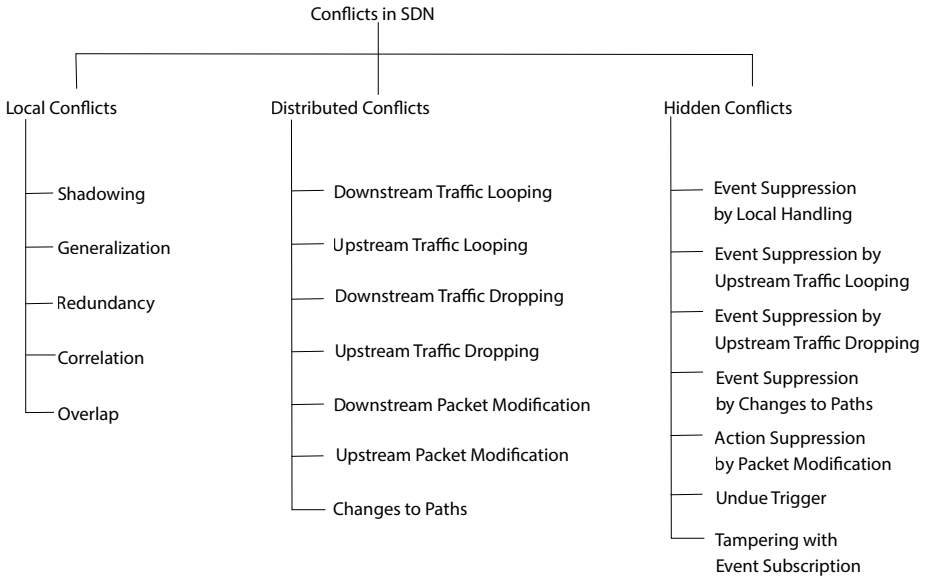


Figure 37: The taxonomy of conflicts in SDN

4.1 Local conflicts

As the name indicates, local conflicts occur between rules within a single device and these rules are installed by different control applications. Among the components of SDN rules, we find that the *priority, match fields* and *actions* involve in shaping this kind of conflicts.

Conflicts are potential only when there exists an overlap in the match fields of two rules. Table 4.1 shows all possible relationship combinations of the *priority, match fields* and *actions* of an SDN rule pair. We do not list the cases of the disjointed *match fields* since they yield no conflict.

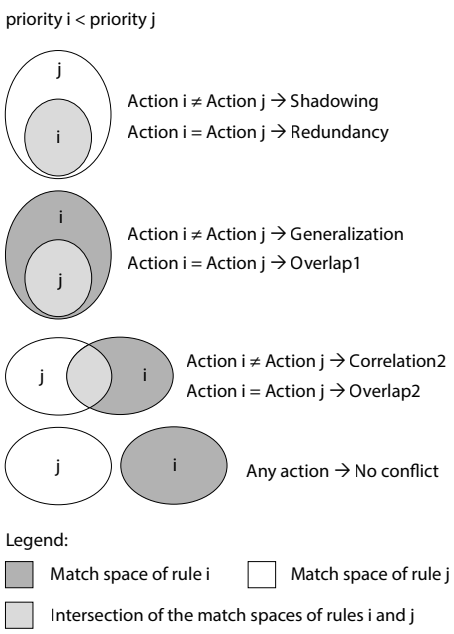


Figure 38: Local conflicts between two rules i and j having different priority in Venn diagram

Depending on the network effect caused by the relationship of these components (priority, match fields and actions) of a rule pair, we are able to categorize local conflicts in different classes: *shadowing, generalization, redundancy, correlation and overlap*. They are also noted in Table 4.1 and visualized in Figures 38 and 39.

Local conflicts have been studied in network security policies by Hamed and Al-Shaer [43], in SDN by Pisharody [86], and are referred to in their work as *intra-policy* conflicts. Our work based on the experimental approach confirms their results and makes improvements by a more comprehensive classification outcome.

Local conflict classes are featured by their patterns. We describe in the following for each class its pattern, the effect and give examples.

4.1.1 Shadowing

Pattern

Rule i is shadowed by rule j if the relationship between the two rules follows the pattern:

$$Shadowing : priority_i < priority_j, match_i \subseteq match_j, action_i \neq action_j$$

Effect

The shadowed rule (rule i) becomes ineffective.

#	pri_i vs pri_j	match_i vs match_j	action	Conflict class and note
1	$pri_i < pri_j$	$m_i \subset m_j$	$a_i \neq a_j$	Shadowing: rule i is shadowed by rule j
2	$pri_i < pri_j$	$m_i \subset m_j$	$a_i = a_j$	Redundancy: rule i is redundant
3	$pri_i < pri_j$	$m_i = m_j$	$a_i \neq a_j$	Shadowing: rule i is shadowed by rule j
4	$pri_i < pri_j$	$m_i = m_j$	$a_i = a_j$	Redundancy: rule i is redundant
5	$pri_i < pri_j$	$m_i \supset m_j$	$a_i \neq a_j$	Generalization: rule i is a generalization of rule j
6	$pri_i < pri_j$	$m_i \supset m_j$	$a_i = a_j$	Overlap
7	$pri_i < pri_j$	$m_i \not\subset m_j \wedge m_i \not\supset m_j \wedge m_i \cap m_j \neq \emptyset$	$a_i \neq a_j$	Correlation: a part of rule i is ineffective
8	$pri_i < pri_j$	$m_i \not\subset m_j \wedge m_i \not\supset m_j \wedge m_i \cap m_j \neq \emptyset$	$a_i = a_j$	Overlap
9	$pri_i = pri_j$	$m_i \subset m_j$	$a_i \neq a_j$	Correlation: **critical**(**)
10	$pri_i = pri_j$	$m_i \subset m_j$	$a_i = a_j$	Redundancy: rule i is redundant
11	$pri_i = pri_j$	$m_i = m_j$	$a_i \neq a_j$	Correlation: (*)
12	$pri_i = pri_j$	$m_i = m_j$	$a_i = a_j$	Redundancy: (*)
13	$pri_i = pri_j$	$m_i \supset m_j$	$a_i \neq a_j$	Correlation: **critical**(**)
14	$pri_i = pri_j$	$m_i \supset m_j$	$a_i = a_j$	Redundancy: rule j is redundant
15	$pri_i = pri_j$	$m_i \not\subset m_j \wedge m_i \not\supset m_j \wedge m_i \cap m_j \neq \emptyset$	$a_i \neq a_j$	Correlation: **critical**(**)
16	$pri_i = pri_j$	$m_i \not\subset m_j \wedge m_i \not\supset m_j \wedge m_i \cap m_j \neq \emptyset$	$a_i = a_j$	Overlap: a part of either rule is ineffective (**)
17	$pri_i > pri_j$	$m_i \subset m_j$	$a_i \neq a_j$	Generalization: rule j is a generalization of rule i
18	$pri_i > pri_j$	$m_i \subset m_j$	$a_i = a_j$	Overlap
19	$pri_i > pri_j$	$m_i = m_j$	$a_i \neq a_j$	Shadowing: rule j is shadowed by rule i
20	$pri_i > pri_j$	$m_i = m_j$	$a_i = a_j$	Redundancy: rule j is redundant
21	$pri_i > pri_j$	$m_i \supset m_j$	$a_i \neq a_j$	Shadowing: rule i shadows rule j
22	$pri_i > pri_j$	$m_i \supset m_j$	$a_i = a_j$	Redundancy: rule j is redundant
23	$pri_i > pri_j$	$m_i \not\subset m_j \wedge m_i \not\supset m_j \wedge m_i \cap m_j \neq \emptyset$	$a_i \neq a_j$	Correlation: a part of rule j is ineffective
24	$pri_i > pri_j$	$m_i \not\subset m_j \wedge m_i \not\supset m_j \wedge m_i \cap m_j \neq \emptyset$	$a_i = a_j$	Overlap: a part of rule j is ineffective

Table 4.1: Local conflict classes of two rules based on their priority, match fields and action. In OpenFlow SDN, the match fields of a rule span the OSI layers 2, 3, 4. The conflict classes noted with (*) in rows 11, 12 do not occur in OpenFlow switch implemented by Open vSwitch in our test-bed (OpenFlow 1.3, OVS 2.6.2). The effect of rows noted with (**) depends on the implementation.

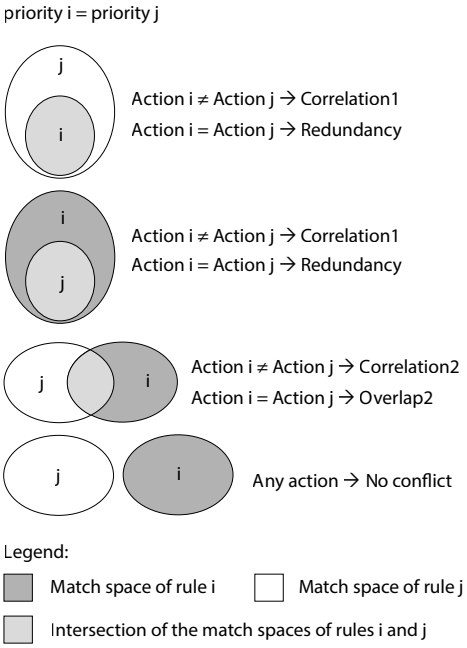


Figure 39: Local conflicts between two rules i and j having the same priority in Venn diagram

Example

```
Rule i: priority=1, match={ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.4, ip_protocol=6,
        tcp_src=48350, tcp_dst=5001}, action=output:2
Rule j: priority=2, match={ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.4, ip_protocol=6},
        action=output:3
```

Rule i with lower priority is more specific than rule j, thus all packets could be matched by rule i are actually matched by rule j. Since they have different actions, these packets are not handled as expected by rule i.

4.1.2 Generalization

Pattern

Rule i is generalized by rule j if the relationship between the two rules exhibits the pattern:

$Generalization : priority_i > priority_j, match_i \subset match_j, action_i \neq action_j$

Effect

A part of the general rule (rule j) becomes ineffective.

Example

Rule i: priority=2, match={ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.4, ip_protocol=6, tcp_src=48350, tcp_dst=5001}, action=output:2
Rule j: priority=1, match={ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.4, ip_protocol=6}, action=output:3

Rule i of higher priority is more specific than rule j, therefore all packets matched by rule i will not be matched by rule j. Because they have different actions, these packets are not handled as expected by rule j.

4.1.3 Redundancy

Pattern
Rule i becomes redundant due to the presence of rule j if the relationship between the two rules exhibits the pattern:

$Redundancy : priority_i \leq priority_j, match_i \subseteq match_j, action_i = action_j$

Effect
The redundant rule is harmless in terms of its influence on the network behaviour. However, it increases the size of a rule table unnecessarily, which can degrade the performance in searching rules, e. g., to match a packet or to compare rules for conflict detection.

Example

Rule i: priority=1, match={ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.4, ip_protocol=6, tcp_src=48350, tcp_dst=5001}, action=output:2
Rule j: priority=2, match={ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.4, ip_protocol=6}, action=output:2

Rule i with lower priority is more specific than rule j, hence rule i is never used to match a packet once rule j is present. However, they have the same action, packets matched by rule j are still handled as expected by rule i.

4.1.4 Correlation

Pattern
Rule i and rule j are correlated if the relationship between the two rules follows the patterns:

$Correlation_1 : priority_i = priority_j, match_i \subseteq match_j \vee match_i \supseteq match_j, action_i \neq action_j$

or

$Correlation_2 : match_i \not\subseteq match_j \wedge match_i \not\supseteq match_j \wedge match_i \cap match_j \neq \emptyset, action_i \neq action_j$

Effect
If the rules have different priority, a part of the lower priority rule becomes ineffective. If they have the same priority, it becomes critical since it is unclear which rule should get effective and which not.

Example
Rules i and j follow the pattern $correlation_2$:

```
Rule i: priority=2, match={ether_type=0x0800, ipv4_src=192.168.1.1}, action=output:2
Rule j: priority=2, match={ether_type=0x0800, ipv4_dst=192.168.1.4}, action=output:3
```

In this case,
$match_i \cap match_j$ = *{ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.4}*
If a packet entering the switch containing these two rules belongs to the flow:

```
ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.4, ip_protocol=6, tcp_dst=5001
```

it is unclear how this packet is handled since it can be matched by either rule i or rule j and their actions are different.

The conflicts between rules 6, 7 in Table 3.5 of experiment 1 and rules 2, 4 in Table 3.6 of experiment 2 in Chapter 3.6.3 exhibit the pattern $correlation_1$.

4.1.5 Overlap

Pattern
Rule i and rule j are overlapped if the relationship between the two rules follows the pattern:
$Overlap_1 : priority_i > priority_j, match_i \subseteq match_j, action_i = action_j$
or
$Overlap_2 : match_i \nsubseteq match_j \wedge match_i \not\supseteq match_j \wedge match_i \cap match_j \neq \emptyset, action_i = action_j$

Effect
The overlap between rules is harmless in terms of their influence on network behaviour. These rules can be subject for optimization, e. g., to reduce the rule table size to obtain higher performance in searching rules. Consider three rules i, j and k in a rule table having the same priority, the same action, and their match spaces intersect each other as shown in Figure 40, apparently, each pair of these rules expose an *overlap* local conflict and none of them is redundant within a pair. However, all traffic matched by rule j can

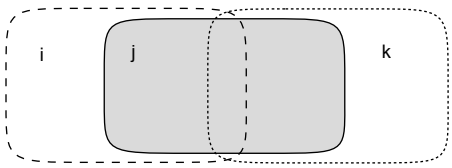


Figure 40: Overlap between the match spaces of rules i, j and k that expose pairwise an *overlap* local conflict, rendering rule j redundant

be handled by either rule i or rule k in the same manner, which means that rule j can be removed from the rule table without any consequence.

Example
Rules i and j follow the pattern $overlap_2$:

```
Rule i: priority=2, match={ether_type=0x0800, ipv4_src=192.168.1.1}, action=output:2
Rule j: priority=2, match={ether_type=0x0800, ipv4_dst=192.168.1.4}, action=output:2
```

In this case,
$match_i \cap match_j$ = *{ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.4}*
As these two rules have the same action, packets matched by either one are handled in the same way, as expected by both.

4.1.6 Discussion

If we consider the timeout of a rule, the relationship of two rules exposing a local conflict is then temporal. For example, consider row 2 in Table 4.1, rule i is made redundant by rule j, if the timeout of rule i is longer than that of rule j, then rule i is redundant only when rule j is still present. Once rule j gets removed, rule i becomes effective if it is not shadowed or made redundant by any other rule. In resolving conflict, it seems worthwhile to take into account the temporal aspects of rules.

Cases 6 and 18 in Table 4.1 are classified as *redundant* in the work of Pisharody [86]. We notice that the effect can be different in the perspective of the whole rule tables containing more than these two rules. If a rule is deemed redundant, it means that this rule can be removed without any issue because the traffic matched by it would be treated in the same way. However, case 6 exposes a different effect. As rule i is broader than rule j (see Table 4.1), there can be traffic matched by rule i but not by rule j. Hence, the removal of rule i could lead to this slice of traffic being handled differently by some rule other than rule j. On the other hand, there might exist some rule k that is shadowed by rule j but is a specific case of rule i, i. e., rule i and k expose a *generalization* conflict where rule k has higher priority. In this case, if rule j is removed, traffic that should be matched by rule i as a result of removing rule j is matched instead by rule k, thus the effect of redundancy is not true. We classify this case therefore as *overlap*, indicating that no rule of the two (rules i and j) is redundant. The same argument applies for case 18 in Table 4.1.

Pisharody [86] introduces *imbrication* conflicts, arising due to rules specifying match fields at different layers and those rules are directly compared without being expressed in the same scale. This class is not incorporated in the local conflicts as all rules are represented in the same format before being checked for conflicts in our work.

4.2 Distributed conflicts

The combined effect of rules in different network devices or in different rule tables can lead to unexpected network behaviour, e. g., traffic looping or traffic dropping. We name this kind of anomalies *distributed conflicts* as opposed to *local conflicts* mentioned in the previous section, which occur due to the contention on handling traffic between rules residing in the same rule table.

Distributed conflicts lead to the suppression of control applications' policies since their rules are not active on the intended traffic as expected. Based on our experiments' results and existing work in literature, we are able to conclude the possible causes of distributed conflicts including: *traffic looping*, *traffic dropping*, *packet modification* and *changes to paths*. From the perspective of a control application, the suppression of its policies can effect in either *downstream* or *upstream* direction of its target network devices.

Cause \ Direction	downstream	upstream
Traffic looping	✓	✓
Traffic dropping	✓	✓
Packet modification	✓	✓
Changes to paths	✗	✓

Table 4.2: Distributed conflicts' causes and directions

The metaphoric meaning of these terms originates from the water flow direction in nature: *downstream* indicates the direction that a river or a water stream flows and *upstream* refers to the reverse direction. Policy suppression is said to take place in the *downstream* direction of a network device if its occurrence is after the involved traffic flow leaving that device. On the contrary, the suppression is in the *upstream* direction of a device if it happens before the traffic flow reaching that device. The involved traffic flow with respect to a control application is the one that is influenced by that application's policies in its isolated deployment, but is not in its co-deployment with other applications. This temporal attribute does impact the conflict detection (see Chapter 5) and thus contributes to the classification of conflicts. The *downstream* direction attribute is not applicable to the *changes to paths* cause, otherwise the involved traffic would have traversed through the target devices of the control application, which contradicts this cause. The cause and direction combination shown in Table 4.2 founds the base for the classification of distributed conflicts.

4.2.1 Policy suppression by downstream traffic looping

The combined effect of rules stemming from different control applications can lead to their matched traffic being forwarded in a loop. A traffic loop occurs among a sequence of rules if there exists a packet handled by them in a loop, i. e., one of these rules appears at least twice in the sequence of rules that the packet traverses.

Property
The rules of a control application take effect on its target traffic at its target network devices. However, its overall policies are not achieved as its target traffic is caught in a loop occurring afterwards.

Effect
The influenced traffic cannot be handled as intended by the mentioned control application but gets stuck in a loop. The effect is more catastrophic if the traffic is replicated at some switch in the loop as the network can quickly be overloaded.

Example
Experiment 3 in Chapter 3.6.3 shows a conflict of this class. We reproduce briefly this experiment here including the conflict effect depicted in Figure 41. If we denote rule i at switch j as $r_i@S_j$, the rule sequence causing traffic looping in this experiment is: $r_2@S_5, r_2@S_3, r_2@S_6, r_2@S_5$, in which $r_2@S_5$ and $r_2@S_3$ are installed by the application Destination-based Path Load Balancer (PPLB4D) and $r_2@S_6$ by the Traffic Engineering (TE) application. The combined effect of the rules from these two control applications causes the traffic sent by PC1 not able to reach PC4 as expected (shown in the left part of Figure 41) but stuck in a loop among three switches S5–S3–S6–S5. The loop occurs on the traffic after (downstream) it left the target switch of PPLB4D (switch S5), suppressing the overall policy of this application reflected in its isolated deployment.

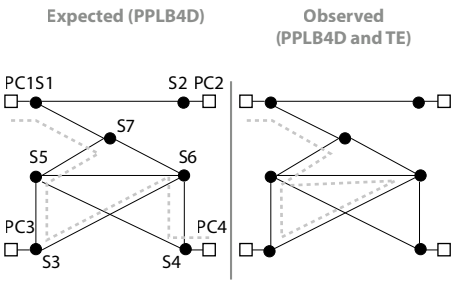


Figure 41: An example of the distributed conflict class *policy suppression by downstream traffic looping*, observed when deploying together the control applications Destination-based Path Load Balancer (PPLB4D) and Traffic Engineering (TE) (reproduced from experiment 3 in Chapter 3.6.3)

$r_2@S_5$ (installed by PPLB4D):

```
priority=2, match={ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.4, ip_protocol=6, tcp_src
    ↪ =48350, tcp_dst=5001}, action=output:2 (send to switch S3)
```

$r_2@S_3$ (installed by PPLB4D):

```
priority=1, match={ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.4, ip_protocol=6},
            action=output:3 (send to switch S6)
```

$r_2@S_6$ (installed by TE):

```
priority=2, match={ether_type=0x0800, ipv4_dst=192.168.1.4, ip_protocol=6, tcp_dst=5001},
            action=output:4 (send to switch S5)
```

From the above rules, we can infer the impacted traffic, which belongs to the flow:

```
ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.4, ip_protocol=6, tcp_src=48350, tcp_dst=5001
```

Discussion
Traffic looping may be caused by mistake of a single control application, which is a bug (see Chapter 2.2), or by different intention of control applications on the same traffic. In any case, its consequence is not expected and the traffic looping needs to be detected and resolved to ensure the stable operation of the network.

We notice an interesting example (not applicable in OpenFlow SDN but possibly achievable in POF or P4 SDN) in which a rule with the action modifying its matched packets can have the traffic escape the rule loop:
$r_1@S_1$ (*output:1* in its action is to send matched traffic to $r_2@S_2$):

```
priority=1, match={ether_type=0x0800, ipv4_src=192.168.1.0/30, ipv4_dst=192.168.1.5, ip_protocol=6,
        tcp_src=48350, tcp_dst=5001}, action=output:1
```

$r_2@S_1$ (*output:2* in its action is to send matched traffic to end-point):

```
priority=1, match={ether_type=0x0800, ipv4_src=192.168.1.4, ipv4_dst=192.168.1.5, ip_protocol=6,
        tcp_src=48350, tcp_dst=5001}, action=output:2
```

$r_2@S_2$ (*output:1* in its action is to send matched traffic to $r_1@S_1$):

```
priority=1, match={ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.4, ip_protocol=6},
        action={increase ipv4_src by 1, output:1}
```

If a packet belonging to the flow

```
ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.5, ip_protocol=6, tcp_src=48350, tcp_dst=5001
```

enters switch S1 and is matched by rule $r_1@S_1$, it is sent to switch S2 and is handled by $r_2@S_2$, which increases its *ipv4_src* by 1 and sends it back to switch S_1; its *ipv4_src* becomes 192.168.1.2, and is sent by $r_1@S_1$ to $r_2@S_2$; after the next two rounds ping-ponged between $r_1@S_1$ and $r_2@S_2$, its *ipv4_src* is now 192.168.1.4 and is not matched by $r_1@S_1$ any more but by $r_2@S_1$, which forwards it out of the loop to the end-point. This example may have a use case in network debugging or used by some kind of attack to increase the network load gradually so that it is not easily detected by common security method. Although the loop in this case occurs with an eventual escape and the effect is less aggressive, its detection for closer examination is still worthwhile.

4.2.2 Policy suppression by upstream traffic looping

The traffic loop can also occur before the traffic could reach the target network devices of a control application.

Property
A control application places its rules in its target network devices; however, its target traffic is caught in a loop and could not reach these devices as expected in its isolated deployment.

Effect
The influenced traffic cannot be handled as intended by the mentioned control application. The effect is more catastrophic if the traffic is replicated at some switch in the loop as the network can quickly be overwhelmed.

Example
A conflict of this class is depicted in Figure 42, observed when co-deploying the control applications Destination-based Path Load Balancer (PPLB4D), Traffic Engineering (TE) and active Host Shadowing (aHS). In its isolated execution, the rules installed by aHS at switch S4 take effect to forward TCP/UDP traffic destined to PC4 to PC3. In the co-deployment of the applications, this traffic gets stuck in a loop due to the same reason as in the previous example (see Figure 41) and could not approach switch S4. Consequently, the loop in the upstream direction of switch S4 suppresses the policies enforced by aHS.

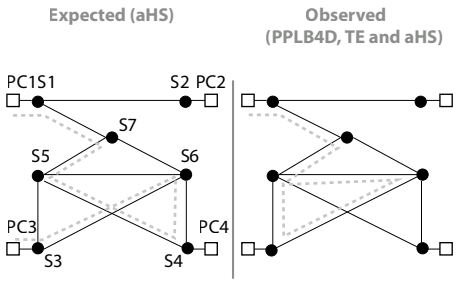


Figure 42: An example of the distributed conflict class *policy suppression by upstream traffic looping*, observed when deploying together the control applications Destination-based Path Load Balancer (PPLB4D), Traffic Engineering (TE) and active Host Shadowing (aHS)

Discussion
The above example shows an extreme case containing conflicts of two distributed conflict classes: policy suppression by both downstream and upstream traffic looping. In general, multiple conflicts of various classes can co-exist when running different applications together. The discussion points in Section 4.2.1 also apply for this conflict class.

4.2.3 Policy suppression by downstream traffic dropping

A rule with a *drop* action, a non-forwarding action or an invalid forwarding action, e. g., forwarding matched packets to a non-existent port of the switch, will cause its matched traffic to be dropped.

Property
The rules of a control application take effect on its target traffic at its target network devices. However, this traffic gets dropped by other applications afterwards.

Effect
The traffic cannot be handled as intended by the affected application, but is dropped at the switch containing the rules dropping traffic. Therefore, the overall policies of the affected application could not be achieved.

Example
A conflict belonging to this class is presented in experiment 4 in Chapter 3.6.3, which is summarized in Figure 43 including the conflict effect. The rule sequence causing the issue in this experiment is: $r_2@S_5, r_2@S_3$; the first rule is deployed by the application Destination-based Path Load Balancer (PPLB4D) and the second by Firewall. As PPLB4D is not aware of the intent of the Firewall application, it balances some traffic sent from PC1 to PC4 on the path S5–S3–S6–S4, which gets dropped by the Firewall's rule at switch S3 and thus could not reach the expected destination PC4. The traffic dropping occurs after (downstream) the traffic left the target switch of PPLB4D (switch S5), suppressing the accomplishment of this application's policies as reflected in its isolated run.

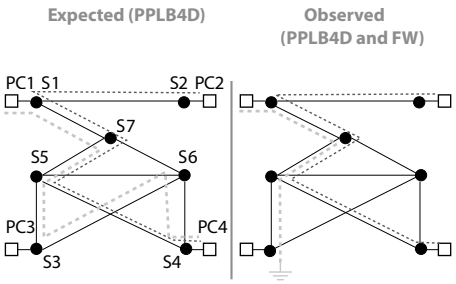


Figure 43: An example of the distributed conflict class *policy suppression by downstream traffic dropping*, observed when deploying together the control applications Destination-based Path Load Balancer (PPLB4D) and Firewall (FW) (reproduced from experiment 4 in Chapter 3.6.3)

$r_2@S_5$:

```
priority=2, match={ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.4, ip_protocol=6, tcp_src
    ↪ =48354, tcp_dst=5001}, action=output:2 (send to switch S3)
```

$r_2@S_3$:

```
priority=1, match={ether_type=0x0800, ipv4_dst=192.168.1.4, ip_protocol=6, tcp_dst=5001}, action=drop
```

From the above rules, we can infer the impacted traffic, which belongs to the flow:

```
ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.4, ip_protocol=6, tcp_src=48354, tcp_dst=5001
```

Discussion
A firewall can be deployed at the network boundary and has predefined rules dropping malicious traffic. This case is usual and should not raise any concern. In detecting conflicts, an option for this exception should be taken into account.
A rule with a non-forwarding action, e. g.,:

```
priority=2, match={ether_type=0x0800, ipv4_src=192.168.1.1, ipv4_dst=192.168.1.3},
                action=set_field:192.168.1.2 -> ip_dst
```

or with an invalid forwarding action (e. g., sending traffic out of a non-existent port) will lead to the same consequence as a rule having a *drop* action. However, it appears more to be a bug since the normal role of a rule should be to keep the traffic flowing or to drop it. Still, the arisen unexpected network behaviour needs to be detected and handled.
Downstream traffic dropping is referred to in Reyes' thesis [90] as *spuriousness*, similar to the term used in another research on security policy conflicts in traditional networks by Hamed and Al-Shaer [43], which is bound to the firewall application. We observe that this conflict class can also occur between control applications other than the firewall and opt for this more neutral term.

4.2.4 Policy suppression by upstream traffic dropping

Traffic can be dropped on its way to the target network devices of a control application.

Property
A control application installs its rules in its target network devices; however, its target traffic gets dropped by other applications and could not reach these devices as expected in its isolated deployment.

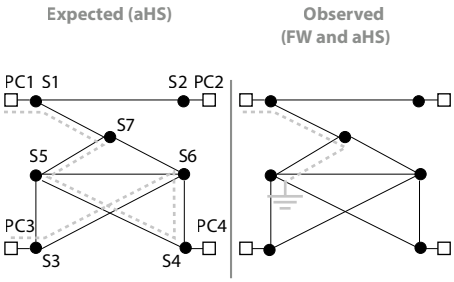


Figure 44: An example of the distributed conflict class *policy suppression by upstream traffic dropping*, observed when deploying together the control applications Firewall (FW) and active Host Shadowing (aHS)

Effect
The influenced traffic cannot be handled as intended by the affected control application.

Example
An instance of this conflict class is observed when co-deploying the control applications Firewall (FW) and active Host Shadowing (aHS), as illustrated in Figure 44. In its isolated execution, the rules installed by aHS at switch S4 take effect to forward TCP/UDP traffic destined to PC4 to

PC3. In the co-deployment of the applications, this traffic gets dropped at switch S5 by FW rules and could not approach switch S4. Consequently, the traffic dropping in the upstream direction of switch S4 suppresses the policies enforced by aHS.

Discussion
The discussion points in Section 4.2.3 also apply for this conflict class.

4.2.5 Policy suppression by downstream packet modification

Network traffic can be influenced by policies from multiple control applications at different points on its path to the destination. There are cases in which an application controls traffic in a manner that suppresses the intent of another application whose policies were enforced earlier.

Property
A control application enforces its policies to its interested traffic at some points in the network, that traffic is later modified by other control applications, which invalidates the overall policies of the first one observed in its isolated deployment.

Effect
The overall policies of the affected control application is not fulfilled, though its rules are properly deployed.

Example
An instance of this conflict class can be observed when deploying the End-point Load Balancer (EpLB) and passive Host Shadowing (pHS) applications as depicted in Figure 45. EpLB is deployed on switch S7 to balance TCP/UDP traffic destined to PC3 over PC3 and PC4, pHS puts its rules on switch S5 to direct all TCP/UDP traffic towards PC4 to PC3. As a result, a part of traffic heading to PC3 that was redirected to PC4 by EpLB at switch S7 is re-modified by pHS at switch S5 to send it to PC3. This downstream modification of traffic with reference to the viewpoint of EpLB annuls partly its overall policies compared to its isolated deployment.

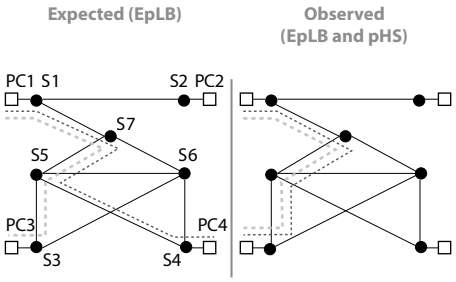


Figure 45: An example of the distributed conflict class *policy suppression by downstream packet modification*, observed when co-deploying the End-point Load Balancer (EpLB) and passive Host Shadowing (pHS) control applications

Discussion
This conflict class is synthesized from the distributed conflict class *general multi-transform* examined by Reyes [90]. A *general multi-transform* conflict occurs if there

exist packets that are modified (transformed) twice or more times by different control applications at different network devices. We use a more neutral name and notice that a conflict of this class can also occur even in case the packets are transformed only once. An example can be inferred from the above example (Figure 45) if the pHS application swaps its front-end and back-end targets, i. e., to directs all TCP/UDP traffic destined to PC3 to PC4. In this case, traffic from PC1 to PC3 is modified only once either at switch S7 by EpLB or at switch S5 by pHS and the conflict still happens.

4.2.6 Policy suppression by upstream packet modification

Network traffic can be altered on its path to the destination. There are cases in which a control application modifies the traffic at one point, leading to unavailable traffic that would otherwise be influenced by another control application at subsequent points in the path to the destination.

Property
Traffic interested by a control application, before reaching its target network devices, was modified by other control applications once or more times. Consequently, the modified traffic is not matched by rules from that application at its target network devices anymore and its overall policies observed in the isolated deployment are not achieved.

Effect
The policies of the control application does not take effect, as its rules do not match the modified traffic.

Example
Figure 46 shows an example of this conflict class. The Firewall control application (FW) is deployed on switch S5 to drop all traffic sent from PC1 to PC3. The End-point Load Balancer (EpLB) installs its rules on switch S7 to balance TCP/UDP traffic destined to PC3 over PC3 and PC4. In the co-deployment of these two applications, some traffic to PC3 is modified at switch S7 by changing its destination to PC4; this traffic then arrives at switch S5 and is not matched by Firewall's rules anymore, but get forwarded to PC4. Hence, from the Firewall application's perspective, its policies are suppressed due to the upstream modification of the packets.

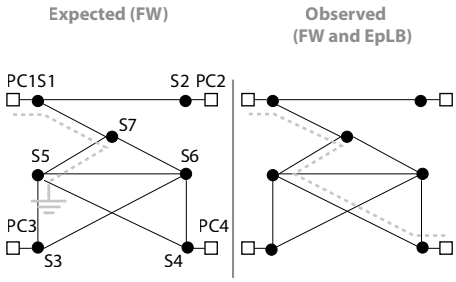


Figure 46: An example of the distributed conflict class *policy suppression by upstream packet modification*, observed when co-deploying Firewall (FW) and End-point Load Balancer (EpLB)

Discussion
The above example implies a type of security attack relevant to this conflict class, namely *firewall bypassing* by means of packet modification, which was illustrated by Porras et al. [88]. In this attack, traffic is modified before getting through the firewall in such a way that it is not filtered there, and then re-modified afterwards to reach the destination targeted by the attacker. The authors introduced in this work a security policy enforcement kernel called *FortNOX* to check for rule conflicts based on the alias-set rule reduction algorithm. Due to the focus on security compliance, conflicts between control applications are not addressed in detail as in our work. Moreover, conflict checking is omitted between a rule pair if they both have forwarding action, though in reality, a conflict can still occur between them.

A similar conflict class was examined by Reyes [90] and referred to in his work as *occlusion*. We notice it to be an instance of the hidden conflict class *action suppression by packet modification* in our work (see Section 4.3.3) as this case reflects a disruption in the control plane and could not be discerned by rules in the data plane.

4.2.7 Policy suppression by changes to paths

Network traffic interested by a control application can be forwarded on some path not containing its target network devices. As a result, the policies of that application do not take effect.

Property
A control application puts rules on its target network devices to control a certain slice of network traffic. However, that traffic gets forwarded on a path not containing that set of devices due to the influence of other applications.

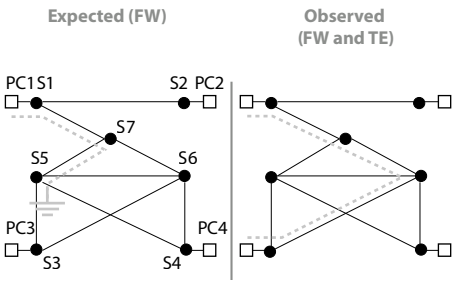


Figure 47: An example of the distributed conflict class *policy suppression by changes to paths*, observed when co-deploying the Firewall (FW) and Traffic Engineering (TE) control applications

Effect
The policies of the affected control application is not fulfilled, even though its rules are deployed properly.

Example
Figure 47 illustrates a conflict of this class observed when executing the Firewall (FW) and Traffic Engineering (TE) control applications together. FW puts its rules on switch S5 to drop all TCP/UDP traffic from PC1 to PC3; however, this traffic is directed to switch S6 by TE rules installed at switch S7. The path changing due to TE rules thus renders FW rules ineffective.

Discussion

The example above shows a naive scenario of firewall bypassing by changing the traffic path. In practice, the firewall should be placed on the network boundary or cover all possible paths that the involved traffic may traverse. The same tactic is applicable for all control applications to avoid conflicts of this class.

A subtle case of this conflict class is examined by Reyes [90], which is referred to in his work as *incomplete transformation*. This case arises when the packet modification intended by a control application for both forward and backward directions, but the backward modification fails due to path changing triggered by another application. The consequence could be failure in communication between end-points. We reproduce this case in the example illustrated in Figure 48. Active Host Shadowing (aHS) rules are active at switch S5 to forward all TCP/UDP traffic from PC1 targeting PC3 to PC4 by modifying the destination (IP and MAC addresses) from PC3 to PC4. This modification should be transparent to the sender (so that PC1 thinks that it is communicating with PC3 and not PC4); therefore, the returning traffic from PC4 must be modified again to appear as if it originates from PC3. aHS installs its rules for both the forward and backward transformation of packets at switch S5. While everything goes fine in the isolated deployment of aHS, an unexpected case is observed in the co-deployment of aHS and Traffic Engineering (TE). In this case, TE is active at switch S4 to forward all TCP/UDP traffic from PC4 to PC1 on the path via switch S6, leading to this traffic not being transformed at switch S5 by aHS rules. The final outcome is unpleasing: PC1 is trying to establish a connection with PC3 but receiving instead unsolicited answers from PC4, PC1 drops these packets of PC4 and the communication session initiated by PC1 fails.

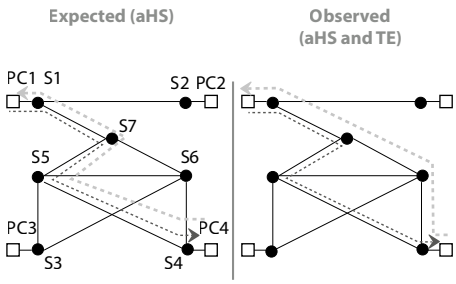


Figure 48: A subtle case of the distributed conflict class *policy suppression by changes to paths*, observed when co-deploying the active Host Shadowing (aHS) and Traffic Engineering (TE) control applications

This subtle case, similar to the naive scenario with Firewall, can be rectified by letting aHS install its rules on switches covering all possible paths between PC1 and the set of PC3 and PC4, two intuitive options for the target switches of aHS in this case would be either i) S7 or ii) S5 and S6. It is worth noting that while conflicts of this class can be avoided by this way, conflicts of other classes can still occur. The investigation towards a comprehensive conflict avoidance is beyond the scope of this work.

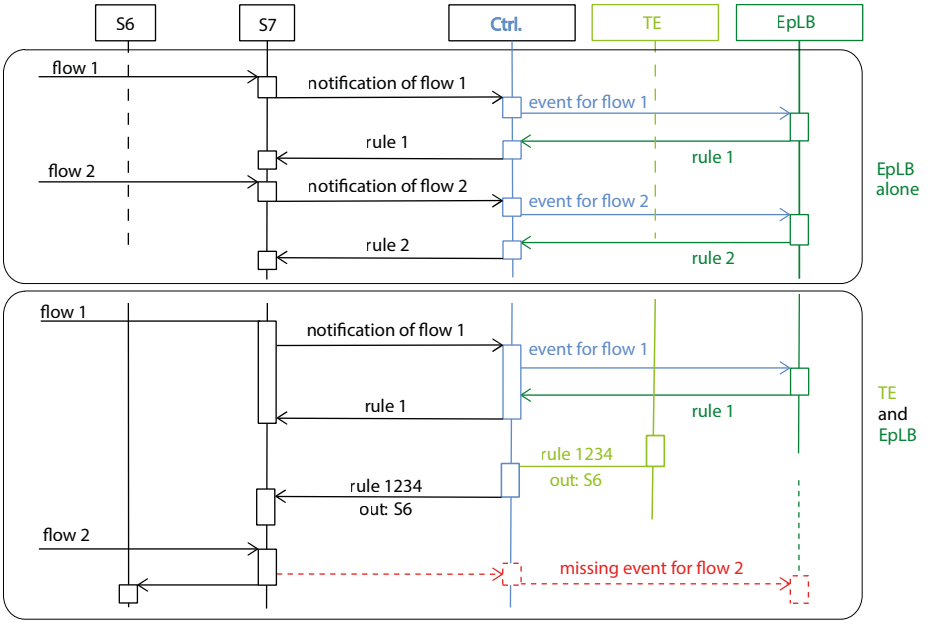


Figure 49: An example of a hidden conflict observed when co-deploying the End-point Load Balancer (EpLB), Traffic Engineering (TE) and Shortest Path First Routing (SPF) control applications, observed in experiment 6 in Chapter 3.6.3. S6, S7 are switches in the data plane, Ctrl is the controller. The match fields of rule 1 cover only traffic flow 1, of rule 2 only traffic flow 2, and of rule 1234 both. Rule 1234 has lower priority than rule 1 and rule 2.

4.3 Hidden conflicts

We found a type of conflict that are orthogonal with the local and distributed conflicts in that it cannot be discerned by purely analysing rules in the data plane, but needs the control plane assertion to determine its existence. We name this type of conflict as *hidden conflict*.

Experiments 5, 6, 7, 8 and 9 in Chapter 3.6.3 depict the traits of hidden conflicts. We reproduce the one from experiment 6 in Figure 49 to facilitate our analysis (this example was also presented in a shorter form in Chapter 1.3.3 to illustrate hidden conflicts). The End-point Load Balancer (EpLB) application, in its isolated execution, reacts to each notification of new traffic flows (e. g., via packet-in events) with respective rules (see the upper box of Figure 49). In its co-deployment with the Traffic Engineering (TE) application (see the lower box of Figure 49), after TE installed rule 1234 covering traffic flows 1 and 2, EpLB still handles flow 1 correctly but does not install its rules to handle flow 2 as observed in its isolated run. Our analysis reveals the cause to be the missing notification to EpLB for flow 2, since this flow is handled locally by rule 1234 from TE. This cause differs completely from what can be concluded by purely examining rules in the data plane: rules 1 and 1234 expose a *generalization* local conflict (see Section 4.1.2), whose commonly accepted effect is that the broader rule (rule 1234) defers to the more specific one (rule 1) if the incom-

ing traffic matches both. In other words, the main effect manifested via data plane's rules can have its side-effect impacting the control applications. Hidden conflicts are those arising from such side-effects.

Events	Actions
packet	out
flow removal by internal timer[1]	drop
packet with action	modify
device/link/port startup	escalate data packet
device/link/port shutdown	escalate status
device/link/port failure	escalate notification
device's state query	modify and out
	modify and escalate

Table 4.3: Device primitives [25]

Events	Actions
device/link/port enabled	publish event
device/link/port disabled	install rule
device/link/port failure	modify rule
packet escalated	delete rule
notification escalated	nop
function call from application	
device's state response	

Table 4.4: Controller primitives [25]

Events	Actions
startup	install rule
shutdown	modify rule
packet-in	delete rule
topology change	delegate packet to device[2]
notification	nop
device's state response	

Table 4.5: Application primitives [25]

[1] flow timeout
[2] by sending packet-out

The characteristic of hidden conflicts suggests a method to explore them via the examination of the potential influences used by applications and the SDN control mechanics that are susceptible to each influence.

We examine hidden conflicts with reference to the operational model of the OpenFlow SDN [79]. As far as we are concerned, this model is also valid for the common SDN technologies in principle, e. g., POF [98, 61], P4-based SDN using P4 [12] and P4Runtime [81]. At first, we derive the interaction primitives between devices, the controller and control applications, which are triggered by events in the network, e. g., packets arriving at a device. The influence on each combination of interactions can lead to hidden conflicts, we assess thus each combination in terms of its susceptibility to influence. We acquire thereby a conflict model including: i) the susceptible combinations of interaction primitives, ii) the conditions in which they may be influenced and iii) the potential impact on the control application that relies on a combination at the time when one of the conditions is met. Based on how the interaction primitive combinations can be disturbed, we are able to organize hidden conflicts in different classes.

4.3.1 Interaction primitives

The interactions transpire in OpenFlow SDN between two pairs of entities: the device and the controller, the controller and the control application. The triggering events along with the possible actions being triggered are listed in Tables 4.3, 4.4 and 4.5.

	Device		Controller		Application		Disturbance	Note
#	event	action	event	action	event	action	factor	
1	startup	escalate status	device/port/link enabled	publish event	topology change	NoP		
2	startup	escalate status	device/port/link enabled	publish event	topology change	install rules	a,b,c,d,e	
3	startup	escalate status	device/port/link enabled	publish event	topology change	modify rules	a,b,c,d,e	
4	startup	escalate status	device/port/link enabled	publish event	topology change	delete rules	a,b,c,d,e	
5	startup	escalate status	device/port/link enabled	publish event	topology change	packet-out	a,b,c,d,e	
6	shutdown	escalate status	device/port/link disabled	publish event	topology change	NoP		
7	shutdown	escalate status	device/port/link disabled	publish event	topology change	install rules	a,b,c,d,e	
8	shutdown	escalate status	device/port/link disabled	publish event	topology change	modify rules	a,b,c,d,e	
9	shutdown	escalate status	device/port/link disabled	publish event	topology change	delete rules	a,b,c,d,e	
10	shutdown	escalate status	device/port/link disabled	publish event	topology change	packet-out	a,b,c,d,e	
11	failure	escalate status	device/port/link failure	publish event	topology change	NoP		
12	failure	escalate status	device/port/link failure	publish event	topology change	install rules	a,b,c,d,e	
13	failure	escalate status	device/port/link failure	publish event	topology change	modify rules	a,b,c,d,e	
14	failure	escalate status	device/port/link failure	publish event	topology change	delete rules	a,b,c,d,e	
15	failure	escalate status	device/port/link failure	publish event	topology change	packet-out	a,b,c,d,e	
16	state query	state response	device state response	publish event	device state response	NoP		
17	state query	state response	device state response	publish event	device state response	install rules	a,b,c,d,e	
18	state query	state response	device state response	publish event	device state response	modify rules	a,b,c,d,e	
19	state query	state response	device state response	publish event	device state response	delete rules	a,b,c,d,e	
20	state query	state response	device state response	publish event	device state response	packet-out	a,b,c,d,e	
21	packet	out	–	–	–	–		device only
22	packet	drop	–	–	–	–		device only
23	packet	modify(+out)	–	–	–	–		device only
24	packet	escalate	packet esc.	publish event	packet-in	NoP		
25	packet	escalate	packet esc.	publish event	packet-in	install rules	a,b,c,d,e	
26	packet	escalate	packet esc.	publish event	packet-in	modify rules	a,b,c,d,e	
27	packet	escalate	packet esc.	publish event	packet-in	delete rules	a,b,c,d,e	
28	packet	escalate	packet esc.	publish event	packet-in	packet-out	a,b,c,d,e	
29	flow timeout	escalate notification	notification escalation	publish event	flow removed	NoP		
30	flow timeout	escalate notification	notification escalation	publish event	flow removed	install rules	a,b,c,d,e	
31	flow timeout	escalate notification	notification escalation	publish event	flow removed	modify rules	a,b,c,d,e	
32	flow timeout	escalate notification	notification escalation	publish event	flow removed	delete rules	a,b,c,d,e	
33	flow timeout	escalate notification	notification escalation	publish event	flow removed	packet-out	a,b,c,d,e	

Table 4.6: Combinations of interaction primitives [25]

Controller		Application		Disturbance	Note
event	action	event	action	factor	
mock event	publish event	event	NoP		mock event sent by apps relayed via controller
mock event	publish event	event	install/modify/delete rules/packet-out	a,b,c,d,e,f,g	mock event sent by apps relayed via controller
		mock event	NoP		app sends mock event directly to app
		mock event	install/modify/delete rules/packet-out	a,b,c,d,e,f,g	app sends mock event directly to app

Table 4.7: Mock events based on the interaction primitives [25]

4.3.2 Interaction combinations

The combinations of interaction primitives between devices, controller and applications are listed in Table 4.6. We assume that there is no direct interaction between devices and applications, i. e., all interactions are relayed and translated by the controller. Some interaction combinations are impossible in practice and can be eliminated from further analysis, these include

- the items marked as "device only", since they do not reflect an actual interaction;
- the items where the application action is void, marked "NoP".

Mock events that have no base in an actual state change in the network can be generated to exploit the interactions between the controller and applications, either for productive use, e. g., to diagnose a network problem, or with malicious intent. Table 4.7 shows the combinations of the interactions between applications and the controller resulting from the mock events being introduced at the controller level or at the application level. Any of the events from Table 4.6 intended for the controller or the application could be exploited as a mock event.

4.3.3 Classifying hidden conflicts based on disturbance factors

Network behaviour can be influenced unexpectedly by the disruption of the above interactions listed in Table 4.6, causing hidden conflicts. We classify hidden conflicts in the following according to the disturbance factors that have been observed in experiments or that are conceivable.

a) Event suppression by local handling
Property A switch handles an incoming packet locally, instead of escalating it to the controller.
Effect The involved application is deprived of the event notification and does not function properly.
Example An illustration for a conflict of this class is observed in experiment 6 in Chapter 3.6.3, which is reproduced in Figure 49: the presence of rule 1234 from the

Traffic Engineering application results in the missing notification for End-point Load Balancer when flow 2 arrives at switch S7. Experiments 5 in Chapter 3.6.3 reveals another conflict of this class.

b) Event suppression by upstream traffic looping

The term *upstream* was explained in Section 4.2. In essence, the *upstream* direction, considered in reference to a target network device of a control application and its interested traffic flow, indicates the case in which the mentioned incident (e. g., traffic dropping, looping) happened on a traffic flow, causing this flow unable to reach the target network device.

Property The target traffic of a control application is caught in a loop and could not reach its target network devices as expected in its isolated deployment.

Effect The involved application is deprived of the event notification and does not function properly.

Example The effect of a conflict of this class is depicted in Figure 50, observed when co-deploying the control applications Destination-based Path Load Balancer (PPLB4D), Traffic Engineering (TE) and passive Host Shadowing (pHS). In its isolated execution, pHS installs its rules at switch S4 in reaction to packet-in events triggered by the traffic flow from PC1 to PC4, these rules forward TCP/UDP traffic destined to PC4 to PC3. In the co-deployment of these applications, the combined effect of the rules from PPLB4D and TE causes the traffic sent by PC1 not able to approach switch S4 but stuck in a loop among three switches S5–S3–S6–S5. Consequently, the *traffic loop* in the *upstream* direction of switch S4 suppresses the events expected by pHS for its correct functionality.

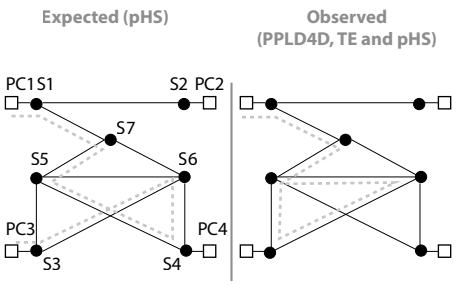


Figure 50: An example of the hidden conflict class *event suppression by upstream traffic looping*, observed when deploying together the control applications Destination-based Path Load Balancer (PPLB4D), Traffic Engineering (TE) and passive Host Shadowing (pHS)

Discussion Unlike distributed conflicts, there is no *downstream* counterpart for hidden conflicts in terms of the *upstream* characteristic, since the incidents occurring in the downstream direction of a control application's target switch could not influence the control mechanics of that application.

c) Event suppression by upstream traffic dropping

Property The target traffic of a control application gets dropped by other applications, thus could not reach its target network devices as expected in its isolated deployment.

Effect The involved application is deprived of the event notification and does not function properly.

Example An instance of this conflict class is observed when co-deploying the control applications Firewall (FW) and passive Host Shadowing (pHS), as illustrated in Figure 51. In its isolated execution, pHS installs its rules at switch S4 in reaction to packet-in events triggered by the traffic flow from PC1 to PC4, these rules forward TCP/UDP traffic destined to PC4 to PC3. In the co-deployment of the applications, this traffic gets dropped at switch S5 by FW rules and could not approach switch S4. Consequently, the *traffic dropping* in the *upstream* direction of switch S4 suppresses the events expected by pHS for its correct functionality.

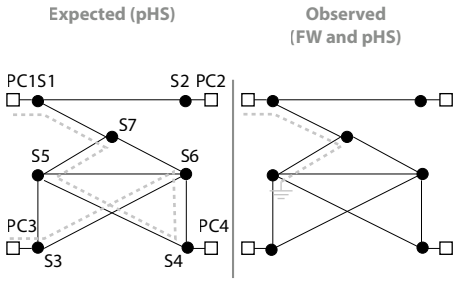


Figure 51: An example of the hidden conflict class *event suppression by upstream traffic dropping*, observed when deploying together the control applications Firewall (FW) and passive Host Shadowing (pHS)

d) Event suppression by changes to paths

Property Prevention of escalation by changes to paths, i. e., when a traffic flow interested by an application is forwarded around the switch holding a rule that would escalate packets of that flow to the controller.

Effect The involved application is deprived of the event notification and does not function properly.

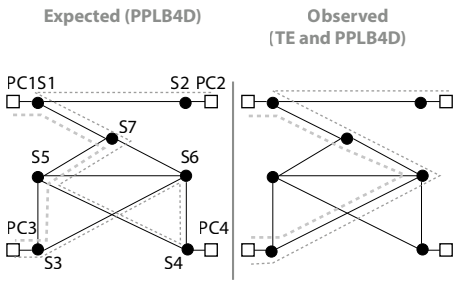


Figure 52: An example of the hidden conflict class *event suppression by changes to paths*, observed when deploying together the control applications Traffic Engineering (TE) and Destination-based Passive Path Load Balancer (PPLB4D)

Example Experiment 8 in Chapter 3.6.3 illustrates this hidden conflict class, the expected and observed network behaviour is reproduced in Figure 52. The control application Destination-based Passive Path Load Balancer (PPLB4D) reacts to packet-in events triggered by switch S5 for TCP/UDP sessions destined to PC3 by installing rules to balance these sessions on different paths. When being deployed with Traffic Engineering (TE), these TCP/UDP sessions are forwarded by TE rules at switch S7 to switch S6. Switch S5 does not receive the traffic flow as in its isolated execution, thus does not trigger any packet-in event required by PPLB4D for its correct functionality.

e) Action suppression by packet modification

Property A device executing rules that modify packets before they are escalated to the controller. The escalation can be performed by that device or by subsequent devices. The escalated packets are no longer accepted within an application's scope.

Effect The affected control application does not react against the event notification containing the altered packet. Its intention thus is not fulfilled.

Example This hidden conflict class is demonstrated in Experiment 9 in Chapter 3.6.3, the expected and observed network behaviour is reproduced in Figure 53. The control application Destination-based Passive Path Load Balancer (PPLB4D) balances TCP/UDP traffic destined to PC3 by placing its rules at switch S5 in reaction to packet-in events originated from this switch for this slice of traffic. End-point Load Balancer (EpLB) registers for packet-in events sent from switch S7 also for TCP/UDP sessions destined to PC3. It reacts by installing rules in this switch to balance these sessions between PC3 and PC4, the relevant fields in each packet are rewritten (using the OpenFlow's *set_field* function) where necessary. When executing these two applications together, some TCP/UDP sessions from PC1 and PC2 to PC3 are not balanced by PPLB4D as they were in its isolated run. The identified cause is that the traffic of these sessions are modified at switch S7 by EpLB rules to have its new destination as PC4, this traffic is then sent to switch S5, PPLB4D receives the associated packet-in event but is not interested in and ignores it. In summary, the *packet modification* by EpLB leads to the *suppression of action* from PPLB4D and hence its intention is not accomplished.

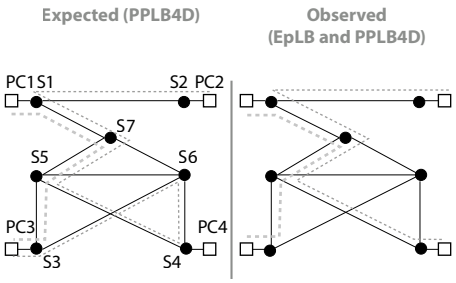


Figure 53: An example of the hidden conflict class *action suppression by packet modification*, observed when deploying together the control applications End-point Load Balancer (EpLB) and Destination-based Passive Path Load Balancer (PPLB4D)

f) Undue trigger

Property Contrary to *action suppression by packet modification*, a control application can be "tricked" into installing, modifying or removing rules by packets modified by other applications before escalation. This can also happen in the course of an attack by mock packets or by falsified events sent by attackers.

Effect The affected control applications reacts (by installing, modifying or removing rules) against the undue events, which could result in unexpected network behaviour.

Example An instance of this conflict class is observed when co-deploying the End-point Load Balancer (EpLB) and passive Host Shadowing (pHS) as shown previously in Figure 45, which exposes also a sample of the distributed conflict class *policy suppression by downstream packet modification*. A slice of traffic altered by EpLB rules at switch S7 triggers packet-in events at switch S5, inducing pHS to install its rules to re-modify this traffic. As a result, the overall policies of EpLB are not achieved. Although this experiment reveals a case of this hidden conflict class, it is controversial in terms of the conflict consequence on the affected application, being pHS, as its

intent does not appear to be violated, thus this hidden conflict seems negligible. Yet, the final assessment of the consequence is up to the network operator.

An attacker that wangles to act as an application or to control an existing one could fabricate counterfeit events and send them to other applications to provoke for their undue reactions. A counterfeit event can be any event in Table 4.6 meant for the controller or the application.

g) Tampering with event subscription
Property The disruption entailing this conflict class is contingent on applications being able to modify each other's subscriptions. As a result, a control application might cause an *undue trigger* hidden conflict to another application or suppress events by unsubscribing events for it, or subscribes it with unsolicited events. This case can also transpire in a security attack.
Effect The affected application does not function as expected.
Discussion Although this conflict class has not been observed in our set of experiments, it is conceivable in case an SDN implementation allows a control application to modify the subscription of the others, e. g., one of them acts as a coordinator that intercepts the other's activities concerning rule deployment. The event subscription of a control application could also be tampered in a security attack if the attacker is able to intercept the communication channel between the controller and that application. He could then act as the victim application and, for example, register for new events to spy the network, cause *undue trigger* hidden conflicts to other applications, or install malicious rules in the network.

4.3.4 Susceptible interactions and impact

The interaction combinations shown in Tables 4.6 and 4.7 may be susceptible to one or more hidden conflict causes, i. e., disturbance factors. Each of them is analyzed and the disturbances that they are sensitive to are determined, the results are noted in the *Disturbance factor* column of these tables. A combination is sensitive if it can be disrupted by one of the disturbance factors. We notice that the consequences of a disruption strongly relate to the purpose of the interaction set including missing rules, redundant rules or wrong rules in one device or more, which may provoke anomalous network behaviour.

4.4 Summary

We have presented various conflict classes in SDN grouped in three broad categories: local, distributed and hidden conflicts. Local conflicts feature those occurring between rules in the same rule table of an SDN device, they are classified by comparing rules regarding their priority, match space overlap and actions. Distributed conflicts are determined based on the combined effect of rules situated in different devices,

which influence the same traffic flow in an unexpected manner, e. g., causing traffic looping or dropping. In contrast to the first two categories, hidden conflicts are contingent on the disruptions in the control plane that lead to undesired network behaviour.

Our systematic analysis in classifying local conflicts supplements the existing results introduced by Hamed and Al-Shaer [43] and Pisharody [86]. Reyes presents a taxonomy of distributed conflicts in SDN [90], a part of that is reused in our work for distributed conflict assortment while the irrelevant part is not incorporated. For example, we consider the class *invariant contention* (containing two subclasses *dispersion* and *focusing*) in his work not to be a conflict class according to our conflict definition since the involved applications' intents are still fulfilled, although there exists certain deviation in network behaviour between the isolated and concurrent deployment of applications. More discussion on his work is provided in Chapter 2.3.4. Distributed conflicts have been also researched in other forms, e. g., as network invariants [52] or policy compliance [51]. However, with different focuses, these are not meant to detect conflicts between rules installed by various control applications as ours. The completely new conflict category discovered in our work, namely hidden conflicts, opens another research area in the ecosystem of conflicts in SDN.

5 Conflict Detection

Conflicts need to be handled to avoid unexpected network behaviour. We present in this chapter methods to detect conflicts based on their patterns or properties identified in Chapter 4. These patterns and properties portray the relationships between rules in the same rule table or in different ones. They can be determined from the rule components including the priority, match fields and actions. The detection of conflicts demands the comparison of these rule components.

Existing solutions do not facilitate the general rule comparison at the granular level associated with our conflict classification, e. g., in determining if the match space of a rule is a subset, superset of another rule, or if they intersect each other. We introduce thus the general concept of *multi-property set* and a method to extract the relationship between sets of this type using the $\cdot r$ (pronounced as "dot r") operator, both of which enable the rule comparison without restrictions endured by existing research. As the match and action components can be expressed differently in each rule, for instance, some match component specifies only layer 2 addresses, some only layer 3 information, their homogeneous representation is required for the automatic comparison. We propose the so-called *matchmap* and *actmap* concepts for this purpose. These concepts are roughly illustrated in the below example with rules 1 and 2 following the OpenFlow SDN standard:

```
rule 1: priority=3, match={ipv4_src=192.168.1.1, ipv4_dst=192.168.2.2, ip_proto=6, tcp_dst=80}, action={
    ↪ output:3}
rule 2: priority=2, match={ipv4_src=192.168.1.0/24, ip_proto=6}, action={set_field:ipv4_src=192.168.1.3,
    ↪ output:2}
```

By manual analysis, it is obvious that the match space of rule 1 is a subset of rule 2's, and thus, their match overlap is that of rule 1. In conjunction with their priority and action relationship, we can infer a local conflict of the *generalization* class between them. In order to automate these steps, rules 1 and 2 need to be represented in the same format, one possibility is:

```
rule 1: priority=3, match={ipv4_src=192.168.1.1, ipv4_dst=192.168.2.2, ip_proto=6, tcp_dst=80},
        action={set_field:none, output:3}
rule 2: priority=2, match={ipv4_src=192.168.1.0/24, ipv4_dst = any, ip_proto=6, tcp_dst = any},
        action={set_field:ipv4_src=192.168.1.3, output:2}
```

The original match fields and actions of these rules are now converted to their *matchmap* and *actmap*, which are uniform. Each match or matchmap corresponds to a multi-property set with the individual properties comprising *ipv4_src, ipv4_dst, ip_proto, tcp_dst*. The formulated algebra associated with the $\cdot r$ operator presented

in this chapter enables the automatic reasoning about the relationship between multi-property sets.

Having these tools available, we show how they can be employed in detecting conflicts. For distributed conflicts and some hidden conflict classes characterised by rules in multiple rule tables, we establish the connections of rules in different tables that handle the same traffic flow and reason about all possible anomalies therein, these rules' connections are referred to as *rule graph*. Some distributed and hidden conflict classes entail extreme complexity or expose the interpretative nature in their identification, some appear to be security-related concerns. We provide insightful discussions on measures to cope with them and on practical implications.

5.1 Multi-property set and $\cdot r$ operator

The key information of SDN rules for reasoning about conflicts according to our established conflict taxonomy (see Chapter 4) includes their priority, match and actions. While the comparison of the rule priority as number is intuitive, it is not simple in comparing rules' matches and actions because each rule can specify them differently. The rule match, being a set containing multiple fields, adds more complexity. As discussed in Chapter 2.3.4, the rigidity of the existing methods hampers their utilization in comparing SDN rules. Therefore, we introduce a new notion, namely *multi-property set*, corresponding to the rule match and the operator $\cdot r$ for deriving relationships between sets of this type. In this section, we elaborate these tools and their general application from the mathematical viewpoint. Their application for SDN rules' comparison is demonstrated in the subsequent sections.

5.1.1 Multi-property set

Match fields of an SDN rule can be compared to a set composed of different single properties, e. g., source IP address, destination IP address, source TCP port, destination TCP port. We refer to such a set as *multi-property set*, as opposed to *single-property set*. Some examples of single-property sets are the set of people in the age of 30 (we may add the context, e. g., in country X), the set of students passing the math exam (context: in university Y). Multi-property sets can be the set of people (in country X) in the age of 30, above 1,5 meter tall, having driver's licences, or the set of students (in university Y) passing the math, physics and chemistry exams. It is evident that a multi-property set is formed by the intersection of multiple single-property sets, each associates with the constraint specified for one of the properties of the multi-property one. For example, the set of students passing the math, physics and chemistry exams is formed by the intersection of the three single-property sets, the first being those passing the math exam, the second the physics exam, and the third the chemistry exam.

To be more precise, we give the definitions of single-property set and multi-property set as follows.

Definition 5.1 (Single-property set). *A single property set is a set that is associated with only one property. All elements in the set must satisfy the constraint specified for that property.*

Definition 5.2 (Multi-property set). *A multi-property set is a set that is associated with more than one property. It corresponds to the intersection of the single-property sets, each is created from the constraint specified for one of the properties of the multi-property set.*
Let S be a multi-property set of n properties, $n \geq 2$,
let S_k be a single-property set corresponding to the constraint specified for the k^{th} property of S, $1 \leq k \leq n$.
$\Rightarrow S = \cap_{k=1}^{n} S_k = S_1 \cap S_2 \cap \cdots \cap S_n$

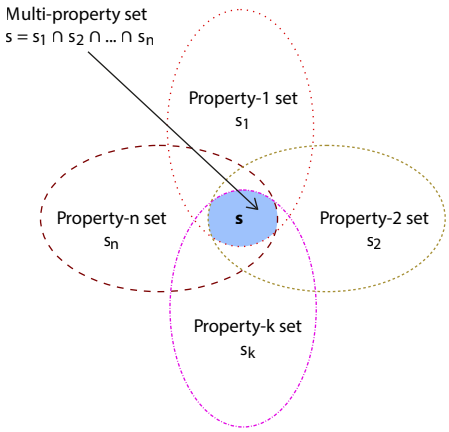


Figure 54: Multi-property set as the intersection of multiple single-property sets. Each property-i set is a single-property set created by the constraint specified for the property i of the multi-property set. The multi-property set has n properties: property 1, property 2..., property k..., property n.

An illustration of the multi-property set is shown in Figure 54.

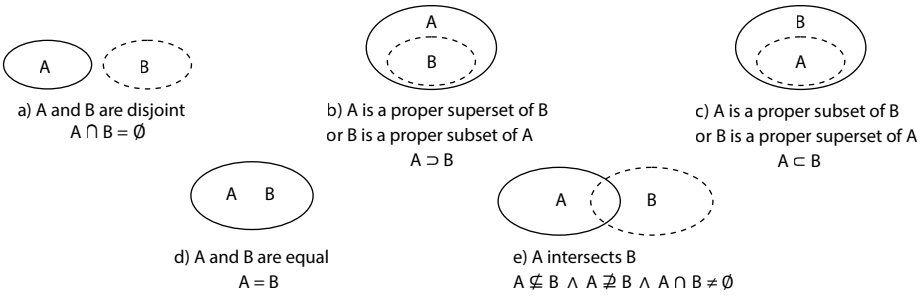


Figure 55: The relationship between two certain sets A and B

5.1.2 Comparison of multi-property sets using $\cdot r$ operator

Given any two sets A and B, their relationship can be:

- Disjoint: $A \cap B = \emptyset$
- Equal: $A = B$
- Proper subset: $A \subset B$
- Proper superset: $A \supset B$

- Intersecting: $A \nsubseteq B \wedge A \nsupseteq B \wedge A \cap B \neq \emptyset$

Figure 55 illustrates these relationships. In this work, two sets are said to overlap if they are not disjoint, i. e., their relationship is either equal, proper subset, proper superset, or intersecting. The relationship of two multi-property sets can be reasoned about through the relationships of their associated single-property sets. This is demonstrated in Figure 56.

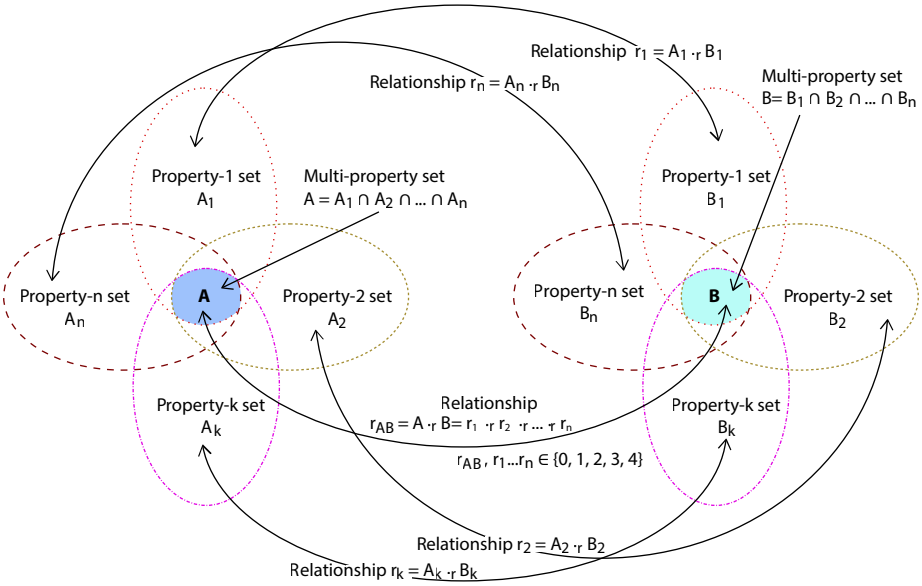


Figure 56: The relationship of two multi-property sets A and B, denoted as $r_{AB} = A \cdot_r B$, is calculated by combining all individual relationships of their associated single-property sets $r_1 = A_1 \cdot_r B_1$, $r_2 = A_2 \cdot_r B_2, \cdots r_n = A_n \cdot_r B_n$. The relationship combination operator is denoted as $\cdot r$ (pronounced as "dot r").

We introduce the relationship combination operator, denoted as $\cdot r$ (pronounced as "dot r"), to combine the relationships of individual single-property sets to obtain the overall relationship between two multi-property sets. Given two multi-property sets A and B, each has n properties, being property 1, property 2…property n, we represent the single-property set created by the constraint corresponding to property i of set A as A_i, of B as B_i. The sets A and B can then be represented via their corresponding single-property sets as:

$A = A_1 \cap A_2 \cap \cdots \cap A_n$

$B = B_1 \cap B_2 \cap \cdots \cap B_n$

The relationship between A and B, denoted as r_{AB} or $A \cdot_r B$, is calculated based on the relationships of their individual single-property sets as following:

$$r_{AB} = A \cdot_r B = (A_1 \cdot_r B_1) \cdot_r (A_2 \cdot_r B_2) \cdot_r \cdots \cdot_r (A_n \cdot_r B_n) = r_1 \cdot_r r_2 \cdot_r \cdots \cdot_r r_n$$

where r_i is the relationship between the single-property sets A_i and B_i, $i \in \{1 \cdots n\}$. The relationship between two sets A and B, written as r_{AB} or $A \cdot_r B$, is determined

by how they intersect each other. The following definitions hold for both single-property and multi-property sets (i. e., regardless of A and B being single-property or multi-property sets).

Definition 5.3 (Encoding of the relationship between two sets)**.** *We define the encoding of the relationships between two certain sets as follows.*

- *A and B are $disjoint$, represented as $r_{AB} = A \cdot_r B = 0$, if $A \cap B = \emptyset$*
- *A and B are $equal$, represented as $r_{AB} = A \cdot_r B = 1$, if $A \cap B = A = B$*
- *A is a $propersubset$ of B, represented as $r_{AB} = A \cdot_r B = 2$, if $\forall x \in A \Rightarrow x \in B \wedge \exists x \in B \mid x \notin A$, or $A \cap B = A \wedge A \neq B$*
- *A is a $propersuperset$ of B, represented as $r_{AB} = A \cdot_r B = 3$, if $\forall x \in B \Rightarrow x \in A \wedge \exists x \in A \mid x \notin B$, or $A \cap B = B \wedge A \neq B$*
- *A $intersects$ B, represented as $r_{AB} = A \cdot_r B = 4$, if $\exists x \in A \mid x \notin B \wedge \exists x \in B \mid x \notin A \wedge A \cap B \neq \emptyset$, or $A \not\subseteq B \wedge A \not\supseteq B \wedge A \cap B \neq \emptyset$*

In the above encoding, the $\cdot r$ operator can be represented by a function with input as two sets and output as a number:
$\cdot r : (A, B) \rightarrow C$, where A, B are (single-property or multiple-property) sets, $C \in \{0, 1, 2, 3, 4\}$

Theorem 5.1. *Using the encoding in Definition 5.3, i. e., $disjoint$ relationship as 0, $equal$ as 1, $proper\ subset$ as 2, $proper\ superset$ as 3, $intersecting$ as 4, the relationship combination operator $\cdot r$ works on the input of these encoding numbers as follows:*
$X \in \{0,\ 1,\ 2,\ 3,\ 4\}$

$$0 \cdot_r X = 0 \tag{5.1}$$

$$X \cdot_r X = X \tag{5.2}$$

$$X \cdot_r 1 = X \tag{5.3}$$

$$2 \cdot_r 3 = 4 \tag{5.4}$$

$$X \cdot_r 4 = 4\ if\ X \neq 0 \tag{5.5}$$

In this case, the $\cdot r$ operator can be represented as a function whose input being the above encoding numbers:
$\cdot r : (X, Y) \rightarrow Z,\ where\ X, Y, Z \in \{0, 1, 2, 3, 4\}$
The relationship combination operator has the commutative and associative properties, i. e.,

$$X \cdot_r Y = Y \cdot_r X \tag{5.6}$$

$$X \cdot_r Y \cdot_r Z = (X \cdot_r Y) \cdot_r Z = X \cdot_r (Y \cdot_r Z)\ where\ X, Y, Z \in \{0, 1, 2, 3, 4\} \tag{5.7}$$

Example:
$2 \cdot_r 3 = 3 \cdot_r 2 = 4$
$1 \cdot_r 2 \cdot_r 3 \cdot_r 2 = (1 \cdot_r 2) \cdot_r (3 \cdot_r 2) = 2 \cdot_r 4 = 4$
Now we need to prove the Equations 5.1 to 5.7.
Due to the associative property of the intersection operation ($\cap$) [85], we have

$A \cap B = (A_1 \cap A_2 \cap \cdots \cap A_n) \cap (B_1 \cap B_2 \cap \cdots \cap B_n)$
$= (A_1 \cap B_1) \cap (A_2 \cap B_2) \cap \cdots \cap (A_n \cap B_n)$
This is expressed with the $\cdot r$ operator as: $r_{AB} = A \cdot_r B = r_1 \cdot_r r_2 \cdot_r \cdots \cdot_r r_n$
Without loss of generality, take $A = A_1 \cap A_2$ and $B = B_1 \cap B_2$
$\Rightarrow A \cap B = A_1 \cap B_1 \cap A_2 \cap B_2 = (A_1 \cap B_1) \cap (A_2 \cap B_2)$ and
$r_{AB} = r_1 \cdot_r r_2$, where r_1 is the relationship between the two sets associated with the first properties A_1 and B_1, r_2 between A_2 and B_2.
We assume that there always exists a multi-property subset X if none of its corresponding single-property subset is empty, which means
$Assumption$:
$$For\ X = X_1 \cap X_2 \cap \cdots \cap X_n, if\ X_i \neq \emptyset\ where\ i \in \{1..n\},\ then\ X \neq \emptyset \tag{5.8}$$

Proof of Property 5.6
The relationship combination operator has the commutative property
$X \cdot_r Y = Y \cdot_r X$ where $X, Y \in \{0, 1, 2, 3, 4\}$
Proof
Due to the commutative and associative properties of the intersection of sets [85], we have
$A \cap B = (A_1 \cap A_2) \cap (B_1 \cap B_2) = (A_1 \cap B_1) \cap (A_2 \cap B_2) = (A_2 \cap B_2) \cap (A_1 \cap B_1)$
which means
$r_{AB} = r_1 \cdot_r r_2 = r_2 \cdot_r r_1$
This can be generalized as $X \cdot_r Y = Y \cdot_r X$ where $X, Y \in \{0, 1, 2, 3, 4\}$ according to our encoding in Definition 5.3.

Proof of Property 5.7
The relationship combination operator has the associative property
$X \cdot_r Y \cdot_r Z = (X \cdot_r Y) \cdot_r Z = X \cdot_r (Y \cdot_r Z)\ where\ X, Y, Z \in \{0, 1, 2, 3, 4\}$
Proof
Consider the certain three-property sets A and B
$A = A_1 \cap A_2 \cap A_3$
$B = B_1 \cap B_2 \cap B_3$
Due to the commutative and associative properties of the intersection of sets, we have
$A \cap B = A_1 \cap A_2 \cap A_3 \cap B_1 \cap B_2 \cap B3 = (A_1 \cap B_1) \cap (A_2 \cap B_2) \cap (A_3 \cap B_3) =$
$((A_1 \cap B_1) \cap (A_2 \cap B_2)) \cap (A_3 \cap B_3) = (A_1 \cap B_1) \cap ((A_2 \cap B_2) \cap (A_3 \cap B_3))$
Which means
$r_{AB} = r_1 \cdot_r r_2 \cdot_r r_3 = (r_1 \cdot_r r_2) \cdot_r r_3 = r_1 \cdot_r (r_2 \cdot_r r_3)$
This can be generalized as
$X \cdot_r Y \cdot_r Z = (X \cdot_r Y) \cdot_r Z = X \cdot_r (Y \cdot_r Z)\ where\ X, Y, Z \in \{0, 1, 2, 3, 4\}$

Proof of Equation 5.1
Equation 5.1: $0 \cdot_r X = 0$
For any set S, we have $S \cap \emptyset = \emptyset$,

If $A_1 \cap B_1 = \emptyset$ (which means $r_1 = 0$ according to our encoding in Definition 5.3) $\Rightarrow A \cap B = (A_1 \cap B_1) \cap (A_2 \cap B_2) = \emptyset \cap (A_2 \cap B_2) = \emptyset$ or $r_{AB} = 0$ regardless the result of $A_2 \cap B_2$ (i. e., r_2), thus $r_{AB} = r_1 \ .r \ r_2 = 0 \ .r \ X = 0$

Proof of Equation 5.2
Equation 5.2: $X \ .r \ X = X$
According to Equation 5.1, we have
$0 \ .r \ 0 = 0$ (5.9)
If $A_1 = B_1, A_2 = B_2$ (which means $r_1 = r_2 = 1$), then
$A_1 \cap B_1 = A_1 = B_1$ and
$A_2 \cap B_2 = A_2 = B_2$, therefore
$A \cap B = (A_1 \cap B_1) \cap (A_2 \cap B_2) = A_1 \cap A_2 = A$ and
$A \cap B = (A_1 \cap B_1) \cap (A_2 \cap B_2) = B_1 \cap B_2 = B$
$\Rightarrow A \cap B = A = B \Rightarrow r_{AB} = 1$ according to our encoding in Definition 5.3), thus
$1 \ .r \ 1 = 1$ (5.10)
If $A_1 \subset B_1 \ and \ A_2 \subset B_2$, then $A = A_1 \cap A_2 \subset A_1 \cap B_2 \subset B_1 \cap B_2 = B$, thus $A \subset B$, which means
$2 \ .r \ 2 = 2$ (5.11)
Likewise,
$3 \ .r \ 3 = 3$ (5.12)
Consider the case
$A_1 \nsubseteq B_1 \wedge A_1 \not\supseteq B_1 \wedge A_1 \cap B_1 \neq \emptyset$ and
$A_2 \nsubseteq B_2 \wedge A_2 \not\supseteq B_2 \wedge A_2 \cap B_2 \neq \emptyset$
Then, $A \cap B = (A_1 \cap B_1) \cap (A_2 \cap B_2) \neq \emptyset$ due to Assumption 5.8 since $A_1 \cap B_1 \neq \emptyset$ and $A_2 \cap B_2 \neq \emptyset$.
As $A_1 \nsubseteq B_1$ and $A_2 \nsubseteq B_2 \Rightarrow A_1 - B_1 \neq \emptyset$ and $A_2 - B_2 \neq \emptyset$
$\Rightarrow \exists x \in (A_1 - B_1) \cap (A_2 - B_2) \subset A_1 \cap A_2 = A$
Evidently, $x \notin B_1 \cap B_2 = B$, therefore: $A \nsubseteq B$
Likewise, $B \nsubseteq A$, thus $A \nsubseteq B \wedge A \not\supseteq B \wedge A \cap B \neq \emptyset$ or
$4 \ .r \ 4 = 4$ (5.13)
From 5.9, 5.10, 5.11, 5.12, 5.13, we have: $X . r X = X$ for $X \in \{0, 1, 2, 3, 4\}$

Proof of Equation 5.3
Equation 5.3: $X \ .r \ 1 = X$
According to Equations 5.1 and 5.2,
$0 \ .r \ 1 = 0$ (5.14)
$1 \ .r \ 1 = 1$ (5.15)
If $A_1 = B_1$ (which means $r_1 = 1$) and $A_2 \subset B_2$ (which means $r_2 = 2$),
$A = A_1 \cap A_2 \subset B_1 \cap B_2 = B$, therefore
$2 \ .r \ 1 = 2$ (5.16)
Likewise,
$3 \ .r \ 1 = 3$ (5.17)

Consider the case $A_1 = B_1$ and $A_2 \nsubseteq B_2 \wedge A_2 \ngeq B_2 \wedge A_2 \cap B_2 \neq \emptyset$
since $A_2 \cap B_2 \neq \emptyset$, we have
$A \cap B = (A_1 \cap B_1) \cap (A_2 \cap B_2) \neq \emptyset$ due to Assumption 5.8
As $A_2 \nsubseteq B_2 \Rightarrow A_2 - B_2 \neq \emptyset \Rightarrow \exists x \in A_1 \cap (A_2 - B_2) \subset A_1 \cap A_2 = A$
Evidently, $x \notin A_1 \cap B_2 = B_1 \cap B_2 = B$, therefore $A \nsubseteq B$
Likewise, $B \nsubseteq A$, thus $A \nsubseteq B \wedge A \ngeq B \wedge A \cap B \neq \emptyset$ or
$4 \cdot_r 1 = 4$ (5.18)
From 5.14, 5.15, 5.16, 5.17, 5.18, we have $X \cdot_r 1 = X$ for $X \in \{0, 1, 2, 3, 4\}$

Proof of Equation 5.4
Equation 5.4: $2 \cdot_r 3 = 4$
Without loss of generality, assume that $A_1 \subset B_1$ and $A_2 \supset B_2$ or $r_1 = 2, r_2 = 3$.
We have $A_1 \cap B_1 = A_1 \neq \emptyset$ and $A_2 \cap B_2 = B_2 \neq \emptyset$
$\Rightarrow A \cap B = (A_1 \cap B_1) \cap (A_2 \cap B_2) \neq \emptyset$ due to Assumption 5.8.
As $A_2 \supset B_2 \Rightarrow A_2 - B_2 \neq \emptyset \Rightarrow \exists x \in A_1 \cap (A_2 - B_2) \subset A_1 \cap A_2 = A$
Evidently, $x \notin B_2 \Rightarrow x \notin B_1 \cap B_2 = B$, therefore $A \nsubseteq B$
Likewise, $B \nsubseteq A$, thus $A \nsubseteq B \wedge A \ngeq B \wedge A \cap B \neq \emptyset$.
Therefore, Equation 5.4: $2 \cdot_r 3 = 4$ holds.

Proof of Equation 5.5
Equation 5.5: $X \cdot_r 4 = 4 \; if \; X \neq 0$
From 5.13 and 5.18, we have $4 \cdot_r 4 = 4$ and $1 \cdot_r 4 = 4$
Consider the case $A_1 \subset B_1$ and $A_2 \nsubseteq B_2 \wedge A_2 \ngeq B_2 \wedge A_2 \cap B_2 \neq \emptyset$ ($r_1 = 2, r_2 = 4$)
since $A_2 \cap B_2 \neq \emptyset$, we have
$A \cap B = (A_1 \cap B_1) \cap (A_2 \cap B_2) \neq \emptyset$ due to Assumption 5.8
As $A_2 \nsubseteq B_2 \Rightarrow A_2 - B_2 \neq \emptyset \Rightarrow \exists x \in A_1 \cap (A_2 - B_2) \subset A_1 \cap A_2 = A$
Evidently, $x \notin B_2 \Rightarrow x \notin B_1 \cap B_2 = B$, therefore $A \nsubseteq B$
Likewise, $B \nsubseteq A$, thus $A \nsubseteq B \wedge A \ngeq B \wedge A \cap B \neq \emptyset$ or
$2 \cdot_r 4 = 4$ (5.19)
Similarly, we can prove that
$3 \cdot_r 4 = 4$ (5.20)
From 5.13, 5.18, 5.19, 5.20, we have $X \cdot_r 4 = 4$ for $X \in \{1, 2, 3, 4\}$

Theorem 5.2. *If two multi-property sets A and B are not disjoint, we can derive their intersection based on the individual intersection of their single-property sets as* $A \cap B = (A_1 \cap B_1) \cap (A_2 \cap B_2) \cap \cdots \cap (A_n \cap B_n)$
The intersection set of A and B is thus also a multi-property set whose associated single-property sets are the intersection of the single-property sets of A and B.

Proof
Due to the commutative and associative properties of the intersection of sets [85], we have

$A \cap B = (A_1 \cap A_2 \cap \cdots \cap A_n) \cap (B_1 \cap B_2 \cap \cdots \cap B_n) = (A_1 \cap B_1) \cap (A_2 \cap B_2) \cdots \cap (A_n \cap B_n)$

5.1.3 Application of multi-property set and the $\cdot r$ operator

These tools, multi-property set and the $\cdot r$ operator, are applicable for any kinds of multi-property sets in general. For example, consider two sets of flowers in a garden based on three properties: color, number of petals, and scent
A = {color ∈ {red, blue, yellow, pink}, number of petals > 3, scent = none}
B = {color ∈ {yellow, pink}, number of petals > 5, scent = any (including scentless flowers)}
We can apply the $\cdot r$ operator to compute their relationship as follows (see Definition 5.3 and Theorem 5.1).

- For the sets corresponding to the first property (color),
  A_1 = *{red, blue, yellow, pink}*, B_1 = *{yellow, pink}* $\Rightarrow A_1 \supset B_1 \Rightarrow r_1 = A_1 \cdot_r B_1 = 3$
- For the sets corresponding to the second property (number of petals),
  A_2 = *{number of petals > 3}*, B_2 = *{number of petals > 5}* $\Rightarrow A_2 \supset B_2 \Rightarrow r_2 = A_2 \cdot_r B_2 = 3$
- For the sets corresponding to the third property (scent),
  A_3 = *{scent = none}*, B_3 = *{scent = any}* $\Rightarrow A_3 \subset B_3 \Rightarrow r_3 = A_3 \cdot_r B_3 = 2$
- The overall relationship of these two flower sets A and B is therefore
  $r_{AB} = A \cdot_r B = r_1 \cdot_r r_2 \cdot_r r_3 = 3 \cdot_r 3 \cdot_r 2 = (3 \cdot_r 3) \cdot_r 2 = 3 \cdot_r 2 = 4$
  indicating that they are intersecting, i. e., $A \nsubseteq B \wedge A \nsupseteq B \wedge A \cap B \neq \emptyset$.
  According to Theorem 5.2, their intersection is
  $A \cap B$= *{color ∈ {yellow, pink}, number of petals > 5, scent = none}*

Flowers have attributes, rules have fields. Another concrete example is described in the Section 5.2.1 (under Definition 5.6), in which match fields of OpenFlow rules are compared by these tools. More use cases are thinkable concerning policies or rules that comprise multiple properties in their condition (also, criteria or match), indicating that their *condition* component corresponds to a multi-property set. The two basic components of a rule or a policy are *condition* and *action*, depending on the domains, other components may be present, e. g., *event*, *time* (see Chapter 2.3.1). In the domain of networked systems, packet filtering rules created by the IPTables[1] tool, ACL rules, those specified in OpenFlow SDN and P4 SDN expose certain similarities in their match (or condition) component. For example, an IPTables rule in a FORWARD chain of a FILTER table looks like:

```
match criteria = {protocol = tcp, source IP = 192.168.1.0/24, destination IP = 192.168.2.0/24, destination
    ↪ port = 80}, target (action) = {ACCEPT}
```

1 https://linux.die.net/man/8/iptables

an ACL rule in a Cisco router:

```
deny udp any 192.168.200.0 0.0.0.255 lt 1024
```

which can be expressed as:

```
criteria (match) = {protocol = udp, source IP = any, destination IP = 192.168.200.0/24, destination udp
    ↪ port less than 1024}, action = {deny}
```

an OpenFlow rule:

```
priority=3, match={ipv4_src=192.168.1.0/24, ipv4_dst=192.168.2.0/24, ip_proto=6, tcp_dst=80}, action={
    ↪ output:3}
```

a P4 rule:

```
match = {ipv4.srcAddr = 192.168.1.1, ipv4.dstAddr = 192.168.1.3}, action = {forward on port 3}
```

These exemplary rules specify various header fields in their conditions (i. e., match or criteria), which are associated with multi-property sets, the $\cdot r$ operator can be employed to compute the relationship between these sets to reason about conflicts. Conceivably, the local conflicts presented in our work (cf. Chapter 4.1) occur not only within SDN rules, but also in general packet filtering rules produced by IPTables, ACL or other tools. In any case, conflicts need to be detected and resolved to ensure correct network behaviour. Our method for comparing SDN rules in the next section demonstrates the application of multi-property sets and the $\cdot r$ operator in practice.

5.2 Comparison of SDN rules

Match fields and actions can be expressed differently between SDN rules, e. g., some rule specifies only a destination IP address in its match and an output port in its actions, some shows a tuple of TCP source port and destination port for its match, its actions denote how a matched packet is modified together with an output port. We need to represent SDN rules in a uniform format in order to employ the multi-property set and the relationship combination operator $\cdot r$ for their automatic comparison. The *matchmap* and *actmap* concepts are introduced as a result.

5.2.1 Matchmap

We need to determine the relationship between rules to check for their possible conflicts. If their match components overlap, i. e., their relationship is either equal, proper subset, proper superset or intersecting, a conflict is likely.

The match component of a rule contains different header fields associated with a packet, such as, source and destination IPv4 addresses, source TCP port. It can also

contain payload patterns of the packet, e. g., POF SDN allows to specify values for the (offset, length) tuple which can match not only packet headers but also payload data. We refer to these match fields including packet headers and payload data as *packet-internal fields*, and the match space including these packet-internal fields as *packet-internal match*. Besides, information other than packet-internal fields can be used as match fields, e. g., the ingress port of a packet arriving at a switch, these fields are regarded as *non-packet-internal fields*.

Definition 5.4 (Packet-internal field). *A packet-internal field is either a header field or a pattern used to match the packet's payload data. It is also a match field. The definitions of packet and header field follow those described in the OpenFlow Switch version 1.5.1 specification [78]. We extend the definition of match field from this specification to include also the pattern of packet's payload data.*

Definition 5.5 (Non-packet-internal field). *A non-packet-internal field is a match field that does not belong to the set of packet-internal fields. It can be technology specific, e. g., an ingress port match field in OpenFlow SDN.*

Definition 5.6 (Packet-internal match). *A packet-internal match of a rule is a multi-property set corresponding to the match part of that rule that contains only packet-internal fields.*

Rule matches are essentially multi-property sets and we can compare them using the multi-property set comparison approach mentioned in Section 5.1.2. However, it is necessary to represent them uniformly before comparing since each rule match can be expressed differently. For example, consider the following two OpenFlow rules' matches $m1$ and $m2$:

```
m1: in_port = 3, ether_type = 0x0800, ipv4_dst = 192.168.2.0/25
m2: ether_type = 0x0800, ipv4_src = 192.168.1.0/24, ipv4_dst = 192.168.2.0/24
```

The match $m1$ corresponds to a multi-property set having the $in\_port, ether\_type$ and $ipv4\_dst$ properties, $m2$ has $ether\_type, ipv4\_src$ and $ipv4\_dst$ properties. These two multi-property sets associated with $m1$ and $m2$ need to be represented in the same format for their comparison. Using their "denominator" of $in\_port, ether\_type, ipv4\_src, ipv4\_dst$, we can describe $m1$ and $m2$ as:

```
m1: in_port = 3, ether_type = 0x0800, ipv4_src = any, ipv4_dst = 192.168.2.0/25
m2: in_port = any, ether_type = 0x0800, ipv4_src = 192.168.1.0/24, ipv4_dst = 192.168.2.0/24
```

The relationship of $m1$ and $m2$ is then determined based on their single-property set relationships as follows.

- The relationship of the first property, being in_port, of $m1$ and $m2$ is: $r_1 = 2$ as the set containing only an element having the value of 3 is a proper subset of the other set containing elements of any value.

- The relationship of the second property, being $ether\_type$, of $m1$ and $m2$ is: $r_2 = 1$ (equal).
- The relationship of the third property, being $ipv4\_src$, of $m1$ and $m2$ is: $r_3 = 3$ (proper superset).
- The relationship of the fourth property, being $ipv4\_dst$, of $m1$ and $m2$ is: $r_4 = 2$ (proper subset).
- The overall relationship of $m1$ and $m2$ is $r_1 \cdot_r r_2 \cdot_r r_3 \cdot_r r_4 = 2._r 1._r 3._r 2 = 4$, which means $m1$ and $m2$ having the $intersecting$ relationship.

The above example raises the issue: which denominator, understood as all properties or all fields that the rule matches in comparison have, should we use? It is obvious that the minimal denominator would be the union of all set of the rules' match fields. A comprehensive denominator would cover all possible match fields, including all fields in any packet's headers and fields specific to the technology in use. As an example, OpenFlow Switch version 1.5.1 specifies up to 45 match fields, the comprehensive denominator for this OpenFlow version, thus, has 45 match fields. In practice, we may be interested in handling conflicts for certain kinds of traffic, e. g., TCP, UDP, SCMP traffic while ignoring others, e. g., ARP, ICMP. In this case, we can choose a simpler denominator bound to the kinds of traffic that we are interested in, rather than an all-inclusive one which may be too redundant and degrade the performance in detecting conflicts. In principle, the denominator needs to cover all header fields at all layers of the interested traffic kinds specified by the technology in use so that the conflicts can be discovered for cross-layer traffic. For example, the denominator specified for TCP traffic using IPv4 implemented in OpenFlow version 1.5.1 looks like:
$in\_port, eth\_src, eth\_dst, eth\_type, ipv4\_src, ipv4\_dst, ip\_proto, tcp\_src, tcp\_dst$
We can use this denominator to detect the overlap between the following two matches $m3$ and $m4$, which feature fields of different layers:

```
m3: in_port = 3, ether_type = 0x0800, dst_ipv4 = 192.168.2.0/25, ip_proto = 6, dst_TCP = 80
m4: ether_type = 0x0800
```

In the same manner as we used to detect the relationship between $m1$ and $m2$, we can see that $m3$ is a proper subset of $m4$. If we use a "narrower" denominator just covering the OSI layer 3 and layer 4 fields, the rule having the match $m4$ can be ignored as $m4$ specifies only a header field belonging to the OSI layer 2, which results in ignoring a possible conflict between that rule and the rule having the match $m3$. In practice, it is sufficient to choose a "compact" denominator covering the match fields of all rules if this information is available beforehand.

We refer the mapping result of a rule match to a denominator as a *matchmap* and for the sake of lucidity, the denominator is referred as *matchmap template*. In the subsequent sections, we encode the any value in the *matchmap* as -1.

Definition 5.7 (Matchmap template). *A matchmap template is a tuple containing the names of packet-internal fields and non-packet-internal fields.*

Definition 5.8 (Matchmap). *A matchmap is the mapping result of a rule match to a matchmap template according to a predefined mapping function. In other words, the matchmap is a tuple of values after the order and the format of the matchmap template, each value is the value of the field in the rule match whose name is specified in the matchmap template.*

Definition 5.9 (Mapping function for matchmap). *Given a matchmap template* $mmt : (f_1, f_2, \cdots, f_n)$*, in which* f_i *stands for field i, which is either a packet-internal or non-packet-internal field,*
the corresponding matchmap mm *has the format:* $mm : (vf_1, vf_2, \cdots, vf_n)$*, where* vf_i *is the value of field i,*
we define the mapping function f_{mat} *to obtain the matchmap* mm *from a rule match* rm *and the matchmap template* mmt*, i. e.,* $mm = f_{mat}(rm, mmt)$ *as follows:*

- *If the value of the field* f_i *is specified explicitly in the rule match* rm*, say* v*, then* $vf_i = v$*.*
- *If the value of the field* f_i *is not specified in the rule match* rm*, then* $vf_i = -1$ *where -1 indicates an* any *value.*

We notice that the rule match may specify a field differently in different SDN implementation, e. g., OpenFlow SDN represents an IPv4 source address explicitly by its name, while POF SDN expresses that in a tuple, such as <offset = 208 bits, length = 32 bits>. In any case, while programming the data plane, the administrator has to figure out and be aware of how to specify a match field for a certain packet header or payload data pattern, which warrants the feasibility in mapping a rule match to a matchmap template.

If the match spaces represented by the matchmap of two rules are not disjoint, we can calculate their intersection, which is also a match space, after the method to derive the intersection of two non-disjoint multi-property sets in Theorem 5.2.

We give additionally the definition of the term *packet-internal matchmap* that is frequently used in the subsequent sections.

Definition 5.10 (Packet-internal matchmap). *A packet-internal matchmap of a rule is a multi-property set corresponding to the packet-internal match of that rule (Definition 5.6). It is formed from the matchmap of that rule by assigning all of its non-packet-internal fields the value -1 while leaving its packet-internal fields unchanged. The value of -1 in the matchmap indicates an* any *value.*

5.2.2 Actmap

SDN rules can have actions that modify matched traffic before passing it to the next device. Consequently, a packet can match rules with completely disjoint packet-

internal match spaces on different devices along its forwarding path. Our approach to detect distributed conflicts (see Section 5.6) requires keeping track of the rule match transformation to establish the connection between rules in different devices. Two rules in two directly connected devices are said to have a connection if there exists a packet that is matched by both of them. This connection can be determined by combining the match and action components of the rule in the first device, the result is compared with the match component of the other rule in the second device. This is elaborated in Section 5.4.1.

Similar to the rule match, the action component can be specified differently from rule to rule. We introduce the *actmap* notion to represent the rule action uniformly which facilitates not only its combination with the rule match based on the *matchmap* but also the comparison of actions between rules to reason about local conflicts. *Actmap* is the mapping result of the rule action to an *actmap template* which defines the *actmap*'s components. An all-embracing *actmap template* corresponding to a given *matchmap template* would contain all packet-internal fields of that *matchmap template* so that any change of packets can be expressed in the associated *actmap*. In practice, it is sufficient to choose a "compact" *actmap template* covering all packet-internal fields that are modified in all rules if this information is known in advance. The *actmap template* includes additional components reflecting how the matched packet is handled, e. g., being forwarded out of a port, being dropped.

Definition 5.11 (Actmap template)**.** *An actmap template is a tuple containing the names of packet-internal fields and the names of actions.*

Definition 5.12 (Actmap)**.** *An actmap is the mapping result of the rule action to an actmap template according to a predefined mapping function.*

Definition 5.13 (Mapping function for actmap)**.** *Given the actmap template, named* amt*:*
$pf_1, pf_2, \cdots, pf_n, a_1, a_2, \cdots, a_p$
The actmap, named am*, has the format:*
$vpf_1, vpf_2, \cdots, vpf_n, va_1, va_2, \cdots, va_n$
In which:
pf_i*: packet-internal field i, which is a header field in a packet or a pattern of the payload data, e. g., Ethernet source address, IPv4 destination address.*
a_i*: name of action i, which is other than modifying packet-internal fields, e. g., to forward out of port 2*
vpf_i*: value of the packet-internal field i after being modified.*
va_i*: value of the action i*
We define the mapping function f *to obtain the actmap* am *from the rule action* ra *and the actmap template* amt*, i. e.,* $am = f(ra, amt)$*, as follows:*

- *If the value of the field* pf_i *is specified explicitly in the rule action* ra*, say* v*, then* $vpf_i = v$*.*

- *If the value of the field pf_i is not specified in the rule action ra, then $vpf_i = -1$ where -1 at the position of the packet-internal field of an actmap indicates a $no\ change$ value.*
- *The mapping of the action types other than modifying packet-internal fields can be customized flexibly for the implementation. For instance, one can define the mapping function for the action of forwarding the matched packet out of a port list as a list of port number, a drop action as an empty list.*

Definition 5.14 (Output matchmap). *The output matchmap of a rule is the matchmap obtained by combining the matchmap and actmap of that rule according to a predefined function.*

Definition 5.15 (Combination function for output matchmap). *Given the matchmap mm:*
$mm : mnpf_1, mnpf_2, \cdots, mnpf_m, mpf_1, mpf_2, \cdots mpf_n$
and the all-embracing actmap am: $apf_1, apf_2, \cdots, apf_n, a_1, a_2, \cdots, a_p$
the corresponding output matchmap omm has the format:
$omm : (onpf_1, onpf_2, \cdots, onpf_m, opf_1, opf_2, \cdots opf_n)$
In which:
$mnpf_i$: non-packet-internal field i of the matchmap, e. g., ingress port of the packet, metadata and pipeline fields in OpenFlow SDN
mpf_i: packet-internal field i of the matchmap, which is a header field in a packet or a pattern of the payload data, e. g., Ethernet source address, IPv4 destination address
apf_i: packet-internal field i of the actmap
a_i: name of action i, which is other than modifying packet-internal fields
$onpf_i$: output of the non-packet-internal field i
opf_i: output of the packet-internal field i
We define the combination function f_{mat_act} to obtain the output matchmap omm from the matchmap mm and the actmap am, i. e., $omm = f_{mat\_act}(mm, am)$, as follows:

- *the non-packet-internal fields of omm have the same value as those of mm, i. e., $onpf_i = mnpf_i,\ i \in \{1..m\}$.*
- *if the packet-internal field of the actmap am is other than -1 (-1 in the actmap means no change), then the corresponding packet-internal field of omm has the same value as that of the actmap, i. e., $opf_i = apf_i\ if\ apf_i \neq -1$*
- *if the packet-internal field of the actmap am is -1, then the corresponding packet-internal field of omm has the same value as that of the matchmap mm, i. e., $opf_i = mpf_i\ if\ apf_i = -1$*
- *A "compact" actmap instead of the all-embracing actmap can be used. In this case, the "compact" actmap is first converted to the "all-embracing" one by inserting the value -1 (no change) to the positions corresponding to the unspecified packet-internal fields of the "compact" actmap compared to the "all-embracing" one. The combining steps above can then be applied to calculate the output matchmap.*

Example:

```
rule match m: in_port = 3, ether_type = 0x0800, ipv4_src = 192.168.1.1, ipv4_dst = 192.168.2.2, ip_proto =
    ↪ 6, tcp_dst = 80
rule action a: change ipv4_dst to 192.168.4.4, change tcp_dst to 8080, send packet out of ports 2 and 3
```

matchmap template mmt:in_port, eth_src, eth_dst, eth_type, ipv4_src, ipv4_dst, ip_proto, tcp_src, tcp_dst
actmap template amt: eth_src, eth_dst, eth_type, ipv4_src, ipv4_dst, ip_proto, tcp_src, tcp_dst, outport list
By mapping the rule match m and action a to the matchmap and actmap template mmt and amt, we have:

```
rule matchmap mm: 3, -1, -1, 0x0800, 192.168.1.1, 192.168.2.2, 6, -1, 80
rule actmap am: -1, -1, -1, -1, 192.168.4.4, -1, -1, 8080, [2, 3]
```

The output matchmap omm obtained by combining the matchmap mm and actmap am, $omm = f_{mat\_act}(mm, am)$, is:

```
output matchmap omm: 3, -1, -1, 0x0800, 192.168.1.1, 192.168.4.4, 6, -1, 8080
```

In addition, the following terms are frequently used in bulding the rule graph for detecting distributed conflicts.

Definition 5.16 (Output match space)**.** *The output match space of a rule is a multi-property set yielded from the output matchmap of that rule.*

Definition 5.17 (Output packet-internal match space)**.** *The output packet-internal match space of a rule is a multi-property set containing only the packet-internal fields of the output matchmap of that rule. Its corresponding matchmap, named output packet-internal matchmap, is formed from the output matchmap of that rule by assigning all of its non-packet-internal fields the value -1 while leaving its packet-internal fields unchanged. The value of -1 in the matchmap indicates an any value.*

In comparing the match components of rules in different devices, it does not make sense to compare their non-packet-internal fields, e. g., ingress port, only their packet-internal fields matter.

Definition 5.18 (Common match space)**.** *A common match space of rule r_1 in device 1 and rule r_2 in device 2, where r_1's action is to forward its matched traffic to device 2, is the intersection of the two match spaces, the first corresponds to the output packet-internal matchmap (Definition 5.17) of r_1, the second corresponds to the packet-internal matchmap (Definition 5.10) of r_2. This common match space is therefore associated with the intersection of the two mentioned matchmaps and the intersection is named common matchmap. This definition also applies in case r_1 and r_2 are in two rule tables of a device if that device contains multiple rule tables.*

5.3 Rule database and topology encoding

The rule database and the topology encoding are crucial for the conflict detection. The rule database, mentioned in our subsequent algorithms as the variable RDB, is maintained to be consistent with rules in the data plane so that the conflict detection can rely on to provide precise results. A rule in a rule table of an SDN device is stored in the rule database by the tuple (device ID, table ID, rule ID), in which

- device ID is the identifier of the SDN device, e. g., the datapath ID in OpenFlow SDN,
- each SDN device can hold multiple rule tables, leading to the use of the table ID corresponding to the ordinal number of the table holding the rule under consideration,
- each rule is numbered uniquely when it is added to the rule database, this number is used as the ID for that rule.

The tuple (device ID, table ID, rule ID) identifies a rule in the whole data plane's rule set. Each rule is maintained in the database with its matchmap, actmap based on the chosen matchmap and actmap templates, and a list of next hops that can be inferred from the topology information. In essence, an entry in the rule database contains the following information:
(device ID, table ID, rule ID, priority, matchmap, actmap, list of next hops, original rule encoding in the data plane)
The original rule encoding in the data plane is useful in determining the rule in the device if it needs to be manipulated, e. g., for resolving conflicts, or in case more information is required for certain purposes. The rule encoding in OpenFlow SDN version 1.3 [79] contains: *cookie, cookie mask, idle timeout, hard timeout, priority, match, actions*, which is employed for illustration of the original rule in the rule database.

A rule can be installed by control applications or by an administrator, and removed also by these actors besides the rule timeout. Interestingly, the installation of a new rule may trigger the removal of an existing one depending on the implementation of SDN devices. For instance, OpenFlow switch based on Open vSwitch (version 2.6.2) lets a new rule overwrite an existing one if they both have the same match fields and priority i. e., these rules expose either a *redundant* or a *correlation* local conflict (cf. Chapter 4.1), consequently, the existing rule gets removed. The rule database needs to be updated in the presence of these events. Besides the rule database, the conflict database (for local, distributed and hidden conflicts) may need to be updated due to the introduction or removal of a rule as new conflicts can arise or some existing conflicts can be purged.

The network topology information, referred to in our algorithms as the variable $TOPO$, can be maintained automatically by the topology discovery service, which is generally a basic function in SDN. The manual maintenance is also possible but could

require much effort and lead to high delay in detecting conflicts in case of incidents causing topology changing, e. g., a device or a link is down. There are also different ways to encode the topology, we opt for the directed multi-graph (also called Multi-digraph in literature) in our prototype (see Chapter 6.1) that allows the encoding of topology with multiple parallel connections between a pair of devices. Each vertex in this graph corresponds to an SDN device with its ID, an edge represents the physical link between two devices, the port through which a link is connected to a device is maintained as the edge's attribute. In addition, the ARPcache control application (see Chapter 3.5.2), which caches the MAC and IPv4 address mapping of end-points and their connections to SDN device, can be used to infer the device and its port that an end-point connects to.

5.4 Rule graph

We can conclude the existence of anomalies, and thus conflicts, based on the observation of how a certain packet being handled in the network and which rules being involved. The rules that together handle the same traffic flow are said to be connected. In this section, we first show how to establish connections between rules in two directly connected SDN devices or in two rule tables. We apply this approach recursively in building a rule graph, which enables the reasoning about conflicts presented in subsequent sections.

5.4.1 Establishing connections between rules

Definition 5.19 (Connection of rules)**.** *Two rules in two directly connected devices are said to have a connection if there exists a packet that is matched by both of them, i. e., that packet is matched by the rule in the first device, possibly being transformed and then sent out to the second device, and is matched by the other rule there. This also applies for two rules in two rule tables of the same device. The connection is directed and the direction is from the first rule to the second one.*

Apparently, after being processed by the first rule, that packet belongs to the common match space (see Definition 5.18) of these two rules. Moreover, it is important to note that if they reside in different devices and the ingress port is specified in the second rule, this port must correspond to the output port stated in the first rule's action according to the network topology in concern, in other words, there needs to be a (physical) link between two devices connecting the output port of the first rule and the input port of the second rule.

The knowledge of the network topology is required for establishing rule connections. This knowledge can be obtained in advance and maintained by the network operator or by using the network topology discovery service if available.

A rule in an SDN device's rule table can become inactive if it is shadowed or made redundant due to local conflicts with other rules in that same rule table. By

conducting local conflict detection (which does not require the use of the rule graph, see Section 5.5), we are able to ascertain if a rule is active. We examine only the connections between active rules as they are involved in the traffic forwarding process which can cause distributed conflicts.

Definition 5.20 (Active and inactive rule)**.** *A rule is said to be active if it is not shadowed or made redundant due to local conflicts with other rules in the same rule table. It is an inactive rule otherwise.*

Consider a network part containing three SDN devices, being switches S1, S2 and S3 shown in Figure 57, each has only one rule table, named *Table 0*, in which its rules reside, S1 has m rules: r_{11}, $r_{12} \cdots r_{1m}$, S2 has n rules: r_{21}, $r_{22} \cdots r_{1n}$, S3 has p rules: r_{31}, $r_{32} \cdots r_{3p}$, each rule has its $matchmap$ and $actmap$ as elaborated in Sections 5.2.1 and 5.2.2. Assume that r_{11} is an active rule and its action is to forward matched traffic out of port 2 of switch S1, which is connected to port 3 of of switch S2, in establishing the connections between r_{11} and rules in switch S2, the following aspects need to be taken into account.

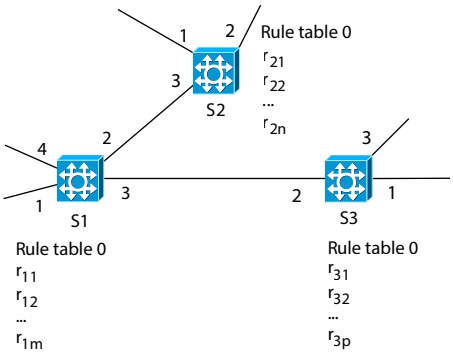


Figure 57: A network part containing three switches S1, S2 and S3. The numbers around a switch indicate the ports from which it connects to other switches or end-points.

- The matched traffic of r_{11} can be transformed by its action. Using the *combination function for output matchmap* in Definition 5.15, we can obtain the *output matchmap omm* (see Definition 5.14) of r_{11} that corresponds to the match space containing all transformed traffic originally matched by r_{11}'s match component.
- If rule r_{21} in switch S2 handles a packet passed by r_{11}, then the following propositions hold:
  - r_{21} is an active rule (see Definition 5.20).
  - r_{11} and r_{21} can have different packet-internal match spaces (see Definition 5.6) due to r_{11}'s action modifying the matched packets. Moreover, they can be installed by different control applications.
  - The packet-internal match space of r_{21} overlaps with the output packet-internal match space (see Definition 5.17) of r_{11}, i. e., their relationship is either equal, proper subset, proper superset, or intersecting. This is evident as there exists a packet matched by both r_{11} and r_{21}.
  - If r_{21} specifies the *ingress port* match field, that port must correspond to the port from which switch S2 connects to the output port stated in the r_{11}'s action. In

this case, as we assume that the output port of r_{11} is 2, the ingress port of that rule must be 3 according to the topology in Figure 57.

- Different packets handled by r_{11} can be matched by different rules in switch S2. According to our definition of rule connection (Definition 5.19), this means there can be multiple connections between r_{11} and rules in switch S2. An exemplary case is that the output packet-internal match space (Definition 5.17) of r_{11} is a proper superset of the packet-internal match space (Definition 5.6) of r_{21}, rule r_{21} generalizes rule r_{22} according to the *generalization* local conflict pattern, i. e., r_{22} has higher priority but its match space is a proper subset of r_{21}'s and their actions are different. In this case:

  – a packet belonging to the match space of r_{22} will be handled by this rule as it has higher priority than r_{21}, this packet belongs to the common match space (Definition 5.18) of r_{11} and r_{22}, say $pfmm_{11\_22}$ (pfmm: packet-internal field matchmap).
  – a packet belonging to the match space of r_{21} but not r_{22}'s will be handled by r_{21}, this packet belongs to the common match space of r_{11} and r_{21}, say $pfmm_{11\_21}$.

  We notice in this case that $pfmm_{11\_22}$ is a proper subset of $pfmm_{11\_21}$, a packet belonging to $pfmm_{11\_22}$ thus also belongs to $pfmm_{11\_21}$, here the priority of r_{21} and r_{22} decides which rule is chosen to match that packet.
  In general, if the output packet-internal match space (see Definition 5.17) of r_{11} overlaps with both packet-internal match spaces (Definition 5.6) of the active rules r_{21} and r_{22}, it can lead to the case where different packets handled by r_{11} can be matched by different rules in switch S2 and this can be generalized for more than two rules in the next hop device, here being switch S2.
  Exceptional case: There are also exceptions happening in case the common match space (Definition 5.18) of r_{11} with a rule in S2 is a subset of that of r_{11} with another rule in S2, and the latter rule has higher priority, then all packets matched by r_{11} will be handled only by the latter rule, this case is illustrated in Figure 58. It appears more complicated when more rules get involved, yet, the rule priority is again the decisive factor in this case.
  It is clear which rule in switch S2 will handle a packet matched by r_{11} if the match spaces of r_{21} and r_{22} are disjoint. We discuss further the case in which the match spaces of r_{21} and r_{22} are not disjoint, which means these two rules expose some kind of local conflicts (see Chapter 4.1). As we assume that both rules are active, the local conflicts of classes *redundancy* and *shadowing* are excluded between them. Which rule will match a packet belonging to the match space of these two rules, depending on which class of local conflicts they expose and possibly the implementation of the data plane.

  - If r_{21} and r_{22} have different priority, the packet is matched against the higher priority one.
  - If the actions of r_{21} and r_{22} are the same, these two rules expose the local conflict of *overlap* class whose consequence is benign.
  - If they have equal priority and intersecting match spaces, i. e., they expose a *correlation* local conflict, it may depend on the SDN technology and the implementation of the data plane to decide which rule dominates, for example, OpenFlow Switch 1.5.1 [78] leaves it undefined and different network vendors may implement the SDN device's behaviour differently, e. g., the newer rule dominates. We notice that this case is critical and recommend the local conflict of this kind to be resolved immediately so that the network behaviour is deterministic, as the network devices can come from different vendors and the network behaviour in this case can become inconsistent.

- If the action of rule r_{11} is to forward its matched traffic out of ports 2 and 3, then besides checking the above points in switch S2, we have to repeat that also in switch S3 as r_{11} has two next hop devices, being switches S2 and S3 in this case according to the topology in Figure 57.
- In case each switch has multiple rule tables, the above points also apply. A rule table can be considered as the next hop of a certain active rule if its action is to submit the matched packets to that table.

In the next section, we show the procedure of building a rule graph to reason about conflicts based on establishing the connection of each rule pair while taking into consideration the above discussed points.

5.4.2 Building the rule graph

A packet can traverse different devices in the data plane and be handled by a sequence of rules in these devices. According to our definition of the connection between two rules (Definition 5.19), each pair of adjacent rules in this rule sequence are connected. It can be seen as if this rule sequence grows from the sequence containing only the first rule by repeatedly searching for the connected rule of the last rule in the sequence and concatenating it there.

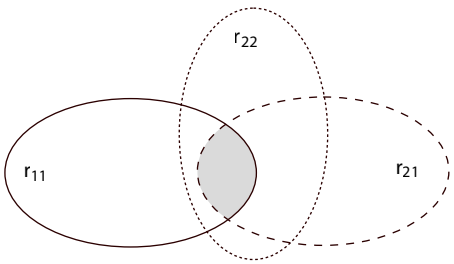


Figure 58: Exceptional case: common match space of r_{11} and r_{21} is a subset of that of r_{11} and r_{22}. r_{21} and r_{22} are both active rules, the latter rule has higher priority.

A rule graph can be built based on the connections of rules. The main idea in reasoning on conflicts from the rule graph is to encode in that graph all possible paths, being a sequence of rules, that various slices of traffic can take once they

enter the network, then point out the questionable paths according to the defined properties of distributed conflicts. Further analysis of this set of paths to confirm the presence of distributed conflicts may be carried out if necessary. For example, if there exists a path containing a loop manifested by the repeated occurrence of the same rule in it, we can conclude with high certainty a conflict case. We encode the information about paths as edge attributes throughout the process of building the rule graph. A path grows together with the growth of a corresponding rule sequence when connections between rules are formed.

Figure 59 illustrates the edge attributes in a simple rule graph. For simplicity, we denote a rule as r_{xy} where x represents the device ID and y the rule ID in Table o of that device, we assume that there is only a rule table, named *Table o*, in each device (with this assumption, the full form of rule r_{xy} would be $(x, 0, y)$). In this graph, rule r_{11} connects to rules r_{21} and r_{22}, a packet matched by r_{11} will then be handled by either r_{21} or r_{22}, depending on the match space and the priority of r_{21} and r_{22} as elaborated in Section 5.4.1. An edge from r_{11} to r_{21} is added to the rule graph with the attribute including *(path* = (r_{11}, r_{21})*, matchmap* = $matchmap_{11\_21}$*, priority = priority of* r_{21}*)*, in which $matchmap_{11\_21}$ is formed by the intersection between the output packet-internal matchmap of r_{11} and the packet-internal matchmap of r_{21} (see Definition 5.17). Apparently, if there exists a packet handled by both r_{11} and r_{21}, that packet will belong to the match space represented by $matchmap_{11\_21}$. Likewise, another edge from r_{11} to r_{22} is formed with the attribute *(path* = (r_{11}, r_{22})*, matchmap* = $matchmap_{11\_22}$*, priority = priority of* r_{22}*)*, in which $matchmap_{11\_22}$ is the intersection between the output packet-internal matchmap of r_{11} and the packet-internal matchmap of r_{22}; a packet processed by both r_{11} and r_{22} will belong to the match space $matchmap_{11\_22}$. Further edges from r_{21} to r_{41}, r_{22} to r_{31}, and r_{31} to r_{41} are added to the rule graph. The matchmap associated with a path is calculated cumulatively from all rules in that path. For instance, in the edge from r_{21} to r_{41}, $matchmap_{11\_21\_41}$ of the path containing three rules r_{11}, r_{21} and r_{41} is calculated as follows:

- $matchmap_{11\_21}$ = intersection between the output packet-internal matchmap of r_{11} and the packet-internal matchmap of r_{21} ($matchmap_{11\_21}$ is also the matchmap of the edge from r_{11} to r_{21})
- omm_{11_21} = output matchmap obtained by combining $matchmap_{11\_21}$ with r_{21}'s action (see Definition 5.15)
- $matchmap_{11\_21\_41}$ = intersection between omm_{11_21} and the packet-internal matchmap of r_{41}

A packet processed by the three rules r_{11}, r_{21} and r_{41} will belong to the match space $matchmap_{11\_21\_41}$ after being handled by r_{21}. This example shows the role of the matchmap in the attribute of an edge: it is necessary in growing a path. If omm_{11_21} does not intersect with the packet-internal matchmap of r_{41}, the path containing r_{11},

r_{21} and r_{41} will not exist. However, the edge from r_{21} to r_{41} can still be present if the output packet-internal machmap of r_{21} overlaps with the packet-internal matchmap of r_{41}. By encoding the edge attribute in this manner, we can observe the following benefits:

- Once a path exhibits a problem, e. g., the same rule occurs twice within it which indicates a traffic loop, we can conclude the presence of anomalies and all the involved rules.
- Given a packet and its entrance point in the rule graph, we can infer its path in the network based on the matchmap and priority of the edges in the rule graph, and thus determine its fate (e. g., being dropped, stuck in a loop, delivered successfully to an end-point) without having to perform the matching of the packet against each individual rules in various network devices.

The priority in the edge attribute is required for the cases mentioned in Section 5.4.1. For instance, a packet matched by r_{11} can then be processed by either r_{21} or r_{22},

- if that packet belongs only to $matchmap_{11\_21}$ and not $matchmap_{11\_22}$ or vice versa, it is clear which rule will take over it,
- if it belongs to the intersection between $matchmap_{11\_21}$ and $matchmap_{11\_22}$ in case these two matchmaps overlap, the priority in the edge attribute will decide. If these two edges from r_{11} to r_{21} and r_{22} have the same priority, then the implementation policy in the (virtual) hardware will decide, e. g., the newer rule dominates.

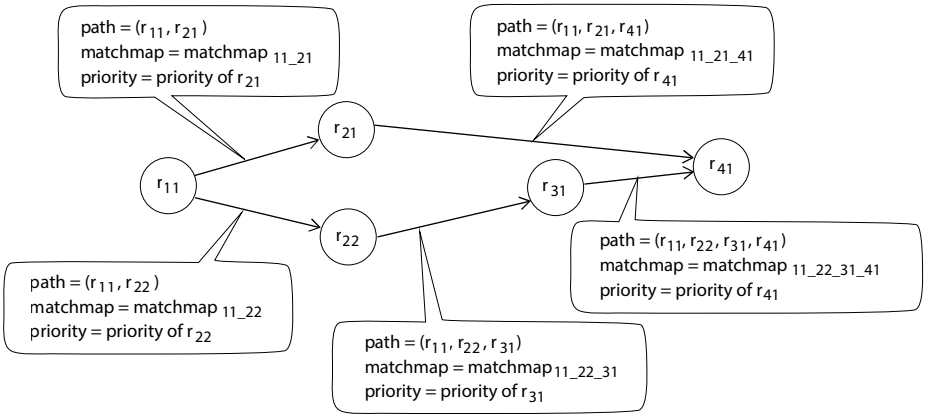


Figure 59: Illustration of edge attributes in a simple rule graph

For simplicity, we illustrate only a tuple (path, matchmap, priority) for the attribute of each edge in the rule graph in Figure 59. In fact, an edge's attribute contains a list of such tuples. More details are presented in the next section concerning the introduction of a new rule in the rule graph.

Unlike the network topology expressed as a directed multi-graph to allow multiple parallel edges between a pair of vertices, the rule graph is encoded by a directed graph, thus, there is only a single directed edge between a pair of vertices. For reasoning about conflicts, it makes no sense to have multiple directed connections between the same two rules in a rule graph.

We present in the following the two basic actions in building the rule graph, including i) adding a new rule to and ii) removing an existing rule from the rule graph, and provide a stepwise example for their illustration.

Introduction of a new rule

The central principle in adding the new rule NR into the rule graph is that, if NR is an active rule (i. e., it is not shadowed or made redundant by another rule) and is not yet in the rule graph, it will become a new vertex there, then all edges whose attributes are impacted by the introduction of NR must have their attributes updated. These edges include:

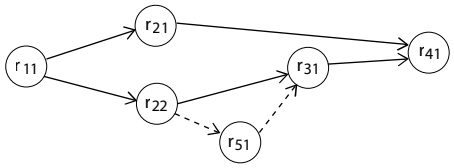


Figure 60: A scenario of adding the new rule r_{51} to the existing rule graph (in Figure 59), in which there exist connections from r_{22} to r_{51} and from r_{51} to r_{31}

- the new edges from some existing rules to NR and from NR to the existing rules if there exist connections between NR and these rules, these are the edges from r_{22} to r_{51} and from r_{51} to r_{31} in the scenario depicted in Figure 60 (as mentioned above, r_{xy} represents the rule number y in table 0 of device x and its full form in the rule graph is $(x, 0, y)$),
- the existing edges if there exist packets handled by the new rule and flowing along the paths containing these edges. The edge from r_{31} to r_{41} in Figure 60 is a candidate one as there can be packets flowing on the new path containing this edge, being $(r_{11}, r_{22}, r_{51}, r_{31}, r_{41})$, besides the existing one $(r_{11}, r_{22}, r_{31}, r_{41})$. The attribute of this edge would then be the tuple list
  [(path = $(r_{11}, r_{22}, r_{31}, r_{41})$*, matchmap =* $matchmap_{11\_22\_31\_41}$*, priority = priority of* r_{41}*),*
  (path = $(r_{11}, r_{22}, r_{51}, r_{31}, r_{41})$*, matchmap =* $matchmap_{11\_22\_51\_31\_41}$*, priority = priority of* r_{41}*)].*

To simplify the first illustration of the rule graph, we leave out some details in Figures 59 and 60, for example, only one tuple in the tuple list of each edge's attribute is sketched. In fact, the attribute of an edge, being a list of tuples, contains at least a "minimal" tuple whose path is composed of two vertices making that edge (except for that edge ending at a vertex representing a loop or a drop action). Obviously, the minimal tuple has the most general matchmap compared to other tuples in the same edge, i. e., its matchmap is equal to or a superset of other tuples' matchmap. This minimal tuple is always present in the edge's attribute and has the benefit in

growing a path or matching a packet to find its path via the rule graph: we need to perform certain comparison related to the matchmap of edges' tuples in these cases, if the comparison reveals a mismatch with the minimal tuple's matchmap, the edge containing this tuple can be excluded without having to consider other tuples' matchmaps of that edge.

There is no redundant tuple in the attribute of an edge. A tuple is redundant if it has the same matchmap as other tuple and its path is a subset of or equal to the other's. The minimal tuple whose path consisting of only two vertices is always present in an edge's attribute and is not deemed redundant.

A vertex in the rule graph can denote a normal rule represented by the tuple (device ID, table ID, rule ID), an end-point by the tuple (0, end-point's IP), a drop action in the form of (device ID, table ID, -1) or a loop as (device ID, table ID, -4). In our prototypical implementation, we use additionally the tuple (device ID, table ID, -2) to embody the drop action due to an invalid (non-existent) output port, and the tuple (device ID, table ID, -3) for a rule without connected rules in its next hop, these cases are mainly related to a bug rather than a conflict and thus are not mentioned further. A path is a sequence of vertices, each corresponds to a rule, except for the last one which can be of any vertex type, i. e., rule, end-point, drop action or loop.

For ease of reference, we illustrate the content of the common variables used in our algorithms in Figure 62.

Algorithm 1 presents the steps conducted to update the rule graph when a new rule NR is added to rule table T of device D. Each rule corresponds to a vertex in the rule graph and is represented by the tuple (device ID, table ID, rule ID) similar to how it is held in the rule database RDB. For each table $T\_$ of device $D\_$, we maintain a list of rules forwarding the matched traffic to that table. The variable holding this list of rules is named $LR2DT$ (a mapping of lists of rules to devices' tables) and the rules sending traffic to table $T\_$ of device $D\_$ is accessed by $LR2DT[D\_][T\_]$ (the pseudo-code's style follows that of the python programming language, in this case, $LR2DT$ is similar to a dictionary data structure in python). When a new rule is added to the rule database RDB, based on its actions and the topology encoding $TOPO$, its next hops are extracted and the mapping $LR2DT$ is updated with this information. We notice that the next hops of a rule can be i) tables in the same device, ii) other devices (i. e., table 0 of other devices), iii) end-points, iv) controller, or v) a drop action; the mapping $LR2DT$ is updated in the first and second cases.

For the introduction of the new rule (D, T, NR), the following cases are considered in Algorithm 1.

- If there is no rule sending traffic to table T of device D, then (D, T, NR) will be a new vertex without any vertex connecting to it. However, this new vertex can connect to other existing vertices (lines 1–4), this is checked and the rule graph is updated via the invocation of the function *ADD_RULE_TO_RULE_GRAPH*. The

same situation occurs if there are rules sending traffic to table T of device D, but these rules do not connect to NR (lines 52–56).
- If there exists a rule (D_R, T_R, R) connecting to (D, T, NR):
  - in case there is no rule connecting to (D_R, T_R, R), we create a new edge between these two rules and let the path containing them grow through the new vertex (D, T, NR) (lines 19–25),
  - otherwise, (D_R, T_R, R) is the ending point of an edge, we let the paths of this edge grow through the new vertex (D, T, NR) (lines 26–49).

The function *ADD_RULE_TO_RULE_GRAPH* determines and updates the attributes of the edges influenced by the new rule (D, T, NR) by calling itself recursively until one of the following conditions holds.

- The next hop of the currently considered rule is empty, i. e., its action is to drop the matched traffic (lines 59–64).
- The next hop of the currently considered rule is an end-point (lines 65–74).
- The loop occurs (lines 103–113). In this case, a rule appears twice in some path.
- The currently considered rule has no connection with any rule in the next hop (lines 122–124).

During the rule graph building process, the attribute of an edge can be updated multiple times by appending new tuples to the tuple list, as long as each tuple is unique in this list (lines 114, 117). As mentioned above, a tuple is added to the tuple list of an edge if it is not redundant in that list. Moreover, any existing tuple being made redundant by the appearance of the new tuple must be removed.

Exceptional case: In this algorithm, we do not directly exclude the exceptional case discussed in Section 5.4.1 in which the common match space (Definition 5.18) of certain rules, say rules r_1 and r_2 is a subset of that of r_1 and r_3, where r_2 and r_3 reside in the same next hop of r_1 and r_3 has higher priority than r_2. It becomes more complicated when the case occurs on more than two rules in the next hop. Our approach to reduce the complexity in this comparison is to let all possible connections between active rules be built, the trade-off while reasoning about distributed conflicts is that we have to verify the paths containing signs of conflicts to make sure all edges (corresponding to the rule connections) in each of these paths are valid. More details are provided in Section 5.4.3.

This algorithm examines the case of adding a single rule to the rule graph. It is common in practice that a set of new rules are deployed in the data plane almost concurrently, meaning that multiple new rules might be added to the rule graph simultaneously. An optimization for this algorithm is conceivable by determining the connections between these new rules in advance, then adding them to the rule graph as new edges without updating their attributes immediately but applying the algorithm only to those new rules which have no connection from other new rules.

This tactic helps reduce the number of rounds expended on updating the new edges' attributes compared to the effort spent on adding each single rule separately to the rule graph.

Removal of an existing rule

When a rule is removed, all paths containing it need to be removed and therefore the influenced edges' attributes holding these paths need to be updated. These influenced edges belong to a subgraph of the original rule graph, being a Directed Acyclic Graph (DAG) having the vertex associated with the removed rules as its single source. Moreover, all edges ending at or starting from the removed rule are also deleted from the rule graph. The removal of an edge does not lead to the removal of its vertices, i. e., the two vertices making that edge are still kept in the rule graph. On the contrary, the removal of a vertex has the consequence of all edges containing it also being removed. Figure 61 illustrates the removal of rule r_{22} from the existing rule graph, all edges containing r_{22} are removed and those belonging to the DAG having the single source as r_{22} must have their attributes updated if their paths containing r_{22}. A question may emerge: why are the two edges (r_{51}, r_{31}) and (r_{31}, r_{41}) not removed as a consequence of the removal of r_{22}, there seems to be no more traffic flowing through these edges? In fact, such traffic can still be generated if there exist end-points connecting to the device with ID 5 and its traffic is matched by r_{51}. The controller can only detect the presence of SDN devices but not end-points, thus not all end-points are present in the rule graph until they send traffic into the network and can then be discovered by the controller.

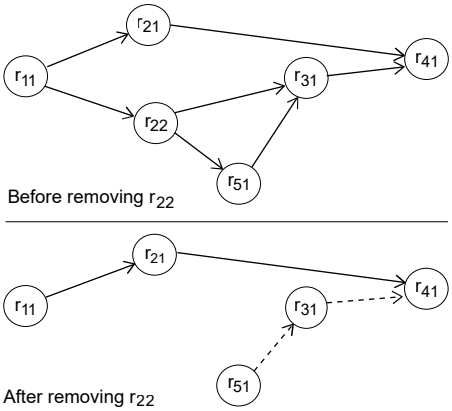


Figure 61: A scenario of removing the rule r_{22} from the existing rule graph, the dashed edges represent those whose attributes might need to be updated

Algorithm 2 shows the process of updating the rule graph when rule RR is removed from table T of device D. All influenced edges have their attributes updated via the recursive invocation of the function *REMOVE_RULE_FROM_RULE_GRAPH*. This function terminates upon one of the following conditions.

- The attribute of an edge has no tuple whose path containing the vertex (D, T, RR). In this case, the variable *COUNT* is still equal to 0 after that edge was examined (line 25).
- There is no more edge in the currently considered subgraph to examine (lines 14–16). This applies also in case vertex V (in line 14) represents a loop, a drop action or an end-point.

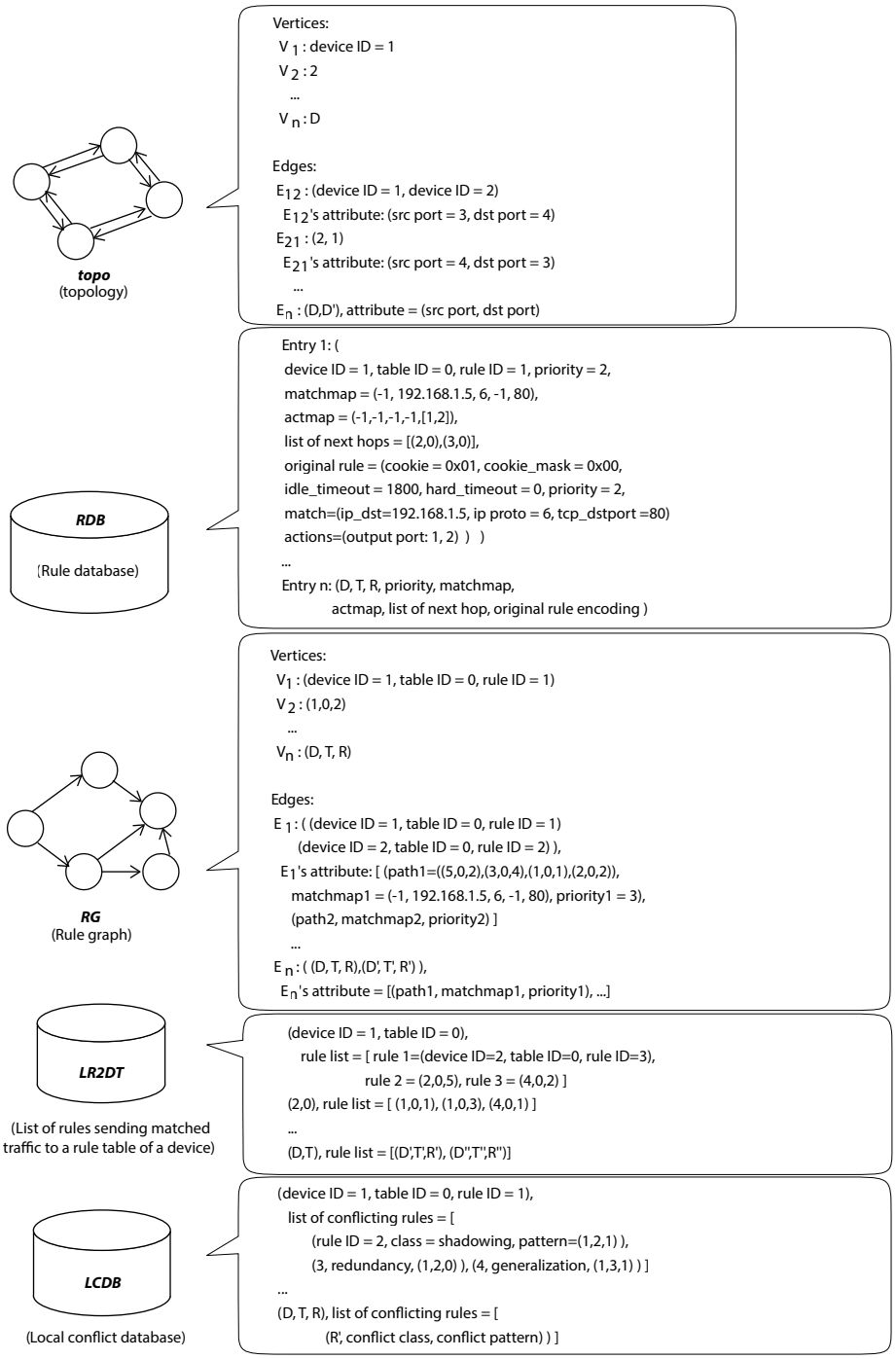


Figure 62: Illustration of the common variables' content used in the algorithms. Each rule is identified by the tuple (device ID, table ID, rule ID) and its information can be retrieved from the rule database.

Algorithm 1 Pseudo-code for updating the rule graph RG when adding a new rule identified by the tuple (D, T, NR)

Input: The tuple (D, T, NR) identifying the new rule in the rule database RDB ▷ *D, T, NR are the ID of the device, the table and the new rule*
Effect (Output): The rule graph RG is updated
Global variables: The existing rule graph RG, the rule database RDB, the network topology $TOPO$, the variable $LR2DT$ containing the mapping between devices' tables and rules sending traffic to them, the local conflict database $LCDB$ (cf. Section 5.5)
Note: More details on the variables are illustrated in Figure 62

1: **if** $LR2DT[D][T]$ is empty **then** ▷ *$LR2DT[D][T]$ is the list of rules sending traffic to Table T of Device D*
2: **for** NH in the list of next hops of rule (D, T, NR) **do** ▷ *Retrieved from the rule database RDB, NH can be i) a subsequent rule table, identified by (device ID, table ID), ii) a device (device ID, 0), i. e., table 0 of that device, iii) an end-point, or iv) a drop action*
3: ADD_RULE_TO_RULE_GRAPH($D, T, NR, NH,$ matchmap of rule $(D, T, NR), ((D, T, NR))$) ▷ *the last variable of this function is a path, in this case contains only a single vertex, being (D, T, NR)*
4: **end for**
5: **else**
6: CR = 0 ▷ *to count the number of rules connected to rule (D, T, NR)*
7: **for** Rule (D_R, T_R, R) in $LR2DT[D][T]$ **do** ▷ *Rule (D_R, T_R, R) sends traffic to Table T of Device D*
8: **if** (D_R, T_R, R) is an active rule **then** ▷ *see Definition 5.20, a rule can be determined as active or inactive based on the local conflict database $LCDB$ defined in Section 5.5*
9: **if** the output port of rule (D_R, T_R, R) does not correspond to the ingress port of rule (D, T, NR) according to $TOPO$ **then** ▷ *i. e., the tuple (output port of rule (D_R, T_R, R), ingress port of rule (D, T, NR)) is not equal to the tuple (src port, dst port) in the attribute of the edge (D_R, D) in $TOPO$*
10: **continue** ▷ *ignore rule (D_R, T_R, R) and check the next rule*
11: **end if**
12: OMM = f_{mat_act}(matchmap of rule (D_R, T_R, R), actmap of rule (D_R, T_R, R)) ▷ *Combine the matchmap and actmap of rule (D_R, T_R, R) to obtain its output matchmap OMM, see Definitions 5.14 and 5.15*
13: Re = (packet-internal match of OMM) $._r$ (packet-internal match of rule (D, T, NR) ▷ *see Section 5.1.2* and Definition 5.6
14: **if** Re == 0 **then** ▷ *0 means a "disjoint" relationship*
15: **continue** ▷ *ignore rule (D_R, T_R, R) and check the next rule*
16: **end if**
17: IMM = (packet-internal match of OMM) ∩ (packet-internal match of rule (D, T, NR)) ▷ *see Theorem 5.2*
18: CR = CR+1 ▷ *increase the number of rules connected to rule (D, T, NR) by 1*
19: Add vertex (D, T, NR) to the rule graph RG
20: Add edge E from vertex (D_R, T_R, R) to vertex (D, T, NR) to the rule graph RG
21: Update the attribute of E with the tuple (path = $((D_R, T_R, R), (D, T, NR))$, matchmap = IMM, priority = NR's priority) ▷ *the attribute of E is a list of tuple, the added tuple in this case becomes the first one in this list*
22: **if** indegree of vertex (D_R, T_R, R) == 0 **then** ▷ *There is no rule connected to rule (D_R, T_R, R) in the rule graph RG*
23: **for** NH in the list of next hops of rule (D, T, NR) **do**
24: ADD_RULE_TO_RULE_GRAPH($D, T, NR, NH, IMM, ((D_R, T_R, R), (D, T, NR))$)
25: **end for**
26: **else**
27: CP = 0 ▷ *to count the number of paths that have connection to rule (D, T, NR)*
28: Extract from the rule graph RG the list of edges $LIER$ ending at vertex (D_R, T_R, R) ▷ *by e. g., looping through all edges of RG and storing the ones satisfying the mentioned condition in $LIER$ ($LIER$: List of IN_EDGES of rule (D_R, T_R, R))*
29: **for** IER in $LIER$ **do** ▷ *IER: IN_EDGE of rule (D_R, T_R, R)*
30: **for** Tuple T in list of tuples of IER **do** ▷ *the attribute of an edge is a list of tuples, each tuple consists of a path, a matchmap associated with that path and a priority*
31: Re_P = (packet-internal match of path P's matchmap) $._r$ (packet-internal match of rule (D, T, NR)) ▷ *relationship of the two multi-property sets, see Section 5.1.2*

32: **if** path P of T contains only 2 vertices and the matchmap in T is the same as that of some other tuple of IER **OR** P represents a loop **OR** Re_P == 0 **then** ▷*A path containing a loop is identified by the last rule following the pattern (device ID, table ID, -4), Re_P == 0 indicates that the two sets are disjoint*
33: **continue** ▷*ignore tuple T and check the next one*
34: **end if**
35: IMM_P = (packet-internal match of path P's matchmap) ∩ (packet-internal match of rule (D, T, NR)) ▷*Intersection of the two multi-property sets, see Theorem 5.2*
36: $CP = CP+1$ ▷*increase the number of paths connecting to rule (D, T, NR)*
37: Create a new path NP by appending the vertex (D, T, NR) to Path P
38: Update the attribute of edge E from vertex (D_R, T_R, R) to vertex (D, T, NR) with the tuple (path = NP, matchmap = IMM_P, priority = NR's priority)
39: **for** NH in the list of next hops of rule (D, T, NR) **do**
40: ADD_RULE_TO_RULE_GRAPH(D, T, NR, NH, IMM_P, NP)
41: **end for**
42: **end for**
43: **end for**
44: **if** CP == 0 **then**
45: **for** NH in the list of next hops of rule (D, T, NR) **do**
46: ADD_RULE_TO_RULE_GRAPH($D, T, NR, NH, IMM, ((D_R, T_R, R), (D, T, NR))$)
47: **end for**
48: **end if**
49: **end if**
50: **end if**
51: **end for**
52: **if** CR == 0 **then** ▷*there is no rule connected to rule (D, T, NR)*
53: **for** NH in the list of next hops of rule (D, T, NR) **do**
54: ADD_RULE_TO_RULE_GRAPH(D, T, NR, NH, matchmap of rule (D, T, NR), $((D, T, NR))$)
55: **end for**
56: **end if**
57: **end if**
58: **function** ADD_RULE_TO_RULE_GRAPH(D, T, R, NH, MM, P)
Input: The device ID D, the table ID T, the rule ID R, the next hop NH of rule (D, T, R), the input matchmap MM and the path P
Effect (Output): The rule graph RG is updated
Global variables: The existing rule graph RG, the rule database RDB, the network topology $TOPO$, the variable $LR2DT$ containing the mapping between devices' tables and rules sending traffic to them, the local conflict database $LCDB$ (defined in Section 5.5)
Note: The input matchmap MM can be the matchmap of rule (D, T, R), or of path P containing this rule as the last rule (also vertex) in this path. The next hop NH can be a rule table with the pattern (Device ID, Rule Table ID), e. g., (D, 1) or (D', 0), can be an end-point or a drop action.
59: **if** NH == "drop" **then**
60: Add vertex $(D, T, -1)$ to the rule graph RG if it does not exist there ▷*Vertex having the value -1 in the position of rule ID (third element in the vertex) means dropping*
61: Add edge E from vertex (D, T, R) to vertex $(D, T, -1)$ to the rule graph RG
62: **if** the tuple list of edge E does not contain the tuple (path=P, matchmap=MM, priority=-1) **then** ▷*priority = -1 means "don't care"*
63: Update the attribute of E by appending to the end of its tuple list the tuple (path=P, matchmap=MM, priority=-1)
64: **end if**
65: **else if** NH is an end-point EP **then**
66: Add vertex $(0, EP's\ IP\ address)$ to the rule graph RG if it does not exist there ▷*We assume that there is no SDN device with ID 0 and use 0 to denote an end-point, so the vertex $(0,\ IP\ address)$*
67: Add edge E from vertex (D, T, R) to vertex $(0, EP's\ IP\ address)$ to the rule graph RG
68: Append vertex $(0,\ EP's\ IP\ address)$ to path P
69: **if** the tuple list of edge E does not contain the tuple (path=P, matchmap=MM, priority=-1) **then** ▷*priority = -1 means "don't care"*
70: Update the attribute of E by appending to the end of its tuple list the tuple (path=P, matchmap=MM, priority=-1)
71: **end if**

72: **if** the tuple list of E does not contain the "minimal" tuple (path=$((D, T, R), (0,\ EP's\ IP\ address))$, matchmap = matchmap of rule (D, T, R), priority=-1) **then**
73: Append to the tuple list of E the "minimal" tuple (path=$((D, T, R), (0,\ EP's\ IP\ address))$, matchmap = matchmap of rule (D, T, R), priority=-1)
74: **end if**
75: **else** ▷*NH is neither a drop action nor an end-point, so it is a rule table identified by a tuple (device ID, table ID)*
76: OMM_R = f_{mat_act}(matchmap of rule (D, T, R), actmap of rule (D, T, R)) ▷*Combine the matchmap and actmap of rule (D, T, R) to obtain its output matchmap OMM_R, see Definitions 5.14 and 5.15*
77: OMM = f_{mat_act}(matchmap MM, actmap of rule (D, T, R)) ▷*Combine the input matchmap MM with the actmap of rule (D, T, R) to obtain the Output Matchmap OMM*
78: $Count$ = 0 ▷*Used to count the number of rules connected by (D, T, R) in the next hop NH*
79: **for** Rule (D_{NR}, T_{NR}, NR) in NH **do** ▷*Next Hop NH is a rule table identified by the tuple (device ID, table ID)*
80: **if** (D_{NR}, T_{NR}, NR) is not an active rule **then** ▷*see Definition 5.20, a rule can be determined as active or inactive based on the local conflict database $LCDB$ defined in Section 5.5*
81: **continue** ▷*Check the next rule in NH*
82: **end if**
83: Re_R = (packet-internal match of OMM_R) $._r$ (packet-internal match of (D_{NR}, T_{NR}, NR)) ▷*Relationship of the two multi-property sets, see Section 5.1.2*
84: **if** Re_R == 0 **then** ▷*The two sets are disjoint*
85: **continue** ▷*Check the next rule in NH*
86: **end if**
87: **if** Output port of (D, T, R) does not correspond to ingress port of (D_{NR}, T_{NR}, NR) according to the topology information in case these two rules reside in two devices, or they do not have the same ingress port in case they are in two tables of the same device **then** ▷*We assume the rule match contains the ingress port field, refer to line 9 of this algorithm for more explanation*
88: **continue** ▷*Check the next rule*
89: **end if**
90: $Count$ = $Count$ + 1 ▷*increase the number of rules in the next hop that have connection from (D, T, R)*
91: IMM_R = (packet-internal match of OMM_R) ∩ (packet-internal match of (D_{NR}, T_{NR}, NR)) ▷*Intersection of the two multi-property sets, see Theorem 5.2*
92: **if** if edge $((D, T, R), (D_{NR}, T_{NR}, NR))$ is not present in RG **then**
93: Add edge E from vertex (D, T, R) to vertex (D_{NR}, T_{NR}, NR) to the rule graph RG
94: **end if**
95: **if** the tuple list of E does not contain the tuple (path = $((D, T, R), (D_{NR}, T_{NR}, NR))$, matchmap = IMM_R, priority = priority of (D_{NR}, T_{NR}, NR)) **then**
96: Update the attribute of E by appending to its tuple list the tuple (path = $((D, T, R), (D_{NR}, T_{NR}, NR))$, matchmap = IMM_R, priority = priority of (D_{NR}, T_{NR}, NR))
97: **end if**
98: Re = (packet-internal match of OMM) $._r$ (packet-internal match of (D_{NR}, T_{NR}, NR))
99: **if** Re == 0 **then**
100: **continue** ▷*Check the next rule in NH*
101: **end if**
102: IMM = (packet-internal match of OMM) ∩ (packet-internal match of (D_{NR}, T_{NR}, NR))
103: **if** Vertex (D_{NR}, T_{NR}, NR) belongs to Path P **then** ▷*Forwarding Loop occurs*
104: Append vertices (D_{NR}, T_{NR}, NR) and $(D_{NR}, T_{NR}, -4)$ to path P ▷*Vertex having the value -4 in the position of rule ID denotes a loop*
105: Update the attribute of edge E from vertex (D, T, R) to vertex (D_{NR}, T_{NR}, NR) by appending to its tuple list the tuple (path = P, matchmap = IMM, priority = priority of (D_{NR}, T_{NR}, NR)) if it is not yet in the list
106: **if** RG does not contain edge the edge $((D_{NR}, T_{NR}, NR), (D_{NR}, T_{NR}, -4)$ **then**
107: Add edge $((D_{NR}, T_{NR}, NR), (D_{NR}, T_{NR}, -4))$, say edge LE, to RG ▷*LE: Loop Edge*
108: **end if**

109: **if** the tuple list of edge LE does not contain the tuple (path = P, matchmap = IMM, priority = -1) **then** ▷*priority = -1 means "don't care"*
110: Update the attribute of LE by appending to its tuple list the tuple (path = P, matchmap = IMM, priority = -1)
111: **end if**
112: **continue** ▷*check the next rule in* NH
113: **end if**
114: Append vertex (D_{NR}, T_{NR}, NR) to path P
115: **if** the tuple list of edge E does not contain the tuple (path = P, matchmap = IMM, priority = priority of (D_{NR}, T_{NR}, NR)) **then** ▷$E = ((D, T, R), (D_{NR}, T_{NR}, NR))$
116: Update the attribute of edge E by appending to its tuple list the tuple (path = P, matchmap = IMM, priority = priority of (D_{NR}, T_{NR}, NR))
117: **end if**
118: **for** NNH in the list of next hops of (D_{NR}, T_{NR}, NR) **do** ▷NNH *is identified by device ID and Table ID, e.g., (D,2) or (D',0) if* NNH *is other than an end-point or a drop action*
119: ADD_RULE_TO_RULE_GRAPH($D_{NR}, T_{NR}, NR, NNH, IMM, P$)
120: **end for**
121: **end for**
122: **if** $Count$ == 0 **then** ▷*No rule in* NH *matches traffic passed by rule* (D, T, R)
123: Add vertex (D, T, R) to the rule graph RG
124: **end if**
125: **end if**
126: **end function**

A vertex representing a loop, a drop action is removed from the rule graph if there is no vertex connected to it (lines 8, 9 and 30, 31). This helps avoid the misleading interpretation in identifying distributed conflicts related to traffic looping or traffic dropping.

Once a tuple whose path containing the removed rule RR is deleted (lines 19–24), the tuple whose path is a subset of the path in the deleted tuple may need to be added to the attribute of the currently considered edge if it is not redundant (lines 25–59). For example, if the path containing rule RR is $[V_1, V_2, V_{RR}, V_3, V_4, V_5]$ (V_{RR} is the vertex corresponding to the removed rule RR), after the tuple containing this path was removed from the edge's attribute, the tuple containing the shorter path $[V_3, V_4, V_5]$ need to be examined and may be added to that edge' attribute.

The handling of loops appears more complex compared to the other cases. If a traffic loop occurs (lines 40–45), the currently considered edge E and the next edge pointing to the vertex representing the loop have their attribute updated. If the loop related to rule RR is removed due to the removal of this rule, the edge pointing to the loop vertex is also removed (lines 28, 29).

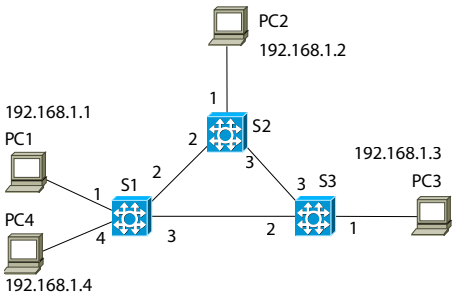


Figure 63: Topology *topo3*. The numbers around a switch indicate the ports from which it connects to other switches or end-points.

In a broader view, besides updating the rule graph due to the removal of an existing rule in the data plane, other actions also need to be carried out: updating the rule database and the conflict database. If a rule shadows or makes another rule

Algorithm 2 Pseudo-code for updating the rule graph RG when removing the rule (D, T, RR)

Input: The tuple (D, T, RR) identifying the removed rule in the rule database RDB ▷D, T, RR *are the ID of the device, the table and the rule*
Effect (Output): The rule graph RG is updated
Global variables: The existing rule graph RG
Note: More details on the variables are illustrated in Figure 62
if Vertex (D, T, RR) does not exist in RG **then**
  Return
end if
for Edge E in the list of edges starting from vertex (D, T, RR) **do**
  V is the vertex at which edge E ends
  REMOVE_RULE_FROM_RULE_GRAPH($(D, T, RR), V$)
  Remove edge E from the rule graph RG
  if V represents a loop, a drop action or an end-point and its indegree is 0 **then**
    Remove vertex V from the rule graph RG
  end if
end for
Remove vertex (D, T, RR) from the rule graph RG
function REMOVE_RULE_FROM_RULE_GRAPH(RV, V)
Input: Vertex RV containing the rule to be removed, vertex V from which edges starting will be examined.
Effect (Output): The rule graph RG is updated
Global variables: The existing rule graph RG
  if Outdegree of V is 0 **then** ▷*there is no edge starting from* V
    return
  end if
  for Edge E in the list of edges starting from V **do**
    $COUNT$ = 0
    for Tuple T in the list of tuples of edge E **do** ▷*The attribute of an edge is a list of tuples*
      if Path P of tuple T contains vertex RV **then** ▷*Each tuple contains a path, a matchmap and a priority*
        $COUNT = COUNT + 1$
        Remove tuple T from the tuple list of edge E
      end if
    end for
    if $COUNT > 0$ **then**
      NV is the vertex in the end of edge E ▷NV*: Next Vertex*
      if NV represents a loop **then** ▷NV *has the form (device ID, table ID, -4)*
        if No tuple in the attribute of edge E contains a path bearing a loop **then** ▷*such path contains the same vertex twice in it*
          Remove edge E from the rule graph RG
          if the indegree of vertex NV is 0 **then**
            Remove vertex NV from the rule graph RG
          end if
        end if
        continue ▷*check the next edge that starts from* V
      end if
      for Edge IE in the list of edges ending at V **do** ▷IE*: in-edge*
        for Tuple IT in the list of tuples of edge IE **do** ▷IT*: input tuple*
          $OMM = f_{mat\_act}$(matchmap in tuple IT, actmap of rule associated with vertex V) ▷*Combine the matchmap of tuple* IT *with the actmap of the rule* V *to obtain the output matchmap* OMM*, see Definitions 5.14 and 5.15*
          IMM = (packet-internal match of OMM) ∩ (packet-internal match of the matchmap of rule NV) ▷*Intersection of the multi-property sets, see Theorem 5.2*

```
40:                     if Vertex NV belongs to path P then                    ▷Forwarding Loop occurs
41:                         Append vertices NV and (device ID in NV, table ID in NV, -4) to the path of tuple IT to
    obtain the new path NP                    ▷Vertex having the value -4 in the position of rule ID denotes a loop
42:                         Update the attribute of edge E by appending to its tuple list the tuple (NP, IMM, NV's
    priority) if it is not yet in the list
43:                         Update the attribute of edge from vertex NV to vertex (device ID in NV, table ID in NV,
    -4) by appending to its tuple list the tuple (NP, IMM, NV's priority) if it is not yet in the list
44:                         continue                                           ▷check the next tuple of edge IE
45:                     end if
46:                     Appending vertex NV to the path of tuple IT to obtain the new path NP
47:                     if NV == "drop" or NV is an end-point then
48:                         Update the attribute of E by appending to the end of its tuple list the tuple (NP, OMM, −1)
    if the list does not contain this tuple                    ▷priority = -1 means "don't care"
49:                         continue                                           ▷check the next tuple of edge IE
50:                     end if
51:                     if NP is a subset of a tuple's path in the tuple list of edge E and IMM is equal to that tuple's
    matchmap then
52:                         continue    ▷The tuple (NP, IMM, priority of NV) is redundant in the attribute of edge E
53:                     else
54:                         Append the tuple (NP, IMM, priority of NV) in the tuple list of edge E
55:                     end if
56:                 end for
57:             end for
58:             REMOVE_RULE_FROM_RULE_GRAPH(RV, NV)
59:         end if
60:     end for
61: end function
```

redundant, its removal would lead to the introduction of this shadowed or redundant rule in the rule graph.

Example

Consider the topology *topo3* in Figure 63, the following rules are to be added to rule table 0 of switches S1, S2 and S3 in the order that they appear, i. e., the first rule of switch S1 is installed first, then the second rule of switch S1, and so on, the last rule of switch S3 (rule 3) is installed last.

```
S1:
1. cookie=1, priority=2, match={ipv4_dst=192.168.1.2, ip_proto=6}, action={output:3}
2. cookie=2, priority=1, match={ipv4_dst=192.168.1.2, ip_proto=6, tcp_dst=80}, action={output:2}
3. cookie=2, priority=1, match={ipv4_dst=192.168.1.3, ip_proto=17}, action={output:3}

S2:
1. cookie=1, priority=1, match={ipv4_src=192.168.1.1, ipv4_dst=192.168.1.2, ip_proto=6, tcp_dst=80}, action
      ↪ ={output:1}
2. cookie=2, priority=1, match={ipv4_src=192.168.1.3, ipv4_dst=192.168.1.2, ip_proto=6, tcp_dst=80}, action
      ↪ ={drop}

S3:
1. cookie=1, priority=1, match={ipv4_dst=192.168.1.2}, action={output:3}
2. cookie=2, priority=2, match={ipv4_src=192.168.1.3, ipv4_dst=192.168.1.2}, action={set_field:ipv4_src
      ↪ =192.168.1.1, output:3}
3. cookie=1, priority=1, match={ipv4_dst=192.168.1.3}, action={output:1}
```

The number in the beginning of each rule is added when it is stored in the rule database for detecting conflicts. This number serves as the identifier of the rule in each rule table. In our prototype (Section 6.1), the newer rule's identifier number is bigger than the older one's. We use *cookie* to identify the rule source, i. e., the control application installing that rule. In this case, there are two control applications, one installs rules with *cookie* 1, the other with *cookie* 2.

Using the below *matchmap template* and *actmap template* (also simplified for the demo purpose), we can represent rules with their *matchmap* and *actmap* once they are stored in the rule database.

***matchmap template**: src IPv4, dst IPv4, IP protocol, src TCP port, dst TCP port, src UDP port, dst UDP port*

***actmap template**: new src IPv4, new dst IPv4, new src TCP port, new dst TCP port, new src UDP port, new dst UDP port, list of output ports*

```
S1:
1. cookie=1, priority=2, matchmap=(-1, 192.168.1.2, 6, -1, -1, -1, -1), actmap=(-1,-1,-1,-1,-1,-1,[3]), nexthop=[3]
2. cookie=2, priority=1, matchmap=(-1, 192.168.1.2, 6, -1, 80, -1, -1), actmap=(-1,-1,-1,-1,-1,-1,[2]), nexthop
      ↪ =[2]
3. cookie=2, priority=1, matchmap=(-1, 192.168.1.3, 17, -1, -1, -1, -1), actmap=(-1,-1,-1,-1,-1,-1,[3]), nexthop=[3]

S2:
1. cookie=1, priority=1, matchmap=(192.168.1.1, 192.168.1.2, 6, -1, 80, -1, -1), actmap=(-1,-1,-1,-1,-1,-1,[1]),
      ↪ nexthop=["end-point:192.168.1.2"]
2. cookie=2, priority=1, matchmap=(192.168.1.3, 192.168.1.2, 6, -1, 80, -1, -1), actmap=(-1,-1,-1,-1,-1,-1,[]),
      ↪ nexthop=["drop"]

S3:
1. cookie=1, priority=1, matchmap=(-1, 192.168.1.2, -1, -1, -1, -1, -1), actmap=(-1,-1,-1,-1,-1,-1,[3]), nexthop=[2]
2. cookie=2, priority=2, matchmap=(192.168.1.3, 192.168.1.2, -1, -1, -1, -1, -1), actmap=(192.168.1.1, -1,-1,-1,-1,-
      ↪ 1,[3]), nexthop=[2]
3. cookie=1, priority=1, matchmap=(-1, 192.168.1.3, -1, -1, -1, -1, -1), actmap=(-1,-1,-1,-1,-1,-1,[1]), nexthop=["
      ↪ end-point:192.168.1.3"]
```

The value -1 in the matchmap indicates any value (i. e., "don't care"), in the actmap it means $no\ change$.

We demonstrate how the rule graph is built by applying the Algorithm 1 in installing each rule one by one, assuming that all rules are in table 0 of the respective switch (as said, there can be multiple rule tables in an SDN device) and the local conflict detection (cf. Section 5.5) has been carried out for the newly installed rule, the device identifiers of switches S1, S2, S3 are 1, 2, 3 respectively. We denote a vertex of the graph as a tuple (device ID, table ID, rule ID), a rule can also be identified by this way in the whole rule database.

The first rule is to be installed in table 0 of switch S1, it is added to the rule database as rule (1,0,1). At this point, there is no rule sending traffic to switch S1, the list $LR2DT[1][0]$ (list of rules sending traffic to table 0 of switch 1) is empty, the code block of lines 1-4 in Algorithm 1 is executed. The next hop of rule (1,0,1) is switch 3, the function ADD_RULE_TO_RULE_GRAPH is invoked with the parameter (1,0,1,3, (-1, 192.168.1.2, 6, -1, -1, -1, -1),((1,0,1),)). The NH (next hop) variable of this function is switch 3, the process proceeds from line 76 in Algorithm 1. As there is not yet any rule in switch S3 at this time, the variable COUNT is equal to 0 (line 112), the vertex (1,0,1) is added to the rule graph RG as the first one.

The second rule (1,0,2) is to be installed in table 0 of switch S1. However, the local conflict detection (cf. Section 5.5) reports that it is shadowed by rule (1,0,1) and thus is inactive (see Definition 5.20) in handling traffic. This rule is not added to the rule graph.

The third rule (1,0,3) is added to the rule graph RG in the same manner as the first rule (1,0,1). RG contains two disconnected vertices (1,0,1) and (1,0,3) at this point.

The list $LR2DT[2][0]$ is empty as there is no rule sending its matched traffic to table 0 of switch S2, the installation of the fourth rule (2,0,1) invokes the function ADD_RULE_TO_RULE_GRAPH with the parameter (2,0,1,"end-point:192.168.1.2", (192.168.1.1, 192.168.1.2, 6, -1, 80, -1, -1),((2,0,1),)) (line 3 of Algorithm 1). The code block of lines 65-74 is performed, which creates a new edge between two vertices (2,0,1) and (0, 192.168.1.1) in the rule graph.

```
((2,0,1),(0,192.168.1.2)), attribute = [(path1 = ((2,0,1),(0,192.168.1.2)), matchmap1 = (192.168.1.1, 192.168.1.2, 6,
    ↪ -1, 80, -1, -1), priority1 = -1)]
```

For the fifth rule (2,0,2), the function ADD_RULE_TO_RULE_GRAPH is called with the parameter (2,0,2,"drop",(192.168.1.3, 192.168.1.2, 6, -1, 80, -1, -1),((2,0,2),)) (line 3 of Algorithm 1). The code block of lines 59-64 is executed, the new edge between two vertices (2,0,2) and (2,0,-1) is added to the rule graph.

```
((2,0,2),(2,0,-1)), attribute = [(path1=((2,0,2),(2,0,-1)), matchmap1 = (192.168.1.3, 192.168.1.2, 6, -1, 80, -1, -1),
    ↪ priority1 = -1)]
```

The sixth rule (3,0,1) is deployed, the list $LR2DT[3][0]$ contains rules (1,0,1) and (1,0,3), the code block between lines 6-56 is active in this case. In the first round of the *for* loop of line 7, say for_{7_1}, rule (D_R, T_R, R) is rule (1,0,1), OMM (line 12) is also the matchmap of rule (1,0,1) as its action does not make changes on the matchmap. The new rule (D, T, NR) is rule (3,0,1), all fields in the chosen matchmap template are packet-internal fields (see Definition 5.4), IMM (line 17) is thus the intersection between the matchmap of rule (1,0,1) and rule (3,0,1) according to the Theorem 5.2, IMM = (-1, 192.168.1.2, 6, -1, -1, -1, -1). The relationship Re (line 14) is also the relationship between the matchmap of rule (1,0,1) and (3,0,1), obtained by applying the $\cdot r$ operator on the two multi-property sets associated with the matchmap of these

two rules (see Theorem 5.1). All fields of rule (D_R, T_R, R) are equal to that of rule (D, T, NR) except the third field of (D_R, T_R, R), being 6, is the subset of the third field of (D, T, NR), being -1 (any). The relationship of their third fields r_3 is encoded as 2 (proper subset), and of other fields $r_1, r_2, r_4, r_5, r_6, r_7$ as 1 (equal). The overall relationship of the matchmap of these two rules is
$Re = r_1 \cdot_r r_2 \cdot_r r_3 \cdot_r r_4 \cdot_r r_5 \cdot_r r_6 \cdot_r r_7 = 1.{}_r1.{}_r2.{}_r1.{}_r1.{}_r1.{}_r1 = 2$
which means the matchmap of (D_R, T_R, R) is a proper subset of that of (D, T, NR) (a more detailed example of applying the $\cdot r$ operator to compute the relationship between two matchmap is described in Section 5.2.1). The value of CR is 1 (line 18), a new edge from vertex (1,0,1) to vertex (3,0,1) is added to the rule graph (lines 19, 20):

```
((1,0,1),(3,0,1)), attribute = [(path1=((1,0,1),(3,0,1)), matchmap1=(-1, 192.168.1.2, 6, -1, -1, -1, -1), priority1 = 1)
    ↪ ]
```

No edge ends at vertex (1,0,1), the function ADD_RULE_TO_RULE-_GRAPH is invoked with the parameter (3,0,1,2,(-1, 192.168.1.2, 6, -1, -1, -1, -1),((1,0,1), (3,0,1))) (line 24). The process goes on from line 76, rule (D, T, R) in this function is rule (3,0,1), OMM_R is equal to the matchmap of rule (3,0,1) since the action of this rule does not alter the matched traffic. OMM (line 77) is (-1, 192.168.1.2, 6, -1, -1, -1, -1). In the first round of the *for* loop at line 79, say for_{79_1}, nested within the first round of the loop for_{7_1}, next hop NH is switch S2, (D_{NR}, T_{NR}, NR) is rule (2,0,1). $Re_R = 3$, indicating that OMM_R is a proper superset of the packet-internal match of rule (D_{NR}, T_{NR}, NR) (line 83). Rule (D_{NR}, T_{NR}, NR) accepts packets from any ingress port (it does not specify the ingress port in its match fields), the condition at line 87 is not satisfied, the process proceeds at line 90, the variable $Count$ has value 1. IMM_R = (192.168.1.1, 192.168.1.2, 6, -1, 80, -1, -1) (line 91). An edge from vertex (3,0,1) to vertex (2,0,1) is added to the rule graph (lines 92, 97).

```
((3,0,1),(2,0,1)), attribute =
    [(path1 = ((3,0,1),(2,0,1)), matchmap1 = (192.168.1.1, 192.168.1.2, 6, -1, 80, -1, -1), priority1 = 1)]
```

OMM (line 98) is (-1, 192.168.1.2, 6, -1, -1, -1, -1), the packet-internal match of (D_{NR}, T_{NR}, NR) (which is rule (2,0,1)) is (192.168.1.1, 192.168.1.2, 6, -1, 80, -1, -1), $Re = 3$ in this line, meaning that OMM is a proper superset of the packet-internal match of rule (D_{NR}, T_{NR}, NR). In line 102, IMM = (192.168.1.1, 192.168.1.2, 6, -1, 80, -1, -1). In line 103, the vertex (2,0,1) does not belong to path P, being ((1,0,1),(3,0,1)), the condition of the *if* statement does not hold. Path P becomes ((1,0,1),(3,0,1),(2,0,1)) (line 114), the attribute of the edge ((3,0,1),(2,0,1)) is updated with path P (lines 115-117):

```
((3,0,1),(2,0,1)), attribute =
  [(path1 = ((3,0,1),(2,0,1)), matchmap1 = (192.168.1.1, 192.168.1.2, 6, -1, 80, -1, -1), priority1 = 1),
```

```
(path2 = ((1,0,1,),(3,0,1),(2,0,1)), matchmap2 = (192.168.1.1, 192.168.1.2, 6, -1, 80, -1, -1), priority2 = 1)]
```

The list of next hop of rule (D_{NR}, T_{NR}, NR) (rule (2,0,1)) is ["end-point:192.168.1.2"] (line 118), the function ADD_RULE_TO_RULE_GRAPH is called recursively in the *for* loop at line 119 with the parameter (2,0,1,"end-point:192.168.1.2", (192.168.1.1, 192.168.1.2, 6, -1, 80, -1, -1), ((1,0,1,),(3,0,1), (2,0,1))). The process continues at line 65, as the edge ((2,0,1),(0,192.168.1.2)) is already present in the rule graph RG, the effect of adding this edge again (line 67) does not cause any change in RG. Path P at line 68 becomes ((1,0,1,),(3,0,1),(2,0,1),(0, 192.168.1.2)). Edge E at line 70 is updated with path P in its attribute.

```
((2,0,1),(0,192.168.1.2)), attribute =
 [(path1 = ((2,0,1),(0,192.168.1.2)), matchmap1 = (192.168.1.1, 192.168.1.2, 6, -1, 80, -1, -1), priority1 = -1),
  (path2 = ((1,0,1,),(3,0,1),(2,0,1),(0,192.168.1.2)), matchmap2 = (192.168.1.1, 192.168.1.2, 6, -1, 80, -1, -1),
       ↪ priority2 = -1)]
```

In the second round of the *for* loop at line 79, say for_{79_2}, nested within the first round of the loop for_{7_1}, next hop NH is switch S2, (D_{NR}, T_{NR}, NR) is rule (2,0,2). Following the steps similar to the first round of the loop for_{79_1}, a new edge between the vertices (3,0,1) and (2,0,2) is added and the attribute of the edge ((2,0,2),(2,0,-1)) is updated.

```
((3,0,1),(2,0,2)), attribute =
  [(path1 = ((3,0,1),(2,0,2)), matchmap1 = (192.168.1.3, 192.168.1.2, 6, -1, 80, -1, -1), priority1 = 1),
   (path2 = ((1,0,1,),(3,0,1),(2,0,2)), matchmap2 = (192.168.1.3, 192.168.1.2, 6, -1, 80, -1, -1), priority2 = 1)]
((2,0,2),(2,0,-1)), attribute =
  [(path1 = ((2,0,2),(2,0,-1)), matchmap1 = (192.168.1.3, 192.168.1.2, 6, -1, 80, -1, -1), priority1 = -1),
   (path2 = ((1,0,1,),(3,0,1),(2,0,2),(2,0,-1)), matchmap2 = (192.168.1.3, 192.168.1.2, 6, -1, 80, -1, -1), priority2
       ↪ = -1)]
```

At this point, the *for* loop at line 79 nested within the first round of the loop for_{7_1} is complete, so is the first round for_{7_1}. In the second round of the *for* loop at line 7, rule (D_R, T_R, R) is rule (1,0,3), rule (D, T, NR) in this round is rule (3,0,1) again, the condition at line 14 holds, this loop terminates.

When the seventh rule (3,0,2) is installed, the process goes on in the similar manner as for the sixth rule (3,0,1), the following edges are added or have their attributes updated:

```
((1,0,1),(3,0,2)), attribute = [(path1=[(1,0,1),(3,0,2)], matchmap1=(192.168.1.3, 192.168.1.2, 6, -1, -1, -1, -1),
      ↪ priority1 = 2)]
((2,0,1),(0,192.168.1.2)), attribute =
 [(path1 = ((2,0,1),(0,192.168.1.2)), matchmap1 = (192.168.1.1, 192.168.1.2, 6, -1, 80, -1, -1), priority1 = -1),
  (path2 = ((1,0,1,),(3,0,1),(2,0,1),(0,192.168.1.2)), matchmap2 = (192.168.1.1, 192.168.1.2, 6, -1, 80, -1, -1),
       ↪ priority2 = -1),
  (path3 = ((1,0,1,),(3,0,2),(2,0,1),(0,192.168.1.2)), matchmap3 = (192.168.1.1, 192.168.1.2, 6, -1, 80, -1, -1),
       ↪ priority3 = -1)]
```

The eighth rule (3,0,3) is deployed, the rule graph RG is updated with the two new edges:

```
((1,0,3),(3,0,3)), attribute = [(path1 = ((1,0,3),(3,0,3)), matchmap1 = (-1, 192.168.1.3, 17, -1, -1, -1, -1), priority1
    ↪ = 1)]
((3,0,3),(0,192.168.1.3)), attribute =
   [(path1 = ((3,0,3),(0, 192.168.1.3)), matchmap1 = (-1, 192.168.1.3, -1, -1, -1, -1, -1), priority1 = -1),
    (path2 = ((1,0,3,),(3,0,3),(0, 192.168.1.3)), matchmap2 = (-1, 192.168.1.3, 17, -1, -1, -1, -1), priority2 = -1)]
```

The complete rule graph is shown in Figure 64. The edges' attribute (not shown in this graph for clarity) can be used to track how a given packet would travel, and some problem can be revealed directly from this graph. For example, from the attribute of the edge ((2,0,2),(2,0,-1)) which embodies traffic dropping

```
[(path1=((2,0,2),(2,0,-1)), matchmap1 = (192.168.1.3, 192.168.1.2, 6, -1, 80, -1, -1), priority1 = -1),
(path2 = ((1,0,1,),(3,0,1),(2,0,2),(2,0,-1)), matchmap2 = (192.168.1.3, 192.168.1.2, 6, -1, 80, -1, -1), priority2 = -
    ↪ 1)]
```

we can infer the impacted traffic: this traffic can be handled by all rules in the rule chain in *path2* or by a subset of rules in this rule chain containing at least two last rules (corresponding to *path1*). In this case, the slice of impacted traffic corresponds to *matchmap2*. We present the algorithms to determine the candidate matchmap of the impacted traffic in Section 5.4.3.

In another example, we show how to exploit the rule graph to determine the "fate" of a packet, which can be employed for other purposes beyond the scope of concerns in this work, e. g., to verify the reachability between end-points, or to check invariants. Consider a TCP packet sent by PC1 (IP address: 192.168.1.1) to PC2 (IP address: 192.168.1.2) with the source and destination TCP ports being 3333 and 80 respectively,

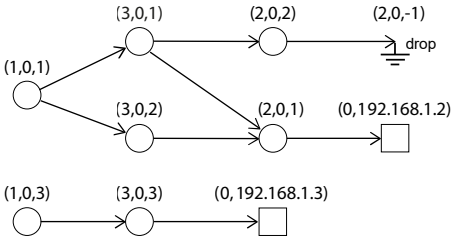


Figure 64: The rule graph after all rules were added

- as PC1 is connected to switch S1, rules in this switch will be checked, this packet is matched by rule (1,0,1);
- based on the rule graph, the next rule handling this packet can be either (3,0,1) or (3,0,2), the matchmap and the priority of the edges from rule (1,0,1) to these two rules will decide: this packet belongs to matchmap1 of the edge ((1,0,1),(3,0,1)) and thus is processed by rule (3,0,1);
- the rule graph suggests the next rule can be (2,0,1) or (2,0,2), the matchmap of these edges reveals the correct one to be (2,0,1);

- the matchmap of the edge ((2,0,1),(0,192.168.1.2)) shows a match with the packet, thus it will be delivered successfully to the end-point 192.168.1.2

In practice, for such a communication between PC1 and PC2, there must also be rules for the backward direction from PC2 to PC1, we show only exemplary rules for the ease of illustration.

The removal of a rule is intuitive. Consider the scenario in which rule (3,0,1) is removed from the rule graph in Figure 64. All edges containing this rule are deleted, the new rule graph is shown in Figure 65. The edges belonging to the DAG having the deleted rule (3,0,1) as source are also influenced. Their attributes need to be updated by removing the tuple whose path containing this rule. These edges include ((2,0,1),(0,192.168.1.2)) and ((2,0,2),(2,0,-1)), their attributes become:

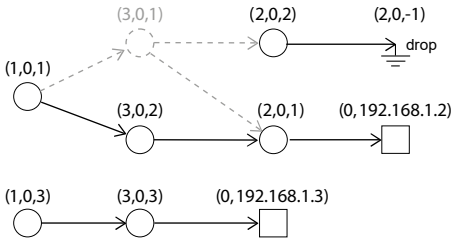


Figure 65: The rule graph after rule (3,0,1) was removed

((2,0,1),(0,192.168.1.2)), attribute =
[(path1 = ((2,0,1),(0,192.168.1.2)), matchmap1 = (192.168.1.1, 192.168.1.2, 6, -1, 80, -1, -1), priority1 = -1),
(path3 = ((1,0,1,),(3,0,2),(2,0,1),(0,192.168.1.2)), matchmap3 = (192.168.1.1, 192.168.1.2, 6, -1, 80, -1, -1),
↪ priority3 = -1)]
((2,0,2),(2,0,-1)), attribute = [(path1=((2,0,2),(2,0,-1)), matchmap1 = (192.168.1.3, 192.168.1.2, 6, -1, 80, -1, -1),
↪ priority1 = -1)]

5.4.3 Verifying the validity of a path in the rule graph

The rule graph can be exploited for various purposes, e. g., to verify the reachability between a pair of end-points in exchanging certain slices of traffic, or to figure out problems related to traffic looping or dropping. These can be accomplished by searching in the rule graph for the leaves (i. e., vertices whose outdegree equal to 0) satisfying a particular pattern (e. g., traffic loops have the pattern $(X, Y, -4)$), then extracting the attributes of the edges ending at these vertices and controlling the validity of the paths therein. If there exists traffic flowing along a path, the problem or the verification can be confirmed to be true. We refer to this traffic as the path's candidate traffic in the following and demonstrate the steps to determine it as well as to confirm if the path is valid.

The entrance point of the candidate traffic is the first rule in the path, it must thus match this rule, so that it can be absorbed by the path. Then we need to check that the traffic is not deviated to another path due to local conflicts of rules in the current path with other rules. The verification is necessary as we allow exceptions to reduce complexity during the rule graph building process.

Algorithm 3 shows the steps to determine the candidate matchmap (CMM) associated with the candidate traffic of the path under consideration. CMM is initialized as the first rule's matchmap of the path. Each field of CMM can be marked as fixed or changeable. Once a field is marked as fixed, it will not be changed anymore. If all fields are marked as fixed, the process of determining CMM is finished. Otherwise, the process goes on until the last vertex of the path is examined. The non-packet-internal fields of CMM stay unchanged until the end of the combination process and are marked as fixed fields. The packet-internal fields of CMM is determined based on the action and the match fields of each rule in the path. If the rule action is to change a field, that field is marked as fixed in CMM. The value of a changeable field in CMM is set to that of the corresponding field of the rule in concern if that rule's field value is a subset of the current value of the CMM's field.

We retain the fields of CMM if the rules' action is to change them. The reason is that the rule action can change the rule match completely, an extreme example is that the rule action modifies all packet-internal fields of the rule match and makes the output match space absolutely disjoint with the original rule match at all packet-internal fields. Since we attempt to determine the candidate traffic starting from the first rule in the path, it must match this rule. The incorporation of the changes made by the rule action into CMM can lead to the consequence that the first rule does not match the candidate traffic anymore.

Algorithm 3 Pseudo-code for determining the candidate matchmap of a path

Input: Path P, which is a sequence of vertices, each corresponding to a rule
Output: Candidate matchmap CMM
CMM = First rule's matchmap ▷*CMM: candidate matchmap*
Mark all non-packet-internal fields of CMM as fixed and all packet-internal fields of CMM as changeable
if P has only one vertex **then**
    Return CMM
else
    Mark those packet-internal fields of CMM as fixed where the action of the first rule is to change them
    if All CMM's fields are marked as fixed **then**
        Return CMM
    end if
end if
for Rule (D, T, R) in path P **do**
    if (D, T, R) is the first rule **then** ▷*Already examined, so ignore it*
        continue
    end if
    Combine each of the changeable fields of CMM with the corresponding fields of (D, T, R) according to the rule: if that field value of rule (D, T, R) is a proper subset of that of CMM, set that field value of CMM to that of (D, T, R) ▷*The information of rule (D, T, R) is retrieved from the rule database RDB, see Figure 62*
    if (D, T, R) is the last vertex of P **then**
        Return CMM
    end if
    Mark those changeable packet-internal fields of CMM as fixed if the action of (D, T, R) is to change them
    if All CMM's fields are marked as fixed **then**
        Return CMM
    end if
end for

After having the candidate traffic represented by its matchmap, we have to verify it twice.

- At its entrance point, which corresponds to the first vertex of the path, the candidate traffic or part of it can be handled by other rules due to possible local conflicts of the rule in the first vertex with other rules in the same table. We need to determine the part of the candidate traffic that is handled exclusively by the rule in the first vertex. The candidate traffic can therefore shrink down in this step.
- We need to check if the candidate traffic would flow according to the given path or it might be deviated to another path due to exceptional cases that were ignored during the rule graph building process.

Algorithm 4 Pseudo-code for determining the candidate matchmap of a path at the entrance point

Input: First vertex V of path P, candidate matchmap CMM
Output: Candidate matchmap CMM of path P after removing its part handled by other rules due to local conflicts.
Global variables:: Local Conflict Database $LCDB$ (see Section 5.5 and Figure 62)

```
1: (D, T, R_e) is the rule in vertex V                ▷(D, T, R_e) at entrance point is an active rule (Definition 5.20)
2: for (D, T, R) in the list of rules with which (D, T, R_e) has local conflicts based on the local conflict database LCDB do
3:     if (D, T, R) is not inactive (Definition 5.20) and the priority of (D, T, R) is greater than the priority of (D, T, R_e) then
4:         Remove from CMM the overlap between CMM and the matchmap of rule (D, T, R)
5:         if CMM is empty then                       ▷this happens when any fields of CMM is empty
6:             Return
7:         end if
8:     end if
9: end for
```

If the match space associated with CMM becomes empty after removing its overlapping part handled by other rules according to Algorithm 4, we can conclude that path P is not valid as there exists no traffic handled by that path. Otherwise, the next step is carried out according to Algorithm 5. Every rule in the path is considered until the last one. At each step, CMM is combined with the rule action and compared with the relevant matchmap of all edges starting from the vertex containing that rule. If during this process, CMM becomes empty due to the removal of its overlapping part with the relevant matchmap, then the path is not valid. If CMM is not empty until the last vertex of the path, we can confirm that path's validity.

5.5 Local conflict detection

Local conflicts presented in Chapter 4.1 can be detected based on their patterns which characterise the relationship of the priority, the match and action components of a pair of rules in concern. The comparison of the priority, being the number, is straightforward; the match relationship is derived from their *matchmap* following the method developed in Section 5.2.1; using the $actmap$ notion introduced in Section 5.2.2, the

Algorithm 5 Pseudo-code for verifying the validity of a path

Input: Path P, candidate matchmap CMM
Output: Conclude if path P is valid
for Vertex V in path P **do**
  if V is the last vertex of P **then**
    Conclude that path P is valid
    Return
  end if
  (D, T, R) is the rule in V
  $CMM = f_{mat\_act}(CMM$, actmap of rule $(D, T, R))$ ▷*output matchmap CMM calculated from CMM and the actmap of rule (D, T, R), see Definitions 5.14 and 5.15, the information of rule (D, T, R) is retrieved from the rule database RDB, see Figure 62*
  E_p is the edge starting from V in path P
  Tuple T_p in the list of tuples of E_p that has its path being a part of path P ▷*the attribute of an edge is a list of tuples of (path, matchmap, priority)*
  Pri_p is the priority in tuple T_p
  for Edge E in the list of edges starting from V other than E_p **do**
    Tuple T in the list of tuples of E that has its path being a part of path P except the last vertex in that path
    MM is the matchmap in tuple T
    Pri is the priority in tuple T
    if MM overlaps with CMM and Pri is greater than Pri_p **then**
      Remove from CMM the overlapping part between CMM and MM
      if CMM is empty **then**
        Conclude that path P is not valid
        Return
      end if
    end if
  end for
end for

action can also be compared intuitively. We can then conclude if a conflict exists between two given rules and of which class it is. By maintaining the information of to which control application a rule belongs, e. g., using *cookie* in OpenFlow SDN, we are able to tell whether local conflicts occur between control applications.
We find that the encoding of conflict patterns by tuples of numbers is convenient for the comparison required in detecting local conflicts. Specifically, for two given rules i and j, we encode the relationship of their priority as:

- $pri_i = pri_j$: 0
- $pri_i < pri_j$: 1
- $pri_i > pri_j$: 2

the relationship between their match fields is encoded in the same manner as for general sets mentioned in Section 5.1.2:

- disjoint: 0 ($match_i \cap match_j = \emptyset$)
- equal: 1 ($match_i \cap match_j = match_i = match_j$)
- proper subset: 2 ($match_i \cap match_j = match_i \wedge match_i \neq match_j$ or $match_i \subset match_j$)
- proper superset: 3 ($match_i \cap match_j = match_j \wedge match_i \neq match_j$ or $match_i \supset match_j$)

- intersecting: 4 ($match_i \nsubseteq match_j \wedge match_i \nsupseteq match_j \wedge match_i \cap match_j \neq \emptyset$)

the relationship of there actions is encoded as:

- same: 0
- different: 1

The comparison of rule relationships and conflict patterns to identify local conflicts is then as simple as comparing tuples of numbers. For example, the *shadowing* local conflict pattern can be represented by the tuple (1, 1, 1) or (1, 2, 1), if the relationships between the priority, match fields and actions of two rules i and j form one of these two tuples, a *shadowing* local conflict between them can be deduced.

Local conflicts in a given rule set can be detected exhaustively by iterating through each pair of rules. In another scenario where a new rule is to be added in the rule table of a switch, we only need to compare it with the existing rules in that rule table to detect local conflicts. The latter case is demonstrated in Algorithm 6 and is implemented in our conflict detection prototype in Section 6.1.

Similar to the rule database, a database of local conflicts needs to be maintained when a rule is added or removed. Each entry of this database has the format *(device ID, table ID, rule ID, a list of tuples)*, each tuple in the list of tuples contains: a rule (represented by an ID number in the rule database) exposing a local conflict with the rule identified by the first three elements (device ID, table ID, rule ID), the corresponding conflict class and conflict pattern.

The introduction of a new rule NR having an overlapping (i. e., non-disjoint) match space with an existing rule ER can have some effect and the local conflict database $LCDB$ may need to be updated accordingly:

- No or negligible effect if NR and ER have the same priority, match, action (lines 11, 12) of Algorithm 6.
- NR overwrites ER if they have the same priority, match but different action (lines 13, 14). This might vary for other SDN implementation but it holds for our chosen test-bed platform (using Open vSwitch-based OpenFlow switches). In this case, ER is removed from the rule set and the rule database; the conflict database (including local, distributed and hidden conflict) needs to be updated respectively.
- NR dominates for the traffic belonging to the overlapping space between the two rules if NR has higher priority than ER (lines 15, 16). In this case, the entry for ER in the local conflict database $LCDB$ is updated and not that of NR.
- ER dominates for the traffic belonging to the overlapping space between the two rules if NR's priority is equal to or lower than ER (lines 17, 18). In this case, the entry for NR in the local conflict database $LCDB$ is updated.

The way we encode the local conflict database in the last two cases eliminates the redundancy of encoding the local conflict for both involved rules (NR and ER) while still ensuring the capability of reasoning about that local conflict when necessary.

Algorithm 6 Pseudo-code for detecting local conflicts when adding the new rule (D, T, NR)

Input: The tuple (D, T, NR) identifying the new rule in the rule database RDB ▷*D, T, NR are the ID of the device, the table and the new rule*
Effect (Output): The local conflict database $LCDB$ is updated if new local conflicts are found
Global variables: The List of Local Conflict Patterns LCP, the rule database RDB, the local conflict database $LCDB$ ▷*As mentioned above, a local conflict pattern can be represented as a triple, e.g., (1,2,1)*
Note: More details on the variables are illustrated in Figure 62
for Rule (D, T, R) in Rule Table (D, T) **do** ▷*(D,T) identifies table T of device D*
  if (D, T, R) and (D, T, NR) are from the same control application **then** ▷*e.g., using cookie in OpenFlow SDN to relate a rule to its control application*
    continue ▷*Do not examine conflicts between rules from the same control application, we consider this to be a bug or an intended behaviour of that application, e.g., to refresh the timeout of its rule by installing the same rule again*
  end if
  Re_{mm} = (matchmap of (D, T, NR) $._r$ (matchmap of (D, T, R)) ▷*relationship of two multi-property sets, see Section 5.1.2, the result can be either 0 (disjoint), 1 (equal), 2 (proper subset), 3 (proper superset), or 4 (intersecting)*
  if Re_{mm} != 0 **then** ▷*not disjoint*
    Re_{pri} = relationship between NR's priority and R's priority ▷*smaller, equal or greater*
    Re_{act} = relationship between NR's action and R's action ▷*same or different*
    for Pattern P in LCP **do**
      if P matches the tuple $(Re_{pri}, Re_{mm}, Re_{act})$ **then**
        if (D, T, NR) and (D, T, R) have the same priority, match and actions **then**
          break ▷*We choose to do nothing in this case, one can however reset the timeout of rule (D, T, R) in the rule database RDB, or remove rule (D, T, R) from and add rule (D, T, NR) to RDB*
        else if (D, T, NR) and (D, T, R) have the same priority, match but different actions **then** ▷*This case depends on the SDN implementation, the behaviour in the next line holds for our test-bed*
          Remove rule (D, T, R) from the rule database RDB ▷*As said, the removal of a rule also triggers the change in the conflict databases*
        else if (D, T, NR) has higher priority than (D, T, R) **then**
          In $LCDB$, create (if not yet available) or append to the tuple list of the entry ((D, T, R),tuple list) the tuple (NR, conflict class of pattern P, pattern P)
        else
          In $LCDB$, create (if not yet available) or append to the tuple list of the entry ((D, T, NR),tuple list) the tuple (R, conflict class of pattern P, pattern P)
        end if
        break ▷*Proceed with the next rule in Table (D, T)*
      end if
    end for
  end if
end for

When a rule is removed from the rule database, the local conflict database $LCDB$ needs to be updated. Algorithm 7 illustrates this case.

5.6 Distributed conflict detection

We organize distributed conflicts in seven classes according to their causes *(traffic looping, traffic dropping, packet modification, changes to paths)* and directions *(upstream, downstream)* (see Chapter 4.2). The rule graph (Section 5.4) provides an efficient means for detecting conflicts belonging to the two distributed conflict classes:

Algorithm 7 Pseudo-code for updating the local conflict database $LCDB$ when removing rule (D, T, RR)

Input: The tuple (D, T, RR) identifying the removed rule in the rule database RDB ▷D, T, RR *are the ID of the device, the table and the new rule*
Effect (Output): The local conflict database $LCDB$ is updated (if rule (D, T, RR) was present there).
Global variables: the rule database RDB, the local conflict database $LCDB$

```
1: if LCDB contains an entry having the pattern ((D, T, RR),tuple list) then
2:     remove from LCDB this entry
3: end if
4: for Entry E in LCDB containing Device D, Table T do ▷E follows the pattern ((D, T, a rule number), a tuple list)
5:     for Tuple Tup in the tuple list of E do                ▷Tup contains (rule number, conflict class, pattern)
6:         if The first element of Tup is equal to RR then
7:             remove Tup from the tuple list of E
8:         end if
9:     end for
10: end for
```

policy suppression by downstream traffic looping and *policy suppression by downstream traffic dropping*. Conflicts belonging to other distributed conflict classes are largely interpretative due to the lack of an effective method to verify their existence. As far as we are concerned, no existing solution in literature can be employed for their detection either. Therefore we provide instead in-depth discussions on limitations and practical implications for these conflict classes.

5.6.1 Detecting conflicts belonging to *downstream traffic looping/dropping* distributed conflict classes

The encoding used in the rule graph facilitates the reasoning about conflicts belonging to these two distributed conflict classes. After the rule graph is built or updated, we search for vertices having the patterns $(X, Y, -1)$ and $(X, Y, -4)$, which denote the drop action and the traffic loop respectively. The attribute of each edge connected to these vertices contains a list of tuples of (path, matchmap, priority) that reveals the involved rules and the affected traffic. Each path shows a sequence of rules handling the influenced traffic, which is eventually dropped or caught in a loop. Given a path, we can determine the candidate matchmap of the influenced traffic according to Algorithms 3 and 4. If the candidate traffic exists (i. e., the resulting candidate matchmap of these two algorithms is not empty), and the path is verified to be valid based on Algorithm 5, we can conclude the problem of traffic looping or dropping. It becomes a conflict if the rules in the path are installed by different control applications. In our implementation, this can be checked by referencing to the rule *cookie* in the rule database. In any case, the problems related to (unintended) traffic dropping and traffic looping need to be detected and handled timely, even if it is a bug of a control application or an erroneous deployment of rules by an administrator. The intended traffic dropping, e. g., by a firewall, should be noticed in advance to disable unnecessary detection results.

The conflicts detected by this approach are classified to be in the downstream direction as the rules of the "victim" control application(s) were active on the influenced traffic before it is dropped or looped. The incident of traffic dropping or looping is thus considered to arise in the downstream direction with reference to the target device(s) of the application(s).

5.6.2 Coping with other distributed conflict classes

For other distributed conflict classes, we realize that without an effective means to verify the expected network behaviour of each control application, the conclusions based on the signs of conflicts in their co-deployment are rather interpretative. The signs for distributed conflicts can be observed based on how the target traffic of each control application being handled in the network, obtainable via monitoring tools (e. g., with the help of a packet analyzer like *tcpdump*[2]) or by using the rule graph. Still, the following questions could not be addressed reasonably:

- **Q1:** How would the target traffic of a control application be handled if it was executed in isolation? The subsequent questions emerge as a consequence of this question.
- **Q2:** Consider the target traffic of a control application being dropped or looped before reaching its target switch(es), would that traffic be able to reach this switch and processed by the rules of that control application if it were not dropped or looped? The problems related to the traffic looping or dropping must be identified and resolved anyway, but the conclusion that the conflict belongs to the class *policy suppression by upstream traffic looping/dropping* appears unpersuasive.
- **Q3:** A packet was modified once or multiple times by rules from different control applications on its way from the source to the destination, but it gets delivered successfully, and none of the control applications intends traffic dropping or looping, would this case be considered a conflict?
- **Q4:** A packet interested by a control application was modified once or multiple times by rules of the others, therefore this packet is not matched by its rule(s) at its target switch(es) any more, yet gets delivered successfully, none of the control applications intends traffic dropping or looping, does a conflict exist in this situation?
- **Q5:** A packet interested by a control application is forwarded by rules of another and could not reach its target device, yet that packet is delivered successfully, none of the control applications intends traffic dropping or looping, would there be a conflict?

2 https://www.tcpdump.org/

The observation leading to the conflict classification relies on our experimental approach, in which the network behaviour observed in the isolated execution of each control application is compared with that in the co-deployment of control applications. However, the isolated deployment of a control application as a means for detecting conflicts is generally inapplicable in practice: i) it is impractical to stop a network running various applications/services to deploy each of them in isolation and record the network behaviour, ii) the replication of a running network for such testing is too expensive, not to mention the fact that end-points and their generated traffic are ordinarily unknown. Therefore, we could not count on this approach for detecting conflicts in a running network. Due to the interpretative nature of these conflict classes and the lack of an efficient method to detect them, we discuss in the following measures to issue warnings on potential presence of their instances. A warning indicates the uncertainty in concluding the presence of conflicts, but it is worthwhile for a closer look by a network operator to decide if conflicts exist.

Coping with the classes: *upstream traffic looping/dropping*
Conflicts of these two classes occur from the perspective of a control application when it places rules in some device(s) with the intent to regulate certain slice of traffic, yet this traffic gets dropped or looped before reaching the target device of that application (thus, the incident transpires in the upstream direction of the device), leaving its rules never active on that traffic. In spite of the interpretative nature of these conflicts due to questions Q1 and Q2 above, the effect of traffic dropping or looping is obvious. If no control application intends traffic dropping or looping, the problem must be handled. The rule graph can be employed to detect these conflicts in the same manner as for the distributed conflict classes *downstream traffic looping/dropping* (see Section 5.6.1).

Coping with the class: *downstream packet modification*
A conflict of this class is observed when rules of a control application is active on its target traffic at its target device(s), that traffic is modified later on by rules from other applications in such a way that invalidates the first application's intent. A case exposing similar symptoms in the concurrent execution of control applications is interpretative due to questions Q1 and Q3. An approach to identify symptoms and issue warnings on potential conflicts of this class for a control application can be envisioned via the following steps.

1. Determining rules of that control application in its target devices.
2. From each rule R of these rules (corresponding to a vertex in the rule graph), tracing forward along the edges in the rule graph until the last edge. A list of last edges are obtained after this step.
3. Examining the attribute of each edge in the above list of edges, the attribute is a list of tuples of (path, matchmap, priority), path P in the list of all paths containing rule R is inspected. A warning is issued if all of the following conditions hold:

a) Path P does not represent traffic looping or dropping.
b) At least one rule standing between rule R and the last rule in path P performs packet modification.
c) Path P is valid (see Section 5.4.3)

In practice, it might be unusual that the address fields of a packet, e. g., IPv4, MAC addresses or TCP/UDP ports, are modified frequently (except for known scenarios such as SDN devices are functioning as routers). The rule graph can be extended to issue warnings in this situation during its building process. In each tuple in the attribute of an edge, an extra counter can be added to count the number of rules in the path that modify packets. When a connection between a rule pair is established and the ending rule in that connection performs packet modification, the counter is increased. One may decide to raise warnings based on the counter value. Another solution is presented by Reyes [90], which employs the rule graph to determine if any path leading to the newly deployed rule contains a rule modifying traffic besides the current rule (i. e., the newly deployed one). If this holds and these two rules come from two distinct control applications, there exists some traffic modified at least twice by two applications, a conflict of the *general multi-transform* class is raised. As discussed in Chapter 4.2.5, this distributed conflict class – *downstream packet modification* – is synthesized from the *general multi-transform* distributed conflict class in Reyes' work. His proposed algorithm, reproduced in Algorithm 8, to cope with this kind of conflicts is thus applicable in this case.

Coping with the class: *upstream packet modification*
An instance of this class arises when a packet interested by a control application was modified by the others before entering its target device(s), and thus is not matched by its rules anymore. Eventually the overall policies of that application are not obtained. The conclusion for a distributed conflict of this class based on the similar symptom shown in the co-deployment of applications is interpretative due to questions Q1 and Q4. A warning of this kind of conflicts can be released if a valid path in the rule graph reflecting this symptom is found. Consider the case with control application A interested in traffic T and having its rules deployed in device D, D has only one rule table, the candidate path will be composed of a list of vertices divided in three parts.

- Part 1: a list of one or multiple vertices before device D, traffic T is matched and modified by some of these vertices and becomes T'.
- Part 2: a vertex in device D, the rule associated with this vertex is not installed by application A, the modified traffic T' is not interested by application A.
- Part 3: a list of zero (empty list) or multiple vertices after device D.

We can trace for the candidate path in the rule graph with the starting point being each rule R in the target device D of application A that is not installed by A. If a

Algorithm 8 Pseudo-code for detecting *general multi-transform* conflicts (adapted from an existing algorithm in Reyes' work [90] (Algorithm 2, Chapter 6.4, page 82 of his thesis))

Input: The tuple (D, T, NR) identifying the new rule to be added in the rule graph RG ▷*D, T, NR are ID number of the device, the table and the new rule*
Output: Alarms of rules that together with the new rule leading to *multi-transform* conflicts
Global variables: the rule database RDB, the rule graph RG
Note: More details on the variables are illustrated in Figure 62
1: **if** the actions of (D, T, NR) modify matched packets **then**
2: CHECK_MULTI_TRANSFORM((D, T, NR), (D, T, NR), $False$)
3: **end if**
4: **function** CHECK_MULTI_TRANSFORM((D, T, NR), (D_R, T_R, R), $visited$)
Input: the new rule (D, T, NR) to be added in the rule graph RG, rule (D_R, T_R, R) standing before rule (D, T, NR) on the path containing (D, T, NR), boolean variable $visited$ marking the visited state of rule (D, T, NR).
Output: Alarms of rules that together with (D, T, NR) causing *multi-transform* conflicts.
Global variables: The existing rule graph RG, the rule database RDB
5: **if** (D, T, NR) and (D_R, T_R, R) are the same rule **AND** visited == True **then** ▷*There is a loop*
6: Return ▷*The traffic loop is handled separately as mentioned in Section 5.6.1, this function stops here*
7: **end if**
8: Extract from the rule graph RG the list of vertices LV, from which edges connected to the vertex associated with rule (D_R, T_R, R) start ▷*each directed edge in the rule graph starts from a rule and ends at another rule*
9: **for** Vertex V in the list of vertices LV **do**
10: Extract rule (D_{RV}, T_{RV}, RV) corresponding to Vertex V
11: **if** the actions of (D_{RV}, T_{RV}, RV) modify matched packets **AND** the output match space of (D_{RV}, T_{RV}, RV) overlaps with the match space of rule (D, T, NR) **AND** (D_{RV}, T_{RV}, RV) and (D, T, NR) belong to different control applications **then** ▷*the output match space of a rule is the combination of its match fields and its actions (see Definition 5.16), the association of a rule to its control application can be obtained via certain metadata fields of that rule, e.g., rule cookie in OpenFlow SDN*
12: Raise a *multi-transform* conflict with rule (D_{RV}, T_{RV}, RV) and rule (D, T, NR)
13: **else**
14: CHECK_MULTI_TRANSFORM((D, T, NR), (D_{RV}, T_{RV}, RV), $True$) ▷*Recurse with the new rule (D, T, NR) and rule (D_{RV}, T_{RV}, RV). As (D, T, NR) has been already visited by this function in its very first invocation, the associated boolean variable is set to True*
15: **end if**
16: **end for**
17: **end function**

valid path in the attribute of each edge starting from rule R (i. e., the vertex associated with rule R) contains rules satisfying part 1 above, a warning will be given.

The practical point mentioned in the previous section for the distributed conflict class *downstream packet modification* also applies for this conflict class: the rule graph can be adapted to report unusual cases in which a packet is modified multiple times on its way. This is also an efficient measure against the firewall bypassing attack mentioned in Chapter 4.2.6 by means of packet modification (a packet is modified to bypass the firewall, then re-modified after crossing the firewall to reach its destination).

Coping with the class: *changes to paths*

The interested traffic of a control application is forwarded by rules of the others on a path not covering its target devices, leading to this conflict. A case with the similar symptom is interpretative due to questions Q1 and Q5. In order to issue a warning for

this kind of conflicts, we need to check all valid paths in the rule graph that handles the interested traffic of that application but does not contain any vertex (also rule) residing in its target devices. Obviously, the expenditure for this exploration would be intense as we might need to check the attributes of all edges ending at a leaf, i. e., a vertex whose outdegree is zero.

This kind of conflicts can be avoided if the control application chooses the target devices for rule deployment in such a way that all paths along which its interested traffic flows will cross these devices. For instance, a firewall application aiming to filter all traffic from outside should install its rules on all devices on the network boundary. A similar example was shown in Chapter 4.2.7.

5.7 Hidden conflict detection

In coping with hidden conflicts, our first attempt was to consider control applications as black-boxes and employed speculative provocation in probing their behaviour to predict conflict occurrence. This approach has the advantage of not demanding prior knowledge of control applications. The prediction is based on fake events that possibly trigger control applications' reactions, which are intercepted and examined to derive hidden conflicts. However, we observe that, on the one hand, stateful control applications are influenced by fake events and may function incorrectly in reacting to subsequent genuine events. On the other hand, the number of fake events for probing is extremely large, making it too expensive for hidden conflict prediction, another issue is the interference between the fake and genuine events. Therefore, we opt for a "grey-box" approach that takes some input of the involved control applications in order to detect hidden conflicts.

We present a method to detect hidden conflicts of the class: *event suppression by local handling*. The four classes *event suppression by upstream traffic looping/dropping, event suppression by changes to paths* and *action suppression by packet modification* reveal the interpretative quality (see Section 5.6.2) that restricts the reliable identification of their instances in the co-deployment of control applications, we discuss alternatively approaches to cope with them from a practical viewpoint. The other two classes, including *undue trigger* and *tampering with event subscription*, appear to arise in the course of a security attack in the control plane, we point out existing measures in literature applicable for their handling.

5.7.1 Considering the hidden conflict prediction approach

We have developed in our earlier work [25] a hidden conflict predictor applicable for the first hidden conflict class *event suppression by local handling*. We present in the following our experiences with this approach and justify against its employment.

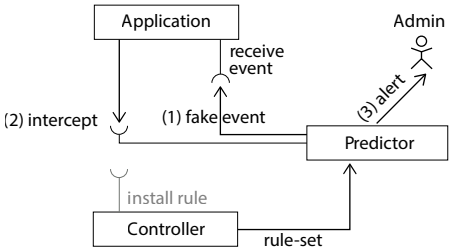


Figure 66: Hidden conflict predictor mechanism [25]

In this approach, each control application is a black-box to the predictor. This view is reasonable as control applications can be implemented by third-party bodies and their precise behaviour can be unknown to a network operator. In essence, the prediction of hidden conflicts is carried out in three steps illustrated in Figure 66.

1. The Hidden Conflict Predictor generates fake events simulating possible incidents in the data plane, e. g., a packet arrives at a switch and is escalated to the controller, or some port of a switch is down.
2. Control applications interested in such events may react by installing, removing or modifying rules in the data plane.
3. The predictor intercepts these reactions, analyses if there exist conflicts and informs the administrator of the issues.

For each of these steps, we notice the obstacles hindering the choice of this approach.

Generating fake events

A controller can create for control applications different kinds of events related to topology changes, monitoring data and requests for instructions on handling certain traffic.

- In general, any change of the topology requires the intervention of the administrator. Fake events related to topology changes can be useful during the network management to probe possible effects to control applications. In a normal run, such events are rare while their occurrence should be handled separately in terms of the network management in conjunction with human efforts, the probe for predicting hidden conflicts based on these events is therefore not encouraged.
- A fake event containing monitoring information can be useful in estimating the network resilience in case of excessive load. We notice, however, that control applications interested in such events tend to be active ones (see Chapter 3.5.1), i. e., they actively monitor the network and make changes if necessary, or there can be a controller service that periodically monitor the network and notify them for critical incidents that need their intervention. Therefore, fake events containing monitor-

ing information are generally not encouraged except when they are intentionally created for certain kinds of estimation mentioned above.
- As hidden conflicts of the first five classes can occur due to control applications being deprived of their interested events or the modification of the content of control applications' interested events, fake events related to requests for instructions of packet handling are more relevant. In OpenFlow SDN, such events are named *packet-in events*; a packet for a packet-in event must be complete, i.e, it contains all relevant headers of a normal packet.

Fake events for the first hidden conflict class: *event suppression by local handling* It is conceivable that the presence of any active rule (see Definition 5.20) in the data plane not escalating matched packets to the control plane can be the source of this hidden conflicts class. A fake event can be generated for a packet matched by an existing rule to probe for reaction of control applications. Traffic completing the two rules causing a local conflict whose match spaces show the subset or superset relationship is more notable as candidate traffic for the fake events of this class, because one of the control application appears to be more granular in installing its rules possibly as a reaction to a packet-in event, its behaviour is accidentally disabled by the other application installing the broader rule, the contention (manifested as a local conflict) also reveals the possibility that these applications are interested in events related to some common traffic.

The number of such fake events is generally large. An event needs to specify the SDN device's identifier (e. g., datapath ID in OpenFlow SDN), the complete header of a packet and possibly the packet payload. Consider an exemplary rule:

priority = 2, ipv4_dst = 192.168.2.0/24, protocol = UDP, dst_port = 5001, action = [output:3]

a packet matched by this rule can have any source IPv4 address and source UDP port, its destination IPv4 address can be one of the 256 values within the IP range 192.168.2.0/24, not to mention the layer 2 headers. The number of fake events for this rule in the worst case is therefore greater than Ω:

Ω = *the number of source IP addresses * the number of destination IP addresses * the number of source UDP ports* = $2^{32} \cdot 2^8 \cdot 2^{16} = 2^{56}$

Suppose that each probe takes 1ms, the total duration needed is $\approx 2.2 \cdot 10^6$ years; to probe this number of fake events to detect conflicts of the first hidden conflict class is impractical!

We may perform the probing with selected samples from the rule's match space to reduce the number of fake events. The existence of a conflict among these probes would be sufficient for the correct conclusion. However, if no conflict is found, the conclusion of a conflict-free setting would be lame.

Fake events for the second and third hidden conflict classes: *event suppression by upstream traffic looping/dropping* We can infer from the rule graph (see Section 5.4.2) the slices of traffic that get dropped or stuck in a loop based on vertices

having the patterns $(X, Y, -1)$ or $(X, Y, -4)$, the corresponding valid traffic paths are also available. For each traffic slice and its path, if the switch under consideration is not inside that path and there is no rule matching a dropped/looped traffic slice in that switch, that traffic becomes the candidate for predicting these hidden conflict kinds (the presence of rules matching this traffic would lead to the distributed conflict *policy suppression by upstream traffic looping/dropping* as discussed in Section 5.6.2 albeit with its interpretative nature). Similar to the case of the first hidden conflict class, the number of fake events is also exceedingly high, inhibiting its employment in the hidden conflict prediction.

Fake events for the fourth hidden conflict class: *event suppression by changes to path* Hidden conflicts of this class are caused by interested traffic of the control application in concern being forwarded around its target switches. From the perspective of the predictor which regards control applications as black-boxes, all information necessary for synthesizing fake events to probe this kind of conflict is not available, including target switches and traffic slices interested by a control application. This means that, to comprehensively detect conflicts of this class, we have to create fake events from all switches in the network; for each switch, the candidate packets are those matched by active rules in other switches which do not forward traffic to that switch (the candidate packets for this class are thus excluded from the set of candidate ones for the first class). Needless to say, the number of such fake events is large, making it not practical to probe all of them.

Fake events for the fifth hidden conflict class: *action suppression by packet modification* The cause for conflicts of this class is that modified packets escalated to the control plane are not interested by control applications while the original packets would be. Packets can be modified at the switch in examination before being sent to the control plane, or they were altered by rules in other switches before arriving at that switch. The candidate packets for fake events in the former case is intuitive, being those matched by the involved rules. In the latter case, all rules in the current switch needs to be inspected; for each rule, we need to trace back all chains of connected rules (see Definition 5.19) containing it in the rule graph to determine any change made to its matched packets; if there exists one or more rules in the chain modifying the packets, the set of unmodified packets corresponds to the candidate packets for the fake events of this class, given that there is no rule in the current switch matching this unmodified set (it would possibly become the first hidden conflict class otherwise). It would be much more complicated in case a packet is modified multiple times, we may have to check all sets of packets after each change before their arrival at the current switch. Another aspect to consider is that if the packets were unmodified, they might be handled differently in the network and did not reach the current switch, thus not causing hidden conflict of this class but of the fourth class. As we do not know the target switches of the control applications, we need to do the check for all switches. The mentioned complexity, together with

the other issues of the fake-event-based approach, keep us from delving further into tackling this hidden conflict class using this approach.

Control applications' reactions
A control application can register for different kinds of events. In the current approach, a control application cannot discern fake and genuine events, it always react by sending instructions to the data plane or simply doing nothing, upon receiving an event. However, stateful applications (see Definition 3.5.1) log also its reactions to fake events and behave differently once receiving genuine events later, therefore they do not function as expected in the presence of both fake and genuine events. Consider an example with a stateful end-point load balancer performing round-robin balancing for TCP sessions on three replicas based on packet-in events; for the first genuine event, it directs the corresponding TCP session to the first replica; during the probing process for hidden conflicts, it receives a fake event which it cannot discern, it reacts with rules sending this new TCP session to the second replica though these rules are intercepted and are not installed in the data plane; then a new event, being a genuine one, reaches this balancer, it notices that it has directed two TCP sessions to the first two replicas, thus it forwards this new session to the third replica; indeed, without the fake event, the latter TCP session would be handled by the second replica, not the third one; thus the fake event does trigger a problem.

Intercepting control applications' reactions to fake events
The reactions of control applications caused by fake events needs to be intercepted and analysed to detect hidden conflicts. It is challenging to differentiate reactions triggered by fake events and those by genuine events, and worse, active actions unrelated to any events can get involved. In principle, the predictor does not know if a control application reacts to the fake event until it catches the reactions. Some control application does not react to a fake event as it is not interested in that event or it is an active application (see Definition 3.5.1) not functioning based on unsolicited events. Therefore, for a single fake event, there can be no reaction, one or many reactions, the predictor needs to considers these cases. In this black-box approach, we have not figured out a solution to distinguish a reaction to a fake event and an active action generated in the meantime. For instance, the predictor sends a fake packet-in event, after a short period, it observes two requests of installing new rules in the data plane, it intercepts both to examine hidden conflicts, both rules are not deployed in the data plane at all; however, one request comes actually from the administrator and should be enforced; in this case, the mistaken interception leads to a problem.

Summary
In employing the speculative prediction of hidden conflicts, we have to face with the critical issues that cannot be remedied effectively and efficiently, including probing a large number of fake events, the state poisoning of stateful control applications,

and the obscure discrimination between control applications' reactions to fake events and their genuine actions. Therefore, we choose a "grey-box" approach in detecting hidden conflicts by asking for input from control applications concerning their interested kinds of traffic, target switches where they place rules and if they function based on packet-in events.

5.7.2 Detecting hidden conflicts with control applications' input

Certain input from control applications is required in detecting hidden conflicts to overcome the problems encountered in the black-box approach. The input corresponds to the required information of a packet-in event including the traffic kinds in which a control application is interested, in the form of packet headers, and the target switches of that application. An exemplary input file for hidden conflict detection of the End-point Load Balancer (EpLB), the Destination-based Path Load Balancer (PPLB4D) and the routing control applications is shown in Listing 5.1.

```
[eplb]
ts = 7 5 6 # a list of target switches
ipv4_src = 192.168.1.1 # a list of source IPv4 addresses
ipv4_dst = 192.168.1.3 192.168.1.4 # a list of destination IPv4 addresses
ip_proto = 6 17 # a list of protocols, which includes TCP and UDP in this case

[pplb4d]
ts = 3 4
ipv4_dst = 192.168.1.3, 192.168.2.0/24
ip_proto = 17 # UDP

[routing]
```

Listing 5.1: Exemplary input file for detecting hidden conflicts

A control application can be interested in some specific traffic kind as the EpLB application or any traffic as the routing application, the administrator may want to detect hidden conflicts for certain control applications or for all of them, these are specified in the hidden conflict input file. In the following, we show how to cope with hidden conflicts of the first five classes with information from the input file. Again, the other two classes require security measures to protect the control plane, which is out of scope of our work and is not discussed here.

The general idea is to form the match space of traffic interested by each control application from its input and compare it with rules in the data plane. The match space is denoted by a special representation form of matchmap using list (or array), which we name *list-matchmap*. For example, using the matchmap template
matchmap template: ingress port, EtherType, src IPv4, dst IPv4, IP protocol, src TCP port, dst TCP port, src UDP port, dst UDP port
the list-matchmap of EpLB from Listing 5.1 is represented by:

[-1], [2048], [192.168.1.1],[192.168.1.3, 192.168.1.4],[6, 17],[-1],[-1],[-1],[-1]
in which the EtherType value is inferred from that of an IP packet (0x800 or 2048 in decimal value), the value -1 of a field represents any value, the list is denoted by the square bracket pair ([]). This way of representation allows the inference of the field combinations to form the granular matchmap, thus enables the flexible specification of control applications' interested traffic in the input file, e. g., the two separate matchmaps inferred from the above list-matchmap of EpLB among others are:
-1, 2048, 192.168.1.1, 192.168.1.3, 6, -1, -1, -1, -1
-1, 2048, 192.168.1.1, 192.168.1.4, 17, -1, -1, -1, -1
The comparison of a rule's matchmap with a list-matchmap of a control application can be carried out directly based on the multi-property set and the $\cdot r$ operator according to Definition 5.3, Theorems 5.1 and 5.2 without having to infer the granular individual matchmaps of the list-matchmap. To illustrate the comparison, we take an exemplary rule R having the matchmap *{3, 2048, -1, 192.168.1.4, 6, -1, 80, -1, -1}*

- Each field in the rule's matchmap is compared with each field in the corresponding list of the list-matchmap. For example,
  + the first field of R is a subset of that of EpLB, their relationship is thus $r_1 = 2$, their overlap value is 3,
  + their second fields are equal, their relationship is $r_2 = 1$,
  + the third field of R is a superset of that of EpLB, their relationship is $r_3 = 3$ and the overlap value is 192.168.1.1,
  + the fourth field of R is a subset of EpLB's, their relationship $r_4 = 2$, their overlap value is 192.168.1.4,
  + the fifth field of R is a subset of EpLB's, their relationship and overlap value are $r_5 = 2$ and 6, respectively,
  + their sixth fields are equal, their relationship is $r_6 = 1$,
  + the seventh field of R is a subset of EpLB's, their relationship $r_7 = 2$ and the overlap value is 80,
  + their two other fields are equal, $r_8 = r_9 = 1$.
- Their overall relationship is
  $r = R \cdot_r$ (list matchmap of EpLB) = $r_1 \cdot_r r_2 \cdot_r r_3 \cdot_r r_4 \cdot_r r_5 \cdot_r r_6 \cdot_r r_7 \cdot_r r_8 \cdot_r r_9 = 2 \cdot_r 1 \cdot_r 3 \cdot_r 2 \cdot_r 2 \cdot_r 1 \cdot_r 2 \cdot_r 1 \cdot_r 1 = 4$
  which means that rule R and the list matchmap of EpLB are intersecting. Their final overlap is *{3, 2048, 192.168.1.1, 192.168.1.4, 6, -1, 80, -1, -1}*

The choice of rules for comparison varies depending on the hidden conflict class under consideration. We present in the following a method to detect conflicts of the first hidden conflict class. The other classes show prohibitively expensive cost for the detection and/or exhibit the similar interpretative nature as some distributed conflict classes (see Section 5.6.2), which restrains the solid judgement about the presence

of their conflicts. We discuss instead the approaches to cope with these kinds of conflicts and the practical implications.

Detecting conflicts of the first hidden conflict class: *event suppression by local handling*

The cause of hidden conflicts of this class is that the existing rules in the target switch(es) of a control application match and handle incoming traffic locally, thus no relevant event is generated to the application. We need to determine these rules by comparing each with the list-matchmap of the control application. Algorithm 9 shows how these rules are identified.

Algorithm 9 Pseudo-code for identifying rules causing hidden conflicts of class *Event Suppression by Local Handling* with input from control applications

Input: input file IF specifying control applications registering for the packet-in events, the characteristics of traffic interested by them and their target switches
Output: Alarms of rules causing hidden conflicts, the influenced traffic in the matchmap form and the influenced control applications.

Create from the input file IF a list of target switches LTS of all control applications, for each target switch TS_i, establish a list of list-matchmaps LMM_i corresponding to the characteristics of traffic interested by the control applications, each list-matchmap is associated with a control application
for Switch S in the list of target switches LTS **do**
  for list-matchmap LMM in the list of list-matchmaps associated with switch S **do**
    for Rule $(S, 0, R)$ in the set of rule that has not been checked for hidden conflicts in table 0 of switch S **do** ▷*Check table 0 as a packet coming to a switch will first be handled by rules in this table*
      if Rule $(S, 0, R)$ and list-matchmap LMM are associated with the same control application **OR** $(S, 0, R)$ is inactive **then** ▷*see Definition 5.20*
        continue ▷*do not check hidden conflicts for control applications against the rules installed by themselves, also skip inactive rules as they do not impact incoming packets and thus do not induce hidden conflicts*
      end if
      if LMM is the last list-matchmap in the list of list-matchmaps associated with switch S **then**
        Mark $(S, 0, R)$ as already checked for hidden conflicts ▷*So it will not be checked more than once for hidden conflicts*
      end if
      Trace the table pipeline in switch S to extract the sequence of actions conducted on packets matched by rule $(S, 0, R)$
      if The sequence of actions includes sending matched packets to the controller without modifying them **then**
        continue ▷*There's no hidden conflicts in this case*
      end if
      Re = (list-matchmap LMM) $._r$ (matchmap of rule $(S, 0, R)$) ▷*Relationship between two multi-property sets, see Section 5.1.2*
      if Re == 0 **then** ▷*disjoint*
        continue ▷*Rule $(S, 0, R)$ does not cause hidden conflicts, ignore it*
      end if
      I = (list-matchmap LMM) $\cap$ (matchmap of rule $(S, 0, R)$) ▷*Intersection between the multi-property sets, see Theorem 5.2*
      Raise an alarm of the presence of the first hidden conflict class caused by rule $(S, 0, R)$ in switch S with the influenced traffic I, and that the control application associated with the list-matchmap LMM is influenced by this hidden conflict
    end for
  end for
end for

Coping with the second and third hidden conflict classes: *event suppression by upstream traffic looping/dropping*

Once the rule graph is built, we can infer the slices of traffic that are dropped or looped and the associated valid paths (also rule chains[3]). For a path among those that does not contain the target switch(es) of the control application under consideration, the corresponding traffic slice of that path is compared with the list-matchmap representing the interested traffic of that application (a conflict of this second or third hidden conflict class does not occur if the path contains the target switch(es)). If there is an overlap between them, we can issue a warning on a potential conflict of these hidden conflict classes. As mentioned in Section 5.6.2, a warning indicates only the potential occurrence of conflicts and serves as guidelines for the network operator in inspecting possible incidents.

We could not conclude with certainty the occurrence of conflicts belonging to these classes due to their interpretative nature. Since the isolated deployment of each application to examine the expected network behaviour is generally inapplicable in practice (see Section 5.6.2), we do not know how the traffic would be handled if there were no incident of traffic looping/dropping. It could be modified or sent on some path not containing the target switch(es) of the control application in concern, or something else. However, the traffic looping/dropping related problems are commonly critical (unless they are intended as in case of a firewall filtering certain traffic) and are identified as a fundamental functions of the rule graph, independently from the measures against hidden conflicts. This mitigates the consequences of these two hidden conflict classes or even lessens them to be negligible.

Coping with the fourth hidden conflict class: *event suppression by changes to paths*

Given that the traffic interested by a control application is known, conflicts of this class cannot be detected just by examining rules in the target switch(es) of that control application, rather rules in the other switches need to be investigated to see if the relevant traffic is sent around that application's target switch(es). We notice that this conflict class and the first hidden conflict class are mutually exclusive. In case there are rules matching the interested traffic of the control application in its target switch(es), conflicts of the first class may occur and there exists no conflict of this class, and vice versa.

The idea in coping with this class' conflicts is to check if traffic interested by a control application appears in the network and is handled completely by rules in switches other than the target switch(es) of that application. This can be achieved by sniffing traffic at all switches' interfaces facing end-points, e. g., by using *tcpdump* tool. Once that traffic appears, we need to check the rule chain handling it and conclude

3 The terms *path* and *rule chain* are used interchangeably to facilitate the understanding in different contexts: a path is a sequence of vertices in the rule graph, each vertex corresponds to a rule. A path thus corresponds to a chain of connected rules, named *rule chain*.

the presence of conflicts of this class if no rule in the rule chain resides in the target switch(es) of the control application. Another way is to check the rule graph to extract all possible rule chains handling traffic interested by the control applications and give conclusion in a similar manner just mentioned.

The realization of the idea is practical in a small network with only a few number of end-points (e. g., less than 10), otherwise it turns out to be too expensive. Moreover, consider an example in which a control application states that it is interested in all traffic with destination TCP port of 80, then all end-points need to be considered, all rules including destination TCP port of 80 or any (wildcard) are involved. We can envision how expensive it is to detect this hidden conflict class in either way of dumping end-points' traffic or inspecting rule chains.

It is also uncertain how the interested traffic of a control application would be forwarded if it were not influenced by rules of the other that causes path changing. This interpretative aspect adds up to the complexity in detecting conflicts of this class.

Although a satisfiable solution in detecting this conflict class could not be provided, we find that it can be avoided by deploying the control application on the set of target switches in a way such that all of its interested traffic will be forwarded through at least one of these switches. For example, in experiment 8 in Chapter 3.6.3, the hidden conflict can be avoided if the PPLB4D2 application is deployed on both switches S5 and S6.

Coping with the fifth hidden conflict class: *action suppression by packet modification*

If a packet is modified in the target switch(es) of the control application before being sent to the controller, we can detect this kind of conflict in very much the same way as for the first hidden conflict class.

Otherwise, conflicts of this class and the first class are mutually exclusive. It can only occur for a control application in its target switches if the first class' conflicts do not exist for that application in those switches. For each rule in the target switches of the control application that escalates packets to the control plane, we need to trace back the rule chain connected to it in the rule graph (see Section 5.4.2), then check if the packet was modified by one of these rules and if the original packet or the intermediate packet is of interest of the considered control application. A warning for this kind of hidden conflict can be raised if this holds.

We take an example to illustrate how the candidate rule chains can be found. Assume that each rule table has at least a table-miss rule that sends packets to the controller (which is typical in OpenFlow SDN), then rules in the adjacent switches directing matched traffic to the target switches of the control application in concern need to be examined, because they can introduce traffic unmatched by any rules in these target switches but only the table-miss rule. If a rule satisfies this condition, all rule chains containing it are candidate for checking against hidden conflicts of this class.

A packet can be transformed multiple times before reaching the target switches. We can only tell that the packet arriving at the target switches used to be the one interested by the control application, but we could not tell with certainty how it would be if that packet were not modified, or if it were modified in some way beneficial to this application. This can only be asserted precisely in a separate test-bed replicating the current network and with isolated execution of the control application, but it is prohibitively expensive as discussed in Section 5.6.2. The conclusion on the presence of hidden conflicts of this class in the co-deployment of control applications is thus interpretative, a warning about their potential occurrence is more suitable.

Coping with the hidden conflict classes: *undue trigger* and *tampering with event subscription*

An attack can employ the mechanism similar to the speculative method employed in the hidden conflict predictor (see Section 5.7.1) to trigger control applications in installing rules or removing rules, which causes hidden conflicts of *undue trigger* class. If the SDN implementation allows the dynamic subscription/unsubscription for certain events by control applications, the attacker can also act as an application and register/deregister events once he is able to intercept the communication channel between the controller and applications, leading to the *tampering with event subscription* hidden conflicts.

In practice, all communication channels between the controller and the applications, and between applications need to be secured to avoid these kinds of attack, e. g., by using strong encryption techniques. In addition, all applications must be authorized before they can participate in controlling the network. Some approaches relevant for this purpose include FortNOX, which is a software extension of the NOX OpenFlow controller employing the role-based authorization and security constraint enforcement [88], and FRESCO, being a framework for security application development [95].

5.8 Complexity

The detection of conflicts includes the detection of the individual conflict classes: local, distributed and hidden conflicts. Consider a network with diameter d having s switches, each switch has at most t rule tables, each table contains at most r rules, there are a control applications, each specifying at most i interested traffic patterns. In building the rule graph for detecting distributed conflicts, we examine the worst case where a rule always has connections to all rules in its next hop (being the next rule tables or the first table in its next hop).

We show in Table 5.1 the worst-case complexity in detecting or coping with each conflict class according to our proposed measures.

- When a new rule is inserted in a rule table, it is checked against r rules in that table for local conflicts. The complexity in the worst case is $O(r)$, for all local conflict classes (LC1,2,3,4,5). It is also checked for hidden conflicts HC1 *event suppression by local handling* by comparing with all interested traffic patterns of each control application, the worst-case complexity is $O(ai)$.
- In building the rule graph, which includes the introduction/removal of a rule to/from the rule graph, the worst case induces the complexity of $O(r^{td-1})$ if that rule is the first rule in the longest rule chain handling the same traffic slice. The longest rule chain's length is td since it contains rules in each rule table of all switches along the network diameter. The best case complexity is $O(1)$ if that rule is in the end of the rule chain.
- Distributed conflicts are inferred from the rule graph, the worst-case complexity of their detection aligns with that of the rule graph building in the worst case. We notice that, as the loop in building the rule graph is detected and excluded from further establishment of the rule chain (also path), there would be no infinite rule chain arising due to possible loops. Once the rule graph is built, the detection of traffic looping or traffic dropping corresponds to the search for vertices of the loop/drop pattern. The worst-case complexity aligns with the maximum number of vertices in the rule graph, being $O(str)$. Our measures against the distributed and hidden conflicts related to traffic looping/dropping (DC1,2,3,4 and HC2,3) have therefore this worst-case complexity.
- The worst-case complexity of the measures to cope with distributed and hidden conflict classes related to *changes to paths* (DC7, HC4) is $O((s-1)r^{td})$ as we have to examine rules in all switches other than the target switches of the control application under consideration. For each of these switches, the number of rule chains to be checked is at most r^{td}.
- For those classes related to *packet modification* (DC5,6 and HC5), the worst-case complexity is $O(r^{td})$ in case we have to examine all the longest rule chains containing rules in the target switches of the control application under consideration.

The worst-case complexity for each class is assessed qualitatively as low, medium or significant in Table 5.1 for intuitiveness. As mentioned in Section 5.7, the hidden conflict classes *undue trigger* and *tampering with event subscription* require security measures in the control plane and are not examined further in our work.

5.9 Practical implications and conclusions

We have introduced the general tools including *multi-property set, $\cdot r$ operator* and its algebra, *match-map, actmap, rule graph*, and presented their application in detecting or coping with conflicts in SDN. These tools enable the intuitive and efficient detection of local conflicts, distributed conflicts of the classes *policy suppression by downstream traffic looping/dropping* and hidden conflicts of the class *event suppres-*

Class	Target Rules	Worst-case Complexity	Output
LC1,2,3,4,5	rules in TS	$O(r)$, low	rules in TS causing LC
DC1,2,3,4	rules (vertices) in RG exposing loop/-drop patterns	$O(str)$, medium	rule chains in RG causing DC
DC5,6	rules in TS and rule chains in RG containing them	$O(r^{td})$, significant	rules chains in RG causing DC
DC7	rules in switches other than TS	$O((s-1)r^{td})$, significant	rules chains in RG causing DC
HC1	rules in TS	$O(ai)$, low	rules in TS causing HC
HC2,3	rules (vertices) in RG exposing loop/-drop patterns	$(O(str))$, medium	rule chains in RG causing HC
HC4	rules in switches other than TS	$O((s-1)r^{td})$, significant	rule chains in RG causing HC
HC5	rules in TS and rule chains in RG containing them	$O(r^{td})$, significant	rule chains in RG causing HC

Table 5.1: Worst-case complexity of the measures to detect or to cope with conflicts. LC: Local Conflict, DC: Distributed Conflict, HC: Hidden Conflict, RG: Rule Graph, TS: Target switches of the control applications. The conflict classes in the first column are numbered for brevity in this table, the mapping of class name-number is shown in Table 5.2.

sion by local handling. The derivation of the other classes' conflicts experiences the interpretative nature, some with extremely high complexity. We have discussed alternatively the measures to cope with them, in which warnings are issued for potential conflicts.

	Class number in Table 5.1	Class name
Local Conflicts (LC)	1	Shadowing
	2	Generalization
	3	Redundancy
	4	Correlation
	5	Overlap
Distributed Conflicts (DC)	1	Policy suppression by downstream traffic looping
	2	Policy suppression by upstream traffic looping
	3	Policy suppression by downstream traffic dropping
	4	Policy suppression by upstream traffic dropping
	5	Policy suppression by downstream packet modification
	6	Policy suppression by upstream packet modification
	7	Policy suppression by changes to paths
Hidden Conflicts (HC)	1	Event suppression by Local Handling
	2	Event suppression by upstream traffic looping
	3	Event suppression by upstream traffic dropping
	4	Event suppression by changes to paths
	5	Action suppression by packet modification

Table 5.2: Mapping between the names of conflict classes and numbers in Table 5.1

In practice, traffic dropping and traffic looping, if not intended (e. g., by a firewall), need to be detected and resolved. The conflicts related to these incidents are therefore remedied accordingly, they include the four distributed conflict classes *policy suppression by downstream/upstream traffic looping/dropping* and the two hidden conflict classes *event suppression by upstream traffic looping/dropping*. Our proposed *rule*

graph paves the way for effective methods to identify traffic looping and dropping in the network. The cases in which a packet is modified multiple times in transit can also be alarming and worth a close examination. The rule graph is handy in determining them according to the algorithm proposed by Reyes (see Algorithm 8 in Section 5.6.2). The two distributed conflict classes *policy suppression by downstream/upstream packet modification* and the hidden conflict class *action suppression by packet modification* are tackled by this manner. It is prohibitively expensive to detect distributed and hidden conflicts related to *changes to paths*. However, these can be prevented by choosing the target switches of a control application in such a way that all of its interested traffic must flow across these switches. The two remaining hidden conflict classes *undue trigger* and *tampering with event subscription* need to be handled by security measures in the control plane and are not inspected in detail.

In the next chapter, we demonstrate the realization of the concepts presented in this chapter via a prototype to detect local conflicts, traffic looping and dropping in general, and hidden conflicts of the class *event suppression by local handling*. The work of Reyes [90] in identifying traffic modified multiple times in transit by employing the rule graph supplements our results.

6 Prototypical Implementation and Evaluation

Having established the conceptual methodology in Chapter 5, we demonstrate its realizability via a conflict detection prototype. The arguments on the practical aspects in detecting conflicts (see Chapter 5.9) drive our choice of functionalities for the prototype, which aims at detecting local conflicts, general traffic looping and dropping, and hidden conflicts of the class *event suppression by local handling*. The prototype is extended by Reyes [90] to identify conflicts related to packet modification, supplementing the results of our work.

The prototype has been incrementally enhanced during the process of extracting conflict patterns and properties according to the methodology in Chapter 3.7, therefore, we consider it to have been already evaluated against the collected dataset by this process. In this chapter, we evaluate the prototype in two manners: designed and randomly checked. We deploy rules in the data plane with known conflicts and have the prototype identify them in the designed cases, the results are then controlled manually. In the randomly checked cases, the framework for automating experiments (see Chapter 3.4) with the integrated conflict detection prototype is employed to perform a large number of tests. When the number of conflicts detected by the prototype exceeds a preset threshold, we select randomly a set of samples and analyse them to confirm the outcome. The evaluation results indicate that the prototype is sound and complete in detecting conflicts for the designed case. The number of conflicts found in the latter case is large, causing the manual control process much time-consuming for the whole set of detected conflicts, however, the quality of soundness can be asserted on the selected conflict samples.

Finally, we share the insights acquired while developing and evaluating the prototype, which would benefit its employment in practice.

6.1 Conflict detection prototype

We have implemented a conflict detection prototype, which we name *Conflict Detector*, based on Ryu[1], a component-based SDN framework completely realized in the Python programming language. The practical aspects in detecting conflicts (see Chapter 5.9) justify our decision on the functionalities of the detector: it covers all local conflict classes, is able to detect the general traffic looping and traffic dropping, and the hidden conflict class *event suppression by local handling*. Reyes [90] extends the detector to cope with conflicts related to packet modification, we portray the result from this part of his work as a supplement for our work. The prototype's codebase is available online[2].

1 https://ryu.readthedocs.io
2 https://github.com/mnm-team/sdn-conflicts

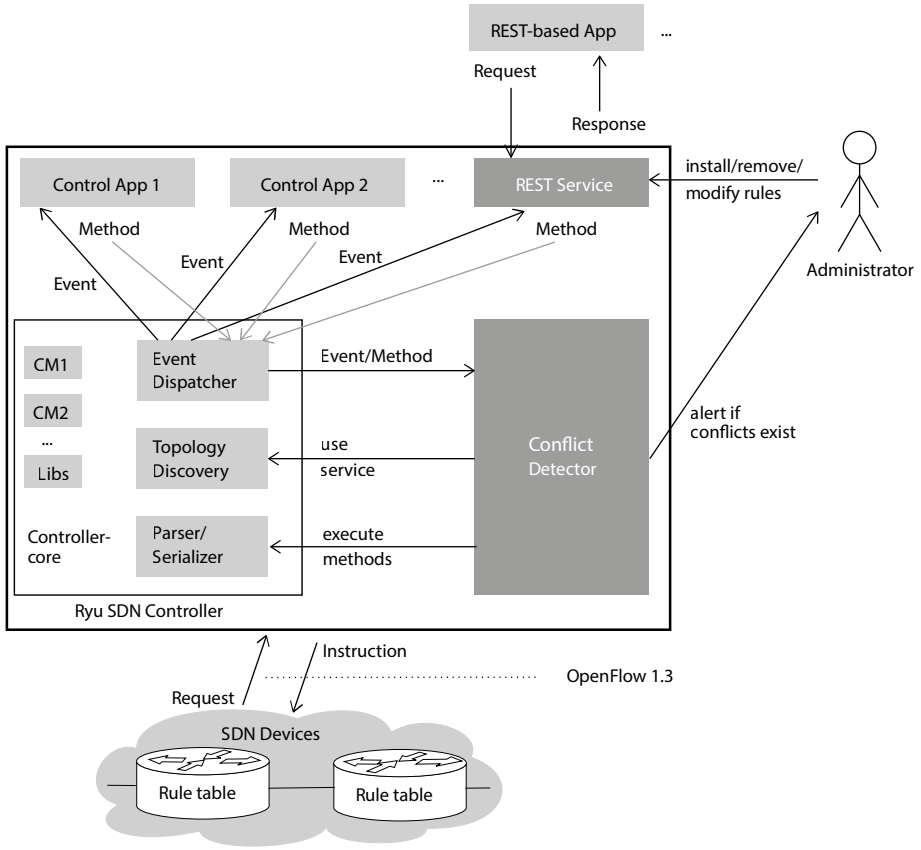


Figure 67: Communication of the conflict detector with other controller modules (CM) of the Ryu SDN framework. The *Conflict Detector* and *REST Service* boxes are our added components.

6.1.1 Overview

The conflict detector is implemented as a controller built-in application. Figure 67 shows its communication with other controller modules, including Topology Discovery, Event Dispatcher and Parser/Serializer.

- Event Dispatcher: each message from network devices sent to the controller is wrapped in an event and is dispatched by the Event Dispatcher module to other controller modules or control applications that have registered for that event. Control applications can also send events to each other via the Event Dispatcher, this mechanism is exploited by the conflict detector to observe the methods/reactions from other control applications to maintain the rule database and detect conflicts therein.
- Parser/Serializer: this module is responsible for transferring messages from the controller to the data plane, which includes installing/modifying/removing rules

in the data plane, sending packets (packet-out messages) or requests (e. g., to ask for statistical data or rule tables of a device) to the data plane. The conflict detector, on the one hand, intercepts methods from control applications to build the rule database, on the other hand, asks the Parser/Serializer to realize these methods in the data plane. Note that the communication of this module is one-way, only from the controller to the data plane. Any response from the data plane to the controller is handled by the Event Dispatcher module. Therefore, although the conflict detector sends its requests, e. g., for retrieving the data plane's rule tables, to the Parser/Serializer, it gets the response from the Event Dispatcher module.

- Topology Discovery: this controller module discovers and maintains the state of the network devices connected to the controller. Any introduction or removal of a switch's interface, a switch itself or a link is notified by this module via the corresponding topology-related events to the interested control applications. The detector indirectly uses the service of the Topology Discovery module to maintain an updated global view of the network.

The conflict detector alerts the administrator if there exist conflicts in the data plane or the rules to be installed will lead to conflicts. The detector can work in two modes: passive or active. In the active mode, it does not allow a new rule to be deployed if that rule would yield conflicts. In the passive mode, it informs the administrator of the situation while letting the rule be deployed in the meantime. For examining conflicts and evaluating the prototype, we execute the detector in the passive mode by default.

We implemented a REST service that allows the REST-based applications (see Chapter 3.5.1) or the administrator to interact with the network. In order to detect conflicts from these sources, this REST service is treated as a normal control application, i. e., its methods are also intercepted and analysed by the conflict detector.

The communication between the controller and the data plane is performed via the OpenFlow 1.3 protocol [79].

6.1.2 Conflict detector as a Ryu application

Figure 68 shows the classes on which the prototype is built. The *RyuApp* is shipped with the Ryu framework, the other classes are implemented on our own. A Ryu application is a subclass of the *RyuApp* class. The *TopologyDiscovery* class uses the service of the existing *Topology Discovery* controller module to maintain the updated global view of the data plane, this information is important in detecting conflicts, e. g., in tracking how a certain packet is handled in the network to reason about distributed conflicts. Our prototype is realized in the *ConflictDetector* class which inherits the *ARPCache* class, *ARPCache* is a subclass of the *TopologyDiscovery* class.

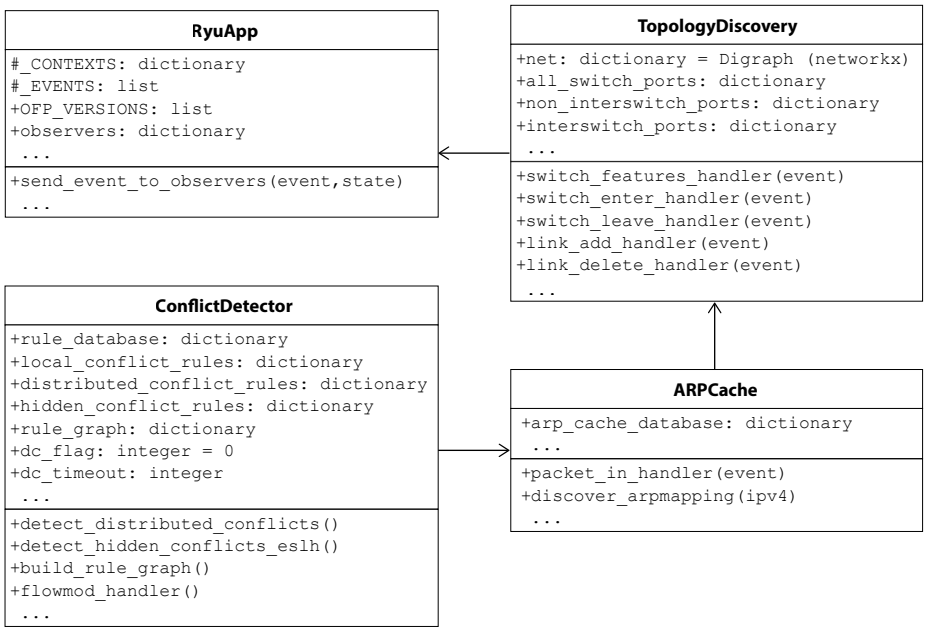


Figure 68: The class diagram of the conflict detector. The visibility of the classes' attributes and operations accords with the UML conventions, i. e., +: *public*, -: *private*, #: *protected*.

TopologyDiscovery class

This class maintains the topology information based on the topology-related events, e. g., switch leaving/entering, link adding/deleting events. The network topology is encoded in a directed graph implemented by the *networkx*[3] library.

ARPCache class

This class maintains the mapping between the MAC addresses and IPv4 addresses of all end-points involved in transmitting traffic in the data plane. It also stores the information regarding the datapath ID of the switch that an end-point is connected to and on which port number of that switch. The information provided by the *ARPCache* and *TopologyDiscovery* classes is sufficient to infer all possible paths (including the shortest path) between two given end-points, which is leveraged by some control applications, e. g., shortest path first routing.

Utility class

In addition to the above classes, we introduce the *Utility* class (not shown in Figure 68). All control applications employed for our study of conflicts must inherit this class to invoke the *add_flow, modify_flow* or *remove_flow* functions once they wish to make some change in the data plane. Thereby all methods from control appli-

3 https://networkx.org/

cations are relayed by the *Event Dispatcher* controller module to the conflict detector, which analyses them to explore conflicts.

6.1.3 Building the rule database

The conflict detection prototype aims at detecting conflicts between control applications that install rules handling TCP or UDP traffic with IPv4 addresses. These rules range from a broad rule that specifies only a single match field, e. g., the ingress port of a packet in a switch or the destination MAC address, to a much more specific rule with most match fields specified. With that goal and the knowledge of the control applications involved, we choose the following *matchmap template* and *actmap template*:

***matchmap template**: ingress port, EtherType, src IPv4, dst IPv4, IP protocol, src TCP port, dst TCP port, src UDP port, dst UDP port*

***actmap template**: new src IPv4, new dst IPv4, new src TCP port, new dst TCP port, new src UDP port, new dst UDP port, list of output ports*

Each request from a control application on change in the data plane corresponds to an event, which is relayed by the *Event Dispatcher* module to the conflict detector. If a new rule governs TCP or UDP traffic, it is stored in the rule database of the conflict detector. Its match fields and actions are mapped to the chosen *matchmap template* and *actmap template* to obtain its *matchmap* and *actmap* according to Definitions 5.9 and 5.13 in Chapter 5.2. The list of next hops necessary for the detection of conflicts spanning multiple devices (e. g., distributed conflicts) is inferred from the rule's action and the information of the *TopologyDiscovery* class. Consequently, a rule entry in the rule database looks like:

switch's datapath ID, table ID, rule number, cookie, cookie mask, hard timeout, idle timeout, priority, match, action, next hops, matchmap, actmap

The tuple (switch's datapath ID, table ID, rule number) identifies a rule in the whole rule database.

If a rule governs layer 3 traffic but specifies only layer 2 addresses, e. g., it contains MAC addresses but not IPv4 addresses, these addresses are mapped to the corresponding layer 3 addresses based on the information from the *ARPCache* class. Hence, we do not formulate the local conflicts of *imbrication* class caused by rules specifying addresses in different layers as pointed out in the work of Pisharody [86].

If the new rule does not govern TCP or UDP traffic, e. g., its field of EtherType is not 0x0800 (for IPv4) or its IP Protocol field is 1 (for ICMP traffic), this rule is not stored in the rule database or checked against conflicts but gets deployed right off in the data plane. Therefore, a number of unrelated rules, such as rules handling ARP, ICMP traffic, are excluded from the rule database, which enhances the performance of the conflict detector.

6.1.4 Conflict detector's mechanics

The conflict detector is executed in two major threads: the main thread responsible for detecting local and hidden conflicts, and the other thread responsible for the distributed conflict detection. Its mechanics is illustrated in Figure 69.

- The main thread intercepts all methods/reactions from control applications to build the rule database, and to control the variables triggering the detection of distributed conflicts in the other thread. Every time a new rule is to be installed, the main thread checks for local and hidden conflicts if that rule is realized according to Algorithm 6 in Chapter 5.5. By default, the detector operates in the passive mode, i. e., it lets a rule deployed even if that rule might be in conflict with others. In this mode, the request from a control application is first implemented in the data plane to minimize the reaction time of the control plane, before other steps are carried out by the conflict detector. In the active mode, conversely, a request must be checked against local and hidden conflicts before it can be deployed in the data plane, thus causing higher latency in the reaction time of the controller.
- The distributed conflict thread executes in a loop that checks *dc_flag* in every *dc_timeout* period. If *dc_flag* is set to 1, it detects distributed conflicts by building or updating the rule graph from the rule database as described in Chapter 5.4. Note that every time a request (as an event) to install a new rule or remove an existing rule emerges, the main thread clears *dc_flag*, processes the request, and sets this flag afterwards, thus the detection of distributed conflicts is only performed if no request comes up during the *dc_timeout* period. Moreover, it is executed only once after the last request and is triggered again only by the next request. This tactic has the following advantages.

  – Reducing the overhead in detecting distributed conflicts compared to the case in which it is called every time a rule is installed or removed. In other words, the detection of distributed conflicts does not necessarily run every time a single rule is installed or removed, but only after a "round" of new rules are deployed or existing rules are removed (note that such a "round" can contain only one rule).
  – Reducing the probability of misleading conclusion of distributed conflict cases. When a packet flow comes into the network, a sequence of rules can be installed in different devices to handle it. The detection of distributed conflicts on each installation of a single rule could cause the detector to mistakenly alert of traffic dropping due to lacking of rule handling a new packet flow. By waiting for a timeout period after the last rule deployment, we assume that all rules handling the new packet flow have been completely installed.

  However, it has the disadvantage that distributed conflicts can only be detected after the rules causing them were deployed instead of a possibly expected measure

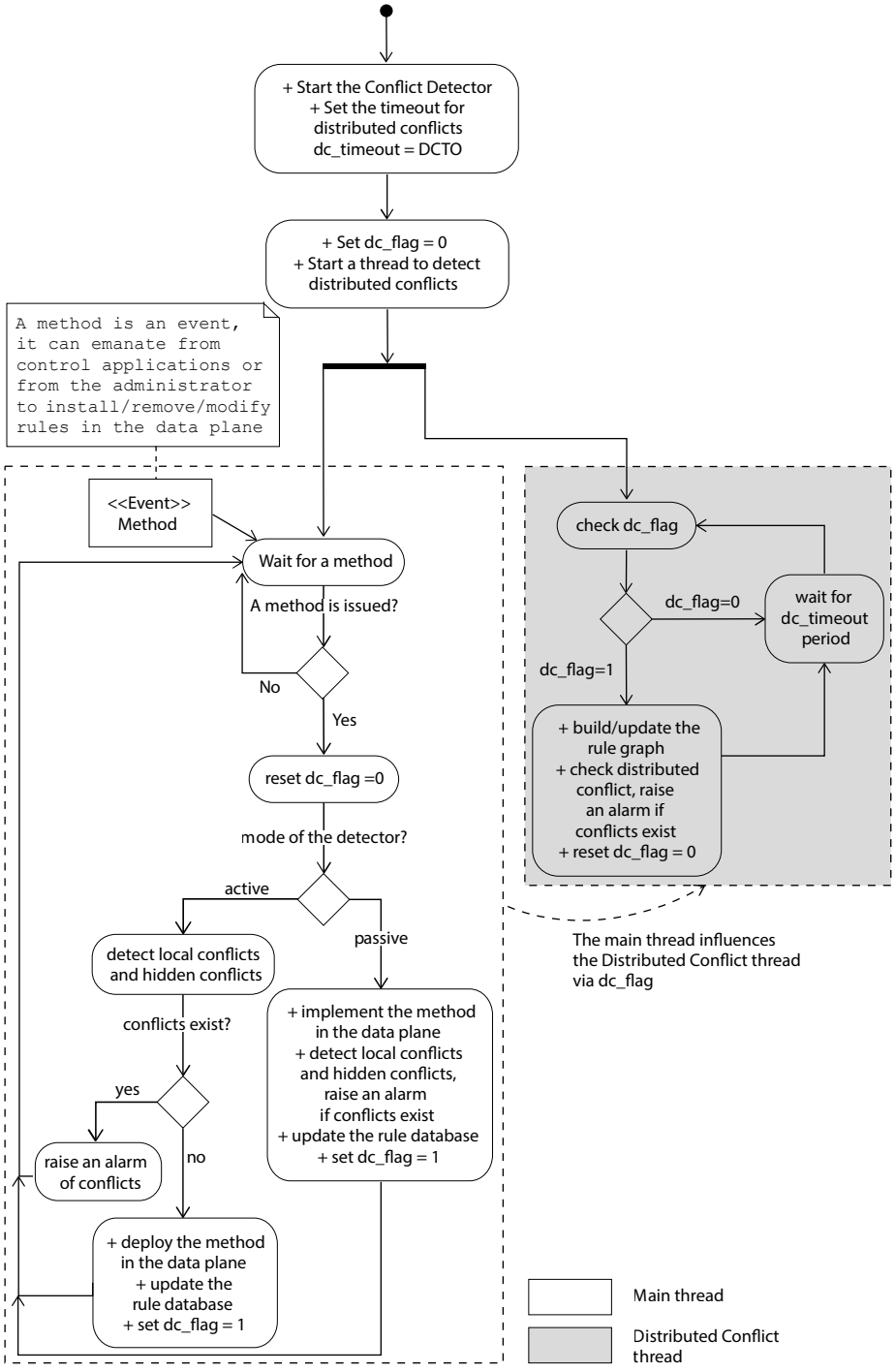


Figure 69: The mechanics of the conflict detector. The Distributed Conflict thread is meant to detect distributed conflicts, in our case it targets the general traffic looping and traffic dropping. Reyes [90] implements the detection of conflicts related to packet modification also in this thread.

to prevent them from happening. In this version of our detector, the thread for distributed conflicts detects general traffic looping and traffic dropping. The detection of conflicts related to packet modification implemented by Reyes [90] is also a part of this thread.

Detecting local conflicts
Upon receiving a new rule to be installed in a rule table of a switch that governs TCP or UDP traffic, the conflict detector collates the tuple (priority, matchmap, actmap) of the new rule with that of each rule in the same rule table to detect local conflicts after the Algorithm 6 presented in Chapter 5.5. The detector reports all rule pairs exposing local conflicts and the conflict classes. The detector also stores this information using the reference number of the rule in the rule database, i. e., the tuple (switch's datapath ID, table ID, rule number) (see Section 6.1.3), this information is exploited in detecting distributed conflicts.

Detecting traffic looping and traffic dropping
The conflict detector builds the rule graph from the rule database to detect traffic looping and traffic dropping as described in Chapter 5.6.1. The rule graph is a directed graph implemented by the *networkx*[4] library. This process excludes the inactive rules (see Definition 5.20 in Chapter 5.4.1) due to the *redundancy* or *shadowing* local conflicts. Each case of traffic looping/dropping is reported together with a sequence of involved rules.

Detecting hidden conflicts of the class *event suppression by local handling*
Hidden conflicts of this class are detected according to Algorithm 9 in Chapter 5.7.2, in which each control application specifies its target traffic as header fields, target switches and if it is interested in packet-in events. The detection is carried out in the main thread in a similar manner applied for local conflicts.

Detecting conflicts related to packet modification
These conflicts are determined based on Algorithm 8 in Chapter 5.6.2. The conflict detector is extended by Reyes [90] to implement this algorithm. The detection is performed in the distributed conflict thread, the rule pair transforming a certain slice of traffic (i. e., that traffic is modified at least twice) are reported if found.

6.1.5 Execution of the conflict detector

The detector program *detector.py* can be executed as a normal Ryu application. For instance, to detect conflicts between the two control applications End-point Load Balancer (EpLB) and Path Load Balancer (PLB), the following command is invoked:

4 https://networkx.org

```
ryu-manager --observe-links detector.py eplb.py plb.py
```

The *observe-links* option enables the topology discovery service required by the detector to maintain the global overview of the network, which is necessary in building the rule graph for detecting conflicts related to rules in multiple devices.

Our REST service, named *utility_rest.py*, can be co-deployed with other control applications and the detector, so that rules can also be installed via REST APIs by REST-based applications or by the administrator:

```
ryu-manager --observe-links detector.py utility_rest.py eplb.py plb.py
```

A rule can then be installed via the *curl* utility[5] and conflicts are raised by the detector if any. An example of the rule installation via *curl*:

```
curl -X POST -d '{"dpid":1,"match":{"in_port":1, "eth_type":2048, "ipv4_src
    ↪ ":"192.168.1.1", "ipv4_dst":"192.168.1.3"}, "actions":[{"type":"OUTPUT","
    ↪ port":2}]}' http://<controller IP>:8080/utility/addrule
```

The detector can also be integrated into our framework for automating experiments to detect conflicts in a massive number of experiments. As described in Chapter 3.4.7, the command used by our framework to deploy applications in the controller is:

```
ssh -n controller "sh -c 'ryu-manager --observe-links $applist &' > /dev/null
    ↪ 2>&1"
```

in which *$applist* contains the list of control applications, e. g., *plb.py, eplb.py*. The integration of the detector is accomplished by appending the program *detector.py* into this *$applist* variable. The reproduction of our experiments, including those used for evaluating the conflict detector, can be achieved by following the instructions mentioned in Chapter 3.4.7. The steps to integrate a new control application into the framework for experiments are also described there.

6.1.6 Output of the conflict detector

The detector logs various information periodically throughout the experiments. The information related to the installed rules, the rule graph and the conflicts is most important in terms of conflict detection. A sample of this information is:

```
self.ft =
1:{0: {1: (512, 2, OFPMatch(oxm_fields={'eth_type': 2048, 'in_port': 1, 'ipv4_src': '172.16.1.1', 'ipv4_dst':
    ↪ '172.16.1.3'}), [OFPActionOutput(len=16,max_len=65509,port=3,type=0)], [2], (1, 2048, 0,
    ↪ '172.16.1.1', '172.16.1.3', -1, -1, -1, -1), (-1, -1, -1, -1, -1, -1, {3})),
```

5 https://linux.die.net/man/1/curl

```
        2: (513, 3, OFPMatch(oxm_fields={'eth_type': 2048, 'ipv4_dst': '172.16.1.3'}), [OFPActionOutput(len
            ↪ =16,max_len=65509,port=4,type=0)], [3], (-1, 2048, 0, -1, '172.16.1.3', -1, -1, -1, -1), (-1, -1, -
            ↪ 1, -1, -1, -1, {4}))}}
2:{0: {1: (512, 2, OFPMatch(oxm_fields={'eth_type': 2048, 'in_port': 1, 'ipv4_src': '172.16.1.1'}), [
      ↪ OFPActionSetField(ipv4_src='172.16.1.2'), OFPActionOutput(len=16,max_len=65509,port=2,type
      ↪ =0)], [-2], (1, 2048, 0, '172.16.1.1', -1, -1, -1, -1, -1), ('172.16.1.2', -1, -1, -1, -1, -1, {2})),
        2: (513, 3, OFPMatch(oxm_fields={'eth_type': 2048, 'ipv4_dst': '172.16.1.3'}), [OFPActionOutput(len
            ↪ =16,max_len=65509,port=2,type=0)], [-2], (-1, 2048, 0, -1, '172.16.1.3', -1, -1, -1, -1), (-1, -1, -
            ↪ 1, -1, -1, -1, {2}))}}
...

number of nodes in rule graph = 18
number of edges in rule graph = 17
((3, 0, 2), (2, 0, 2)) {((3, 0, 2), (2, 0, 2)): ((-1, 2048, 0, -1, '172.16.1.3', -1, -1, -1, -1), 3),
                        ((5, 0, 2), (3, 0, 2), (2, 0, 2)): ((-1, 2048, 0, -1, '172.16.1.3', -1, -1, -1, -1), 3),
                        ((1, 0, 2), (3, 0, 2), (2, 0, 2)): ((-1, 2048, 0, -1, '172.16.1.3', -1, -1, -1, -1), 3)}
((4, 0, 1), (2, 0, 2)) {((4, 0, 1), (2, 0, 2)): ((-1, 2048, 0, '172.16.1.1', '172.16.1.3', -1, -1, -1, -1), 3)}
((5, 0, 4), (5, 0, -1)) {((3, 0, 5), (5, 0, 4)): ((-1, 2048, 0, '172.16.1.2', '172.16.1.6', -1, -1, -1, -1), -1)}
((4, 0, 3), (4, 0, -4)) {((3, 0, 3), (4, 0, 3), (5, 0, 3), (3, 0, 4), (4, 0, 3), (4, 0, -4)): ((-1, 2048, 0, '172.16.1.4',
      ↪ '172.16.1.5', -1, -1, -1, -1), -1)}
...

self.lc_cfl_rules =
1:{0: {1: [(2, 'class Shadowing2 (local conflicts)', (2, 3, 1))]}}
2:{0: {1: [(2, 'class Correlation2 (local conflicts)', (2, 4, 1))]}}
...
Number of local conflicts by dpid: [6, {1: {'sha2': 1}, 2: {'cor2': 1}, 3: {'gen1': 1}, 4: {'red': 1}, 5: {'cor3': 1}, 6: {'
      ↪ ove': 1}}]
Number of local conflicts by class: [6, {'gen1': 1, 'cor3': 1, 'cor2': 1, 'sha2': 1, 'red': 1, 'ove': 1}]

self.hc_cfl_rules =
'pplb4s': [((2, 0, 2), [[-1], 2048, [6, 17], ['192.168.1.3'], ['172.16.1.3'], [-1], [-1], [-1], [-1]])]
Number of rules causing hidden conflicts by control app: {'pplb4s': 1}

self.dt_cfl_rules =
1:[(((3, 0, 3), (4, 0, 3), (5, 0, 3), (3, 0, 4), (4, 0, 3), (4, 0, -4)), (-1, 2048, 0, '172.16.1.4', '172.16.1.5', -1, -1, -1, -1))
      ↪ ]
2:[(((3, 0, 5), (5, 0, 4)), (-1, 2048, 0, '172.16.1.2', '172.16.1.6', -1, -1, -1, -1))]
Number of distributed conflicts by class: {'drop': 1, 'loop': 1}
```

The variable *self.ft* stores the rule tables containing all rules governing interested traffic, which is TCP/UDP traffic in this case. The sample shows rules in table 0 of devices 1 and 2, encoded with the *dictionary* data type of the python programming language. For example, rule (1,0,2) has *cookie* 513, priority 3, next hop as device with ID 3, matchmap *(-1, 2048, 0, -1, 172.16.1.3, -1, -1, -1, -1)*, actmap *(-1, -1, -1, -1, -1, -1, 4)*.

The rule graph has 18 nodes (vertices) and 17 edges, four edges are shown in the sample together with their attributes in the form of *(path: (matchmap, priority))*. The edge ((5, 0, 4), (5, 0, -1)) denotes a *traffic dropping* distributed conflict, the edge ((4, 0, 3), (4, 0, -4)) embodies a *traffic looping* distributed conflict.

The detected local conflicts are logged in the variable *self.lc_cfl_rules*, rule (1, 0, 1) in the sample is shadowed by rule (1, 0, 2), the conflict pattern is (2, 3, 1). The local conflict patterns are encoded as tuples of numbers in our implementation, each tuple reflects the relationship between two rules in terms of (priority relationship, match relationship, action relationship), the pattern (2, 3, 1) indicates that the former rule has lower priority than the latter rule, its match space is a proper subset of the latter and their actions are different (see Chapter 5.5 for more details).

The detected hidden conflicts are stored in the variable *self.hc_cfl_rules*, the sample reports that rule (2, 0, 2) causes a hidden conflict of the class *event suppression by local handling* to the control application PPLB4S (Source-based Passive Path Load Balancer).

The variable *self.dt_cfl_rules* holds information of the detected distributed conflicts. A *traffic looping* conflict and a *traffic dropping* conflict are notified, the path and matchmap of the influenced traffic are shown.

We are able to verify the conflicts arising based on the above logged information. For instance, a local conflict between two rules reported in the variable *self.lc_cfl_rules* can be controlled by checking their details regarding the priority, match fields and actions in the rule tables stored in the variable *self.ft*, a distributed conflict can be examined by referring additionally to the edges of the rule graph. These tactics are applied in evaluating the conflict detector in the next section.

6.2 Evaluation

We evaluate the conflict detector based on the test-bed environment described in Chapter 3.4.1. A set of control applications are tested on two new network topologies besides the REST service that is employed for installing rules with pre-designed conflicts. To be able to confirm the realizability of the conflict detector, we attempt to evaluate the soundness and completeness of its detection results. These qualities are controlled in the designed and in randomly checked scenarios.

6.2.1 Network topologies

We build the evaluation test-beds associated with the topologies simulating the Munich Scientific Network (MWN) backbone[6] and the Stanford University's backbone network [50]. The MWN backbone connects different university campuses, institutions in Bavaria, Germany to serve the research, teaching and study demands. The network is complex, as of 2019, it is constituted of 91 backbone routers, more than 1900 switches and more than 5000 LANs, the statistic data in 2017 shows that up to 220.000 devices connected to this network within a week, it received ≈1.6 PByte/-month and sent out ≈0.7 PByte/month. Our topology (Figure 70) simplifies most of

6 https://www.lrz.de/services/netz/mwn-ueberblick_en/

these aspects while preserving only the backbone network "shape" with 21 switches and uses 21 representative end-points for generating traffic. Similarly, the Stanford University network (Figure 71) containing 26 switches distributed in three layers (backbone, distribution, access) is connected with 14 symbolic end-points.

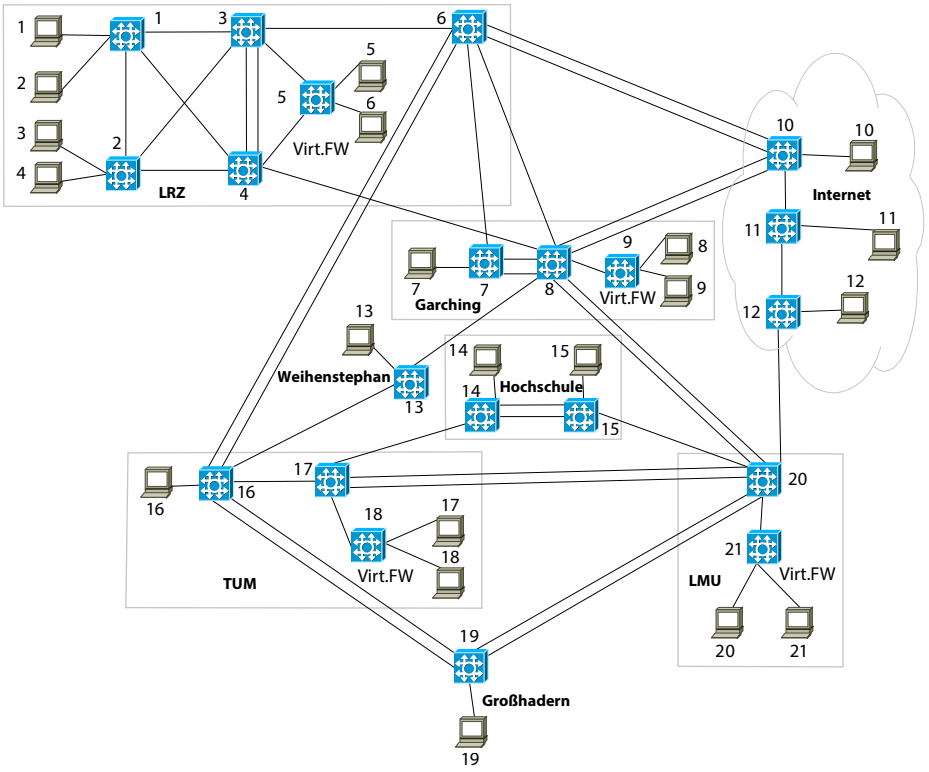


Figure 70: The simulated topology for the MWN's backbone network[7]

6.2.2 Evaluation results in designed cases

We deploy the REST service, named *utility_rest.py*, at the controller and install rules via the *curl* utility as mentioned in Section 6.1.5.

```
ryu-manager --observe-links detector.py utility_rest.py
```

The rules are designed with known conflicts. An example of a designed *shadowing* local conflict and a *traffic looping* distributed conflict on the MWN topology is:

```
{"comment": "LC: shadow2"}
```

7 https://www.lrz.de/services/netz/mwn-ueberblick/backbone.png

{“dpid”:1, “cookie”:“0x200”, “priority”:2, “match”:{“in_port”:1, “eth_type”:2048, “ipv4_src”:“172.16.1.1”, “
↪ ipv4_dst”:“172.16.1.3”}, “actions”:[{“type”:“OUTPUT”,“port”:3}]}
{“dpid”:1, “cookie”:“0x201”, “priority”:3, “match”:{“eth_type”:2048, “ipv4_dst”:“172.16.1.3”}, “actions”:[{“
↪ type”:“OUTPUT”,“port”:4}]}

{“comment”: “DC: Loop 3-4-5-3”}
{“dpid”:3, “cookie”:“0x200”, “priority”:2, “match”:{“eth_type”:2048, “ipv4_src”:“172.16.1.4”, “ipv4_dst
↪ ”:“172.16.1.5”}, “actions”:[{“type”:“OUTPUT”,“port”:4}]}
{“dpid”:4, “cookie”:“0x201”, “priority”:2, “match”:{“in_port”:4, “eth_type”:2048, “ipv4_src”:“172.16.1.4”, “
↪ ipv4_dst”:“172.16.1.5”}, “actions”:[{“type”:“OUTPUT”,“port”:5}]}
{“dpid”:5, “cookie”:“0x201”, “priority”:2, “match”:{“in_port”:2, “eth_type”:2048, “ipv4_src”:“172.16.1.4”, “
↪ ipv4_dst”:“172.16.1.5”}, “actions”:[{“type”:“OUTPUT”,“port”:1}]}

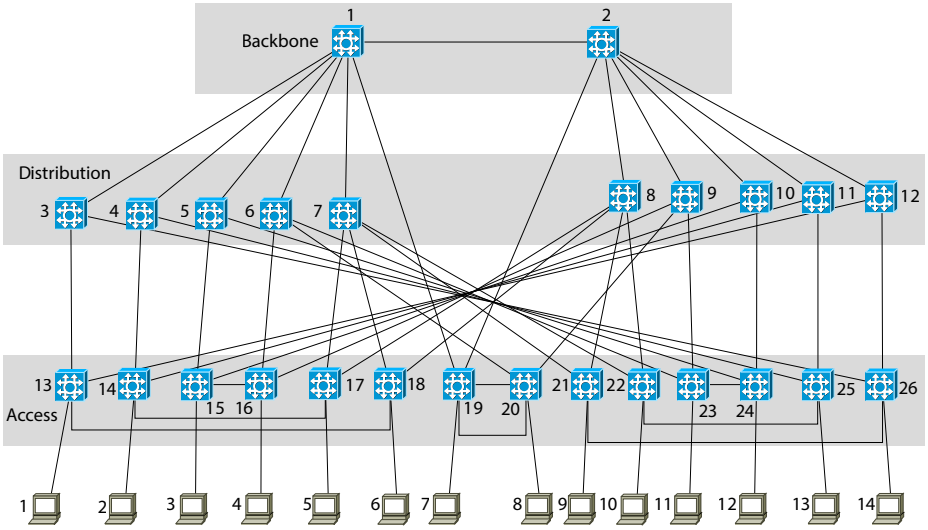


Figure 71: The simulated topology for the Stanford's backbone network [50]

The conflicts detected by the detector are compared with our design to conclude the soundness and completeness. This is possible thanks to the information logged by the detector including the rules, the rule graph and the identified conflicts (see Section 6.1.6). The results are shown in Tables 6.1 and 6.3, all conflicts are precisely identified.

The conflicts concerning *packet modification* are evaluated separately by Reyes [90]. Scenarios for conflicts are carefully designed in advance using the framework for automating experiments (see Section 6.1.5), and the conflicts identified by the detector are controlled manually. The evaluation results from his work are synthesized in Table 6.2. As explained in his work, the detector does not achieve the completeness quality in some tests (e. g., tests 1, 2, 3, 4 on the Stanford topology and test 5 on the MWN topology) because of the high latency in detecting conflicts of this type while each test is restricted within a fixed time frame, this consequence is foreseeable due

to the significant complexity pointed out in Chapter 5.8. Yet, the soundness of the detected conflicts is accomplished.

Test	Local conflicts					Traffic	Traffic	Hidden conflicts
	Shadowing	Generalization	Redundancy	Correlation	Overlap	Loop	Drop	ESLH
1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1
2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3
4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4
5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5

Table 6.1: Evaluation results in the designed cases on the MWN's backbone network topology. Each cell shows the number of conflicts detected by the prototype over the number of designed conflicts. ESLH stands for the hidden conflict class *Event Suppression by Local Handling*.

Test	MWN	Stanford
1	2/2/2	2/1/1
2	5/5/5	2/1/1
3	6/6/6	4/2/2
4	8/8/8	4/2/2
5	10/7/7	2/2/2

Table 6.2: Evaluation results concerning conflicts related to packet modification in the designed cases on the MWN's and Stanford's backbone network topologies . Each cell shows the number of designed conflicts, the number of conflicts detected by the prototype, and the number of valid cases. This table is synthesized from the evaluation results presented by Reyes [90] (Tables 7.3 and 7.4 in Chapter 7, page 92 of his thesis).

Test	Local conflicts					Traffic	Traffic	Hidden conflicts
	Shadowing	Generalization	Redundancy	Correlation	Overlap	Loop	Drop	ESLH
1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1
2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3
4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4
5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5

Table 6.3: Evaluation results in the designed cases on the Stanford's backbone network topology. Each cell shows the number of conflicts detected by the prototype over the number of designed conflicts. ESLH stands for the hidden conflict class *Event Suppression by Local Handling*.

6.2.3 Evaluation results in randomly checked cases

We use the framework for automating experiments with the integrated conflict detector to run the tests automatically (cf. Section 6.1.5). Different control applications (described in Chapter 3.5.2) with various settings are deployed on the simulated MWN and Stanford networks. The number of conflicts that will occur in each test is unknown before the test is finished. We select some samples of conflicts from those reported by the detector to assess its correctness.

Dimensions	Test space for MWN test-bed	Test space for Stanford test-bed
App config.	Each app has 1 config.	Each app has 1 config.
App start order	Same	Same
App priority	All combinations	All combinations
Target switches	EpLB:1, PPLB4S:2, HS:5, PE:10	EpLB: 15 16, PPLB4S: 5 6, PPLB4D: 1 2
Ept traf.prof.	CBR	CBR
Ept combi. (src-->dst)	{3 4 7 8 13 14 15 16 19 21} -->{1 2 5 6 17 18}	{9 10 11 12 13 14} -->{1 2 3 4 5 6 7 8}
Topology	MWN	Stanford
Transport type	TCP/UDP	TCP/UDP

Table 6.4: Evaluation settings for the randomly checked cases

Control applications

The fundamental functions including *topology discovery* and *ARP cache* are involved in all experiments, so is the *shortest path first routing* operating as the background application for the others, i. e., its rules are active at a network device only when other control applications do not effect there. We deploy the control applications End-point Load balancer (EpLB), Source-based Passive Path Load Balancer (PPLB4S), passive Host Shadowing (pHS) and Path Enforcer (PE) on the MWN topology, the Stanford topology is tested with EpLB, PPLB4S and PPLB4D (Destination-based Passive Path Load Balancer). Each application has one configuration in the test.

- In the MWN test-bed, EpLB places rules on switch 1 to balance TCP/UDP traffic destined to end-point 2 over end-points 1 and 2, PPLB4S deploys rules on switch 2 to direct TCP/UDP traffic sent by end-point 3 on all possible paths to the destination, pHS is active on switch 5 to redirect TCP/UDP traffic to end-point 6 if the destination of this traffic is end-point 5, PE enforces the path through switches 10 and 8 for TCP/UDP traffic flowing through switch 10.
- In the Stanford test-bed, EpLB installs rules on switches 15 and 16 to balance TCP/UDP traffic destined to end-point 3 over end-points 3 and 4, PPLB4S is active on switches 5 and 6 to direct TCP/UDP traffic sent by end-points 3 and 4 on all possible paths to the destination, PPLB4D monitors events on switches 1 and 2 and issues rules forwarding TCP/UDP traffic destined to end-points 7 and 8 on all possible paths.

Tested dimensions

Table 6.4 describes the tested dimensions. Each control application has one configuration, they all start at the same time in each experiment, all combinations of the priority in their co-deployment are tested. We use *netcat* and *iperf* to generate TCP and UDP traffic in the CBR traffic profile between all combinations of end-points shown in this table. The test space for the MWN test-bed contains 145 individual experiments, the space for the Stanford test-bed has 22 experiments.

Results

It took more than 62 hours to complete 145 individual tests associated with the dimensions specified for the MWN topology, and more than 9 hours for 22 tests on the Stanford topology. We choose randomly 5 tests on each topology to analyze the results. The number of conflicts detected in most of the tests is large, for example, a test on the MWN topology reports 27 *Correlation* local conflicts and 60 hidden conflicts of the class *Event Suppression by Local Handling*, the total number of rules stored in the rule database of the detector is 790 (see Test 1 in Table 6.5). It would be excessively time-consuming to verify all reported conflicts as this process has to be performed manually. Therefore, we choose to control for each conflict class at most 10 samples. The results are logged in Tables 6.5 and 6.6. For all of these randomly selected samples, we are able to confirm that the conflicts identified by the detector are precise.

Test	App	#	Local conflicts					Traffic	Traffic	HC
	Priority	rules	Sha	Gen	Red	Cor	Ove	Loop	Drop	ESLH
1	(2,2,2,2)	790	0/0/0	0/0/0	0/0/0	27/10/10	0/0/0	0/0/0	0/0/0	60/10/10
2	(2,2,3,4)	803	0/0/0	0/0/0	0/0/0	26/10/10	0/0/0	0/0/0	0/0/0	60/10/10
3	(3,2,2,3)	816	0/0/0	0/0/0	0/0/0	27/10/10	0/0/0	0/0/0	0/0/0	60/10/10
4	(3,5,2,4)	789	0/0/0	0/0/0	0/0/0	25/10/10	0/0/0	0/0/0	0/0/0	59/10/10
5	(5,4,3,2)	791	0/0/0	0/0/0	0/0/0	24/10/10	0/0/0	0/0/0	0/0/0	60/10/10

Table 6.5: Evaluation results in the randomly checked cases on the MWN's backbone network topology. The *App Priority* column shows the priority combination of the control applications (EpLB, PPLB4S, HS, PE) in the test, the *# rules* column notes the number of rules stored in the rule database of the detector. Each cell of the columns for conflicts shows the number of conflicts detected by the prototype, the number of conflicts that we select randomly to control, and the number of conflicts that are identified correctly by the prototype in the selected set confirmed by our manual control. Sha stands for *Shadowing*, Gen: *Generalization*, Red: *Redundancy*, Cor: *Correlation*, Ove: *Overlap*, and HC ESLH stands for the hidden conflict class *Event Suppression by Local Handling*.

Test	App	#	Local conflicts					Traffic	Traffic	HC
	Priority	rules	Sha	Gen	Red	Cor	Ove	Loop	Drop	ESLH
1	(2,2,2)	650	0/0/0	0/0/0	0/0/0	4/4/4	0/0/0	0/0/0	0/0/0	34/10/10
2	(2,3,4)	672	0/0/0	0/0/0	0/0/0	5/5/5	0/0/0	0/0/0	0/0/0	34/10/10
3	(3,2,2)	670	0/0/0	0/0/0	0/0/0	5/5/5	0/0/0	0/0/0	0/0/0	35/10/10
4	(3,4,2)	662	0/0/0	0/0/0	0/0/0	5/5/5	0/0/0	0/0/0	0/0/0	35/10/10
5	(4,3,2)	659	0/0/0	0/0/0	0/0/0	5/5/5	0/0/0	0/0/0	0/0/0	34/10/10

Table 6.6: Evaluation results in the randomly checked cases on the Stanford's backbone network topology. The *App Priority* column shows the priority combination of the control applications (EpLB, PPLB4S, PPLB4D) in the test, the *# rules* column notes the number of rules stored in the rule database of the detector. Each cell of the columns for conflicts shows the number of conflicts detected by the prototype, the number of conflicts that we select randomly to control, and the number of conflicts that are identified correctly by the prototype in the selected set confirmed by our manual control. Sha stands for *Shadowing*, Gen: *Generalization*, Red: *Redundancy*, Cor: *Correlation*, Ove: *Overlap*, and HC ESLH stands for the hidden conflict class *Event Suppression by Local Handling*.

6.3 Discussion

We employ the conflict detection prototype as a means to evaluate the concepts and results established in this research, comprising i) the conflict classes and their patterns or properties, and ii) the methods to detect conflicts based on the *multi-property set* notion, the relationship combination operator $\cdot r$, the *matchmap* and *actmap* concepts, and the *rule graph*. Although their correctness is justified by the soundness and completeness of the prototype in detecting conflicts, the performance aspect has not been optimised. We are able to observe some performance indicators reflecting the expensiveness in terms of detection time and memory usage when executing the detector. This is conceivable as the detector needs to maintain the considerable data about rules in the rule database and the rule graph. The same problem has been raised and addressed to some extent in the related work that aims at the real-time quality, such as VeriFlow [52] and NetPlumber [51]. Their strategies include using efficient data structure, such as *trie* [116], for searching rules and parallelizing the process of reasoning about anomalies by partitioning the rule graph in multiple subgraphs. These can be applied to improve our detector's performance. As pointed out in Section 6.1.4, the detection of distributed conflicts on each rule deployment can yield misleading output if a sequence of rules are still being deployed in reaction to a new traffic flow entering the network, while the real-time detection cannot be achieved if the detector waits and reacts only after that whole rule sequence were installed. No existing solution can tackle this issue, to the best of our knowledge. A probable intriguing topic for future study could be to predict the rule sequence in advance by some machine learning technique, the soundness of the detection results would then be contingent on the precision of the rule sequence prediction.

In principle, it is not necessary that the rule database contains all rules in the data plane, rather only those governing the traffic under consideration. For instance, we may filter away all rules handling ICMP, ARP traffic and insert to the rule database only those influencing TCP/UDP traffic. This is an important factor that helps improve the detector's performance as the size of the rule database and the rule graph is reduced accordingly, thus accelerating the searching time. Another notable factor relates to the choice of the *matchmap* and *actmap templates*. Each new rule is mapped to these templates to obtain its *matchmap* and *actmap*, before being stored in the rule database. The comparison of rules is conducted based on these two "maps". The more "compact" templates lessen the time required for each comparison of a rule pair, which is the crucial unit task for most time-consuming operations of the detector, such as updating the rule database and the rule graph on the presence of new rules to detect conflicts. The type of traffic being examined decides the template fields, e. g., the *matchmap template* in case it is only required to check conflicts for TCP traffic can be:

- ***matchmap template 1**: ingress port, EtherType, src IPv4, dst IPv4, IP protocol, src TCP port, dst TCP port* or
- ***matchmap template 2**: ingress port, src IPv4, dst IPv4, src TCP port, dst TCP port*

Apparently, the latter template is more "compact" and sufficient for comparing rules if the rule database contains only those regulating TCP traffic.

The conflict detector can be extended for checking invariants or policy compliance, which are among the common concerns in operating a network and are often addressed based on a graph of rules, e. g., *forwarding graph* in Veriflow [52] and AP-Keep [119], *propagation graph* in HSA [50], *plumbing graph* in Netplumber [51]. Our *rule graph* is equipped with a richer set of features than these graphs, thus can well be suitable for these goals.

7 Conclusions and Prospects

The SDN paradigm enables an ecosystem in which control applications are loosely coupled with the network infrastructure. On the one hand, this paradigm brings more independence in developing control applications and boosts their innovation cycle compared to the traditional network's. On the other hand, it induces a higher probability of conflicts when putting them together on the same infrastructure, due to the fact that they may have different goals to be enforced in the network. Our work aims at detecting conflicts within a given set of rules deployed in the network infrastructure by various control applications. The entailed sub-goals include the classification and localization of conflicts so that further efforts can be exerted, e. g., conflict orchestration, resolution or avoidance.

Our analysis of the formal analytical approach commonly employed in literature shows that it is not sufficient for exploring conflicts in SDN. We opt for the experimental approach in which we introduce a parameter space, a general methodology for conducting experiments through this space and for studying conflicts. We apply this methodology in various subspaces formed by concrete network topologies, control applications and end-point combinations. The immense experimental space urges the quest of a method to reduce manual attempts in performing experiments. For this purpose, we develop a framework that automates most steps in the methodology. Each subspace is encoded in a structured text file, the framework derives a set of settings from this encoding and conducts experiments, the expected and observed network behaviour are compared to determine settings with anomalies, the results are logged in a dataset. Eventually, more than 11,700 experiments have been conducted by this framework. It also supports the reproduction of an individual experiment from the dataset for examining conflicts. The anomaly cases reported in the dataset are analysed manually in the beginning, conflict classes are formulated and featured by their patterns or properties. In the next step, we employ the continuous integration process in building the conflict detection prototype and re-integrating it in the massive experimentation by the framework. Each anomaly logged in the dataset, unrecognizable by the prototype, is manually analysed. If a new conflict class is found, it is built into the prototype, which is then employed in subsequent experiments. This continuous integration approach not only enhances the conflict detection prototype throughout the experimentation process, but also helps reduce the time and effort in manually analysing anomaly cases, thus enabling the exploration of more subspaces to study conflicts.

The experimental approach yields fruitful outcomes: we have established a conflict taxonomy encompassing local conflicts, distributed conflicts and hidden conflicts. Local conflicts occurs between rules in a single rule table. They are categorised into various classes, being *shadowing*, *generalization*, *redundancy*, *correlation* and *overlap*. Distributed conflicts are featured by the unexpected combined effect of rules in dif-

ferent rule tables or different network devices. They are aligned in seven classes by the causes (*traffic looping, traffic dropping, packet modification, changes to paths*) and the direction of the incidents (*downstream* or *upstream*, the direction does not apply for the *changes to path* cause). Remarkably, we have discovered a completely new anomaly type, namely hidden conflict, which emanates from the side-effect of rule deployment and requires the insight of the control plane for their detection. Hidden conflicts are classified into seven classes: *event suppression by local handling, event suppression by upstream traffic looping, event suppression by upstream traffic dropping, event suppression by changes to paths, action suppression by packet modification, undue trigger* and *tampering with event subscription*.

Each conflict class is distinguishable by its unique pattern or property. This is exploited in conceiving the algorithms and developing the conflict detection prototype in the aforementioned continuous integration process. In order to compare SDN rules, which is crucial in detecting conflicts, we propose a new concept, named *multi-property set*, and the relationship combination operator $\cdot r$. These facilitate the rule comparison without the restrictions or assumptions incurred by existing solutions, such as, a rule match must include layer 4's information. More valuable trait of these tools is that they are general and thus applicable for any kinds of multi-property sets. For example, they can be employed for comparing the conditions (or match criteria) of ACL or *IPTables* rules, or to derive the relationship between two given set of flowers based on various attributes, e. g., color, number of petals, scent. To leverage the *multi-property set* and the $\cdot r$ operator for automatic rule comparison, we introduce additionally the *matchmap* and *actmap* concepts, which allows heterogeneous SDN rules to be represented in a uniform format. The pattern-based identification of local conflicts can then be realized in a simple manner. Conflicts related to rules in different rule tables are handled by the *rule graph*, being a directed graph reflecting how traffic flows in the data plane. We examine hidden conflicts first by a speculative prediction method, in which control applications are considered as black boxes. We observe, however, critical issues concerning the large number of fake events, the state poisoning of stateful control applications and the interference between fake and genuine events. Consequently, we adopt a more "relaxing" approach which requires extra information from control applications for their detection, e. g., target switches, interested traffic.

The excessive complexity and the interpretative nature in identifying conflicts of some classes hamper the efficient realization of their detection. Our discussion on their practical aspects suggests alternatively reasonable measures, which are implemented in our conflict detection prototype. The evaluation of the prototype demonstrates the realisability and correctness of our proposed concepts and algorithms.

Our work provides a firm foundation for farther efforts in handling conflicts in SDN. Conceivable areas that could directly profit by our achievements include conflict orchestration, resolution and avoidance. The more comprehensive the detection's outcome is, the more benefits these areas gain.

Future Work

We can envision the following challenges to be explored in conjunction with our work.

SDN technologies

We employ OpenFlow SDN for experimenting conflicts, the outcome favours thus this technology. Newer prominent SDN technologies emerge during the time of this writing related to P4 [80], P4Runtime [81] and POF [61]. Unlike OpenFlow devices, P4 devices can be configured with diverse capabilities. A device can be equipped with richer features than the others, e. g., some are capable of IPv6 extension headers, some are not. P4-based SDN is enabled by some APIs such as P4Runtime, through which the communication between the data plane and the logically centralized controller is performed. The controller as well as the control applications need to be aware of the data plane devices' diversity while issuing rules. Network functions cannot be deployed "arbitrarily" but only on capable devices (except when all devices are configured uniformly in terms of their capabilities). POF (protocol-oblivious forwarding) SDN allows the representation of match fields by {offset, length} tuples to provide the flexibility against the rigid specification of protocols as in OpenFlow SDN. P4, POF, OpenFlow and future SDN technologies, by themselves, create a new dimension in our parameter space (see Chapter 3.2.1) for researching conflicts.

Topology change

The change in the network topology, e. g., the introduction, removal or failure of a network device or a link, can lead to new conflicts or cause the existing rules conflicting, since the assumptions made by the control applications are possibly invalidated. One of the examples observed through our experiments is that the routing application kept forwarding traffic to a switch on the shortest path that at some point failed and became a black-hole in the network. The case indicates the need of applications in reacting to network dynamics to ensure their proper functions. Intuitively, a garbage collection mechanism should be carried out to remove or update the rules influenced by topology changes. In general, conflicts triggered by topology changes may be an interesting topic to be explored.

Matching policy

Different policies can be employed in selecting rules matching a packet that arrives at a network device, e. g., *best match, deny take precedence, most/least specific take precedence* [6]. Our work favors the first match policy to some extent, which is used by OpenFlow SDN and is also the most prevalent policy in rule-based packet processing. It is unclear how other matching policies may impact the occurrence of conflicts, and if a certain policy can reduce the propensity of conflicts than the

others in the same condition. The insights acquired from studying these aspects could greatly benefit the design, the implementation as well as the operation of SDN.

Real-time conflict detection

A rule can be checked against conflicts before being enforced in the data plane. Some existing study (e. g., [52, 87]) shows encouraging results with about 15.5% overhead added in the rule deployment time by employing efficient techniques and data structures like *trie* [116]. These are promising to achieve real-time detection of local conflicts and hidden conflicts of the first class (*event suppression by local handling*). The control of each single rule for distributed conflicts is excessively expensive as discussed in Chapter 6.1.4. However, this can be alleviated if we could predict the rules to be deployed, which simulates the complete set of rules handling the packet flow in concern, e. g., by using certain machine learning techniques. It is appealing to investigate the possibility of real-time conflict detection by applying these techniques and data structures.

Further conflict handling efforts

The conceivable undertaking after detecting conflicts encompasses their resolution and avoidance.

Conflict resolution
Different strategies for resolving conflicts can be taken into account, e. g., favoring rules of some application by raising its priority in case of conflicts, manually inserting rules to fulfill the intended goal, or disabling a certain application. The resolution strategies appear to be context-specific and require more investigation to identify the appropriate choice in each situation.

Conflict avoidance
A plausible scenario for conflict avoidance is to design a suite of applications that could work together in harmony. One can employ the dry-run approach with varied settings on this application suite to observe conflicts by using a conflict detector like ours. He can then progressively modify the applications or choose configurations that yield conflict-free outcomes. This approach, to some extent, realizes one of the conflict resolution strategies. The dimensions and parameters for customizing the applications to avoid conflicts are among the engaging research avenues.

Bibliography

[1] Adolfo Arteta, Benjamín Barán, and Diego Pinto. "Routing and Wavelength Assignment over WDM Optical Networks: A Comparison between MOACOs and Classical Approaches". In: *Proceedings of the 4th International IFIP/ACM Latin American Conference on Networking*. LANC '07. San José, Costa Rica: Association for Computing Machinery, 2007, pp. 53–63. ISBN: 9781595939074. DOI: 10.1145/1384117.1384126.

[2] Alvin AuYoung et al. "Democratic Resolution of Resource Conflicts Between SDN Control Programs". In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. CoNEXT '14. Sydney, Australia: Association for Computing Machinery, 2014, pp. 391–402. ISBN: 9781450332798. DOI: 10.1145/2674005.2674992.

[3] Arosha K. Bandara, Emil C. Lupu, and Alessandra Russo. "Using Event Calculus to Formalise Policy Specification and Analysis". In: *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*. POLICY '03. USA: IEEE Computer Society, 2003, p. 26. ISBN: 0769519334. DOI: 10.1109/POLICY.2003.1206955.

[4] Albert-László Barabási and Réka Albert. "Emergence of Scaling in Random Networks". In: *Science* 286.5439 (1999), pp. 509–512. DOI: 10.1126/science.286.5439.509.

[5] Paul Barham et al. "Xen and the Art of Virtualization". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: Association for Computing Machinery, 2003, pp. 164–177. ISBN: 1581137575. DOI: 10.1145/945445.945462.

[6] Cataldo Basile, Alberto Cappadonia, and Antonio Lioy. "Algebraic Models to Detect and Solve Policy Conflicts". In: *Computer Network Security*. Ed. by Vladimir Gorodetsky, Igor Kotenko, and Victor A. Skormin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 242–247. ISBN: 978-3-540-73986-9. DOI: 10.1007/978-3-540-73986-9_20.

[7] Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura. "An Architecture for Active Networking". In: *High Performance Networking VII: IFIP TC6 Seventh International Conference on High Performance Networks (HPN '97), 28th April – 2nd May 1997, White Plains, New York, USA*. Ed. by Ahmed Tantawy. Boston, MA: Springer US, 1997, pp. 265–279. ISBN: 978-0-387-35279-4. DOI: 10.1007/978-0-387-35279-4_17.

[8] M. Bjorklund. *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*. RFC 6020. IETF, Oct. 2010. URL: http://tools.ietf.org/rfc/rfc6020.txt.

[9] S. Blake et al. *An Architecture for Differentiated Services*. RFC 2475. IETF, Dec. 1998. URL: http://tools.ietf.org/rfc/rfc2475.txt.

[10] Soren Bleikertz and Thomas Groß. "A Virtualization Assurance Language for Isolation and Deployment". In: *2011 IEEE International Symposium on Policies for Distributed Systems and Networks*. 2011, pp. 33–40. DOI: 10.1109/POLICY.2011.10.

[11] U. Blumenthal and B. Wijnen. *User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)*. RFC 3414. IETF, Dec. 2002. URL: http://tools.ietf.org/rfc/rfc3414.txt.

[12] Pat Bosshart et al. "P4: Programming Protocol-Independent Packet Processors". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890.

[13] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. IETF, Dec. 2017. URL: http://tools.ietf.org/rfc/rfc8259.txt.

[14] Matthew Caesar et al. "Design and Implementation of a Routing Control Platform". In: *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*. NSDI'05. USA: USENIX Association, 2005, pp. 15–28.

[15] Marco Canini et al. "A NICE Way to Test Openflow Applications". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, p. 10.

[16] J.D. Case et al. *Simple Network Management Protocol (SNMP)*. RFC 1157. IETF, May 1990. URL: http://tools.ietf.org/rfc/rfc1157.txt.

[17] Jan Chomicki, Jorge Lobo, and Shamim Naqvi. "A logic programming approach to conflict resolution in policy management". In: *7th International Conference on Principles of Knowledge Representation and Reasoning (KR'2000)*. Citeseer. 2000.

[18] Jan Chomicki, Jorge Lobo, and Shamim Naqvi. "Conflict Resolution Using Logic Programming". In: *IEEE Trans. on Knowl. and Data Eng.* 15.1 (Jan. 2003), pp. 244–249. ISSN: 1041-4347. DOI: 10.1109/TKDE.2003.1161596.

[19] Paul Congdon. *Link Layer Discovery Protocol and MIB v0.0*. IEEE standards association. 2002. URL: https://www.ieee802.org/1/files/public/docs2002/lldp-protocol-00.pdf (visited on 03/13/2022).

[20] Jie Cui et al. "Transaction-Based Flow Rule Conflict Detection and Resolution in SDN". In: *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. IEEE. 2018, pp. 1–9. DOI: 10.1109/ICCCN.2018.8487415.

[21] Nicodemos Damianou et al. "The Ponder Policy Specification Language". In: *Policies for Distributed Systems and Networks*. Ed. by Morris Sloman, Emil C. Lupu, and Jorge Lobo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 18–38. ISBN: 978-3-540-44569-2. DOI: 10.1007/3-540-44569-2_2.

[22] Vitalian Danciu. "Application of Policy-based Techniques to Process-oriented IT Service Management". PhD thesis. LMU, 2007. ISBN: 383707160X. DOI: 10.5282/edoc.7816.

[23] Vitalian Danciu, Tobias Guggemos, and Dieter Kranzlmüller. "Schichtung virtueller Maschinen zu Labor- und Lehrinfrastruktur". In: *9. DFN-Forum Kommunikations-technologien*. Bonn: Gesellschaft für Informatik e.V., 2016, pp. 35–44.

[24] Vitalian Danciu and Cuong Ngoc Tran. "MEADcast: Explicit Multicast with Privacy Aspects". In: *International Journal on Advances in Security* 12.1&2 (Aug. 2019), pp. 13–28. ISSN: 1942-2636. URL: https://www.iariajournals.org/security/tocv12n12.html.

[25] Vitalian Danciu and Cuong Ngoc Tran. "Side-Effects Causing Hidden Conflicts in Software-Defined Networks". In: *SN Computer Science* 1.1 (Aug. 2020), p. 278. ISSN: 2661-8907. DOI: 10.1007/s42979-020-00282-0.

[26] C. Diekmann et al. "Verified iptables Firewall Analysis". In: *2016 IFIP Networking Conference (IFIP Networking) and Workshops*. 2016, pp. 252–260. DOI: 10.1109/IFIPNetworking.2016.7497196.

[27] D. Durham et al. *The COPS (Common Open Policy Service) Protocol*. RFC 2748. IETF, Jan. 2000. URL: http://tools.ietf.org/rfc/rfc2748.txt.

[28] R. Enns et al. *Network Configuration Protocol (NETCONF)*. RFC 6241. IETF, June 2011. URL: http://tools.ietf.org/rfc/rfc6241.txt.

[29] *eXtensible Access Control Markup Language (XACML) version 3.0*. OASIS Standard. OASIS. Jan. 22, 2013. URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf (visited on 03/13/2022).

[30] A. Farrel, J.-P. Vasseur, and J. Ash. *A Path Computation Element (PCE)-Based Architecture*. RFC 4655. IETF, Aug. 2006. URL: http://tools.ietf.org/rfc/rfc4655.txt.

[31] Nick Feamster, Jennifer Rexford, and Ellen Zegura. "The Road to SDN: An Intellectual History of Programmable Networks". In: *SIGCOMM Comput. Commun. Rev.* 44.2 (Apr. 2014), pp. 87–98. ISSN: 0146-4833. DOI: 10.1145/2602204.2602219.

[32] Andrew D. Ferguson et al. "Hierarchical Policies for Software Defined Networks". In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. HotSDN '12. Helsinki, Finland: Association for Computing Machinery, 2012, pp. 37–42. ISBN: 9781450314770. DOI: 10.1145/2342441.2342450.

[33] S. Ferraresi et al. "Automatic Conflict Analysis and Resolution of Traffic Filtering Policy for Firewall and Security Gateway". In: *2007 IEEE International Conference on Communications*. 2007, pp. 1304–1310. DOI: 10.1109/ICC.2007.220.

[34] I. Fette and A. Melnikov. *The WebSocket Protocol*. RFC 6455. IETF, Dec. 2011. URL: http://tools.ietf.org/rfc/rfc6455.txt.

[35] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. IETF, June 1999. URL: http://tools.ietf.org/rfc/rfc2616.txt.

[36] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. University of California, Irvine, 2000.

[37] Nate Foster et al. "Frenetic: A Network Programming Language". In: *SIGPLAN Not.* 46.9 (Sept. 2011), pp. 279–291. ISSN: 0362-1340. DOI: 10.1145/2034574.2034812.

[38] *Framework of Software-Defined Networking*. ITU Recommendation Y.3300, June 2014. ITU. URL: http://www.itu.int/rec/T-REC-Y.3300-201406-I/en (visited on 03/13/2022).

[39] N. Freed. *Behavior of and Requirements for Internet Firewalls*. RFC 2979. IETF, Oct. 2000. URL: http://tools.ietf.org/rfc/rfc2979.txt.

[40] Paul Goransson and Chuck Black. *Software Defined Networks: A Comprehensive Approach*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014. ISBN: 012416675X.

[41] Natasha Gude et al. "NOX: Towards an Operating System for Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.3 (July 2008), pp. 105–110. ISSN: 0146-4833. DOI: 10.1145/1384609.1384625.

[42] E. Haleplidis et al. *Software-Defined Networking (SDN): Layers and Architecture Terminology*. RFC 7426. IETF, Jan. 2015. URL: http://tools.ietf.org/rfc/rfc7426.txt.

[43] Hazem Hamed and Ehab Al-Shaer. "Taxonomy of Conflicts in Network Security Policies". In: *IEEE Communications Magazine* 44.3 (Mar. 2006), pp. 134–141. ISSN: 1558-1896. DOI: 10.1109/MCOM.2006.1607877.

[44] Nikhil Handigol et al. "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 71–85. ISBN: 9781931971096.

[45] Nikhil Handigol et al. "Where is the Debugger for My Software-Defined Network?" In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. HotSDN '12. Helsinki, Finland: Association for Computing Machinery, 2012, pp. 55–60. ISBN: 9781450314770. DOI: 10.1145/2342441.2342453.

[46] *Intent NBI – Definition and Principles*. Tech. rep. Open Networking Foundation, Oct. 2016. URL: https://opennetworking.org/wp-content/uploads/2014/10/TR-523_Intent_Definition_Principles.pdf (visited on 03/13/2022).

[47] *Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model*. Standard. International Organization for Standardization, Nov. 1994.

[48] Yosr Jarraya, Taous Madi, and Mourad Debbabi. "A Survey and a Layered Taxonomy of Software-Defined Networking". In: *IEEE Communications Surveys & Tutorials* 16.4 (2014), pp. 1955–1980. DOI: 10.1109/COMST.2014.2320094.

[49] Lalana Kagal. *Rei: A Policy Language for the Me-Centric Project*. Tech. rep. HP Labs, Palo Alto, Sept. 26, 2002. URL: https://ebiquity.umbc.edu/_file_directory_/papers/57.pdf (visited on 03/13/2022).

[50] Peyman Kazemian, George Varghese, and Nick McKeown. "Header Space Analysis: Static Checking for Networks". In: *Proceedings of the 9th USENIX*

Conference on Networked Systems Design and Implementation. NSDI'12. San Jose, CA: USENIX Association, 2012, p. 9.

[51] Peyman Kazemian et al. "Real Time Network Policy Checking Using Header Space Analysis". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. NSDI'13. Lombard, IL: USENIX Association, 2013, pp. 99–112.

[52] Ahmed Khurshid et al. "VeriFlow: Verifying Network-Wide Invariants in Real Time". In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. HotSDN '12. Helsinki, Finland: Association for Computing Machinery, 2012, pp. 49–54. ISBN: 9781450314770. DOI: 10.1145/2342441.2342452.

[53] Avi Kivity et al. "KVM: The Linux Virtual Machine Monitor". In: *Proceedings of the Linux symposium*. Vol. 1. 8. Dttawa, Dntorio, Canada. 2007, pp. 225–230.

[54] Rosalie Kletzander. *A testbed for researching conflicts in SDN*. Bachelor's thesis. Ludwig-Maximilians-Universität München, 2017. URL: https://www.mnm-team.org/pub/Fopras/klet17/ (visited on 03/13/2022).

[55] Thomas Koch. *Automated management of distributed systems*. Shaker, 1997. ISBN: 3-8265-2594-9.

[56] Robert Kowalski and Marek Sergot. "A Logic-Based Calculus of Events". In: *Foundations of Knowledge Base Management: Contributions from Logic, Databases, and Artificial Intelligence Applications*. Ed. by Joachim W. Schmidt and Constantino Thanos. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 23–55. ISBN: 978-3-642-83397-7. DOI: 10.1007/978-3-642-83397-7_2.

[57] Diego Kreutz et al. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76. DOI: 10.1109/JPROC.2014.2371999.

[58] TV Lakshman et al. "The Softrouter Architecture". In: *Proc. ACM SIGCOMM Workshop on Hot Topics in Networking*. Vol. 2004. ACM, 2004.

[59] Bob Lantz, Brandon Heller, and Nick McKeown. "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. Monterey, California: Association for Computing Machinery, 2010. ISBN: 9781450304092. DOI: 10.1145/1868447.1868466.

[60] Jun Li et al. "A Software-Defined Address Resolution Proxy". In: *Proceedings of the 2017 IEEE Symposium on Computers and Communications (ISCC)*. July 2017, pp. 404–410. DOI: 10.1109/ISCC.2017.8024563.

[61] S. Li et al. "Protocol Oblivious Forwarding (POF): Software-Defined Networking with Enhanced Programmability". In: *IEEE Network* 31.2 (2017), pp. 58–66. DOI: 10.1109/MNET.2017.1600030NM.

[62] Jorge Lobo, Randeep Bhatia, and Shamim Naqvi. "A Policy Description Language". In: *AAAI/IAAI* 1999 (1999), pp. 291–298.

[63] Claas Lorenz et al. "Continuous Verification of Network Security Compliance". In: *IEEE Transactions on Network and Service Management* 19.2 (2022), pp. 1729–1745. DOI: 10.1109/TNSM.2021.3130290.

[64] S. Loreto et al. *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP*. RFC 6202. IETF, Apr. 2011. URL: http://tools.ietf.org/rfc/rfc6202.txt.

[65] E. Lupu and M. Sloman. "Conflict Analysis for Management Policies". In: *Integrated Network Management V: Integrated management in a virtual world Proceedings of the Fifth IFIP/IEEE International Symposium on Integrated Network Management San Diego, California, U.S.A., May 12–16, 1997*. Ed. by Aurel A. Lazar, Roberto Saracco, and Rolf Stadler. Boston, MA: Springer US, 1997, pp. 430–443. ISBN: 978-0-387-35180-3. DOI: 10.1007/978-0-387-35180-3_32.

[66] Emil C Lupu and Morris Sloman. "Conflicts in Policy-based Distributed Systems Management". In: *IEEE Transactions on Software Engineering* 25.6 (Nov. 1999), pp. 852–869. ISSN: 1939-3520. DOI: 10.1109/32.824414.

[67] *Management Framework for Open Systems Interconnection (OSI) for CCITT Applications*. ITU Recommendation X.700, September 1992, ITU. URL: https://www.itu.int/rec/T-REC-X.700-199209-I/en (visited on 03/13/2022).

[68] K. McCloghrie and M. Rose. *Management Information Base for Network Management of TCP/IP-based internets: MIB-II*. RFC 1213. IETF, Mar. 1991. URL: http://tools.ietf.org/rfc/rfc1213.txt.

[69] Nick McKeown. "Software-Defined Networks and The Maturing of The Internet". IET Appleton Lecture. Apr. 2014.

[70] Nick McKeown et al. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746.

[71] Jonathan D Moffett and Morris S Sloman. "Policy Conflict Analysis in Distributed System Management". In: *Journal of Organizational Computing and Electronic Commerce* 4.1 (1994), pp. 1–22.

[72] Jeffrey C. Mogul et al. "Corybantic: Towards the Modular Composition of SDN Control Programs". In: *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. HotNets-XII. College Park, Maryland: Association for Computing Machinery, 2013. ISBN: 9781450325967. DOI: 10.1145/2535771.2535795.

[73] B. Moore. *Policy Core Information Model (PCIM) Extensions*. RFC 3460. IETF, Jan. 2003. URL: http://tools.ietf.org/rfc/rfc3460.txt.

[74] Thomas D Nadeau and Ken Gray. *SDN: Software Defined Networks: An Authoritative Review of Network Programmability Technologies*. O'Reilly Media, Inc., 2013. ISBN: 9781449342302.

[75] T. Narten et al. *Neighbor Discovery for IP version 6 (IPv6)*. RFC 4861. IETF, Sept. 2007. URL: http://tools.ietf.org/rfc/rfc4861.txt.

[76] Duc Minh Nguyen. *Deployment of MEADcast in Stub Software-Defined Networks*. Bachelor's thesis. Ludwig-Maximilians-Universität München, 2019. URL: https://www.mnm-team.org/pub/Fopras/nguy19/.

[77] *OpenFlow Management and Configuration Protocol (OF-Config 1.1. 1)*. Version 1.1.1, ONF TS-008. Open Networking Foundation. Mar. 23, 2013. URL: https://opennetworking.org/wp-content/uploads/2013/02/of-config-1-1-1.pdf (visited on 03/13/2022).

[78] *OpenFlow Switch Specification*. Version 1.5.1. Open Networking Foundation. 2015. URL: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf (visited on 03/13/2022).

[79] *OpenFlow Switch Specification*. Version 1.3.5. Open Networking Foundation. 2015. URL: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.3.5.pdf (visited on 03/13/2022).

[80] *$P4_{16}$ Language Specification*. Version 1.2.1. The P4 Language Consortium. June 2020. URL: https://p4.org/p4-spec/docs/P4-16-v1.2.1.pdf (visited on 03/13/2022).

[81] *P4Runtime Specification*. Version 1.2.0. The P4.org API Working Group. July 2020. URL: https://opennetworking.org/wp-content/uploads/2020/10/P4Runtime-Spec.pdf (visited on 03/13/2022).

[82] Gregorio Martínez Pérez, Félix J García Clemente, and Antonio F Gómez Skarmeta. "Policy-based Management of Web and Information Systems Security: An Emerging Technology". In: *Information Security and Ethics: Concepts, Methodologies, Tools, and Applications*. IGI Global, 2008, pp. 2991–3005. DOI: 10.4018/978-1-59904-937-3.ch200.

[83] B. Pfaff and B. Davie. *The Open vSwitch Database Management Protocol*. RFC 7047. IETF, Dec. 2013. URL: http://tools.ietf.org/rfc/rfc7047.txt.

[84] Ben Pfaff et al. "The Design and Implementation of Open vSwitch". In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. Oakland, CA: USENIX Association, 2015, pp. 117–130. ISBN: 9781931971218.

[85] Charles C Pinter. *A book of set theory*. Courier Corporation, 2014. ISBN: 978-0486497082.

[86] Sandeep Pisharody. "Policy Conflict Management in Distributed SDN Environments". PhD thesis. Arizona State University, 2017.

[87] Sandeep Pisharody et al. "Brew: A Security Policy Analysis Framework for Distributed SDN-Based Cloud Environments". In: *IEEE Transactions on Dependable and Secure Computing* 16.6 (2019), pp. 1011–1025. DOI: 10.1109/TDSC.2017.2726066.

[88] Philip Porras et al. "A Security Enforcement Kernel for OpenFlow Networks". In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. HotSDN '12. Helsinki, Finland: Association for Computing Machinery, 2012, pp. 121–126. ISBN: 9781450314770. DOI: 10.1145/2342441.2342466.

[89] J. Postel and J. Reynolds. *File Transfer Protocol*. RFC 959. IETF, Oct. 1985. URL: http://tools.ietf.org/rfc/rfc0959.txt.

[90] Nicholas Reyes. "Detection of Distributed Conflicts in Software-Defined Networks". MA thesis. Ludwig-Maximilians-Universität München, 2021. URL: https://www.mnm-team.org/pub/Diplomarbeiten/reye21/.

[91] *SDN Architecture, Version 1*. Tech. rep. Open Networking Foundation, June 2014. URL: https://opennetworking.org/wp-content/uploads/2013/02/TR_SDN_ARCH_1.0_06062014.pdf (visited on 03/13/2022).

[92] Ehab Al-Shaer and Saeed Al-Haj. "FlowChecker: Configuration Analysis and Verification of Federated Openflow Infrastructures". In: *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration*. SafeConfig '10. Chicago, Illinois, USA: Association for Computing Machinery, 2010, pp. 37–44. ISBN: 9781450300933. DOI: 10.1145/1866898.1866905.

[93] Ehab Al-Shaer et al. "Conflict Classification and Analysis of Distributed Firewall Policies". In: *IEEE Journal on Selected Areas in Communications* 23.10 (2005), pp. 2069–2084. DOI: 10.1109/JSAC.2005.854119.

[94] Ehab S Al-Shaer and Hazem H Hamed. "Firewall Policy Advisor for anomaly discovery and rule editing". In: *Proceedings of the IFIP/IEEE Eighth International Symposium on Integrated Network Management, 2003.* Colorado Springs, CO, USA, Mar. 2003, pp. 17–30. DOI: 10.1109/INM.2003.1194157.

[95] Seungwon Shin et al. "FRESCO: Modular Composable Security Services for Software-Defined Networks". In: *Proceedings of the 20th Annual Network & Distributed System Security (NDSS) Symposium*. San Diego, CA United States, 2013.

[96] Jonathan M Smith et al. *SwitchWare: Accelerating Network Evolution (White paper)*. University of Pennsylvania. Jan. 1996. URL: https://repository.upenn.edu/cgi/viewcontent.cgi?article=1212&context=cis_reports (visited on 03/13/2022).

[97] *Software-defined networking: The new norm for networks*. ONF White Paper. Apr. 13, 2012. URL: http://opennetworking.wpengine.com/wp-content/uploads/2011/09/wp-sdn-newnorm.pdf (visited on 03/13/2022).

[98] Haoyu 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*. HotSDN '13. Hong Kong, China: Association for Computing Machinery, 2013, pp. 127–132. ISBN: 9781450321785. DOI: 10.1145/2491185.2491190.

[99] Michael Steinke, Iris Adam, and Wolfgang Hommel. "Multi-Tenancy-Capable Correlation of Security Events in 5G Networks". In: *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2018, pp. 1–6. DOI: 10.1109/NFV-SDN.2018.8725633.

[100] Michael Steinke and Wolfgang Hommel. "A Data Model for Federated Network and Security Management Information Exchange in Inter-

organizational IT Service Infrastructures". In: *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*. 2018, pp. 1–2. DOI: 10.1109/NOMS.2018.8406162.

[101] Michael Steinke and Wolfgang Hommel. "Overcoming Network and Security Management Platform Gaps in Federated Software Networks". In: *2018 14th International Conference on Network and Service Management (CNSM)*. 2018, pp. 295–299.

[102] J. Strassner et al. *Policy Core Lightweight Directory Access Protocol (LDAP) Schema*. RFC 3703. IETF, Feb. 2004. URL: http://tools.ietf.org/rfc/rfc3703.txt.

[103] John Strassner. *Policy-based Network Management: Solutions for The Next Generation*. Morgan Kaufmann, 2003. ISBN: 9781558608597.

[104] Peng Sun et al. "A Network-State Management Service". In: *SIGCOMM Comput. Commun. Rev.* 44.4 (Aug. 2014), pp. 563–574. ISSN: 0146-4833. DOI: 10.1145/2740070.2626298.

[105] David L. Tennenhouse and David J. Wetherall. "Towards an Active Network Architecture". In: *SIGCOMM Comput. Commun. Rev.* 37.5 (Oct. 2007), pp. 81–94. ISSN: 0146-4833. DOI: 10.1145/1290168.1290180.

[106] Emir Toktar, Edgard Jamhour, and E Maziero. "RSVP Policy Control using XACML". In: *Proceedings. Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004. POLICY 2004.* IEEE. 2004, pp. 87–96. DOI: 10.1109/POLICY.2004.1309153.

[107] FUJITA Tomonori. *Introduction to Ryu SDN Framework*. Presentation in 2013 Open Networking Summit. 2013. URL: https://ryu-sdn.org/slides/ONS2013-april-ryu-intro.pdf (visited on 03/13/2022).

[108] Cuong Ngoc Tran and Vitalian Danciu. "A General Approach to Conflict Detection in Software-Defined Networks". In: *SN Computer Science* 1.1 (July 2019), p. 9. ISSN: 2661-8907. DOI: 10.1007/s42979-019-0009-9.

[109] Cuong Ngoc Tran and Vitalian Danciu. "Hidden Conflicts in Software-Defined Networks". In: *Proceedings of the 2019 International Conference on Advanced Computing and Applications (ACOMP)*. IEEE. Nha Trang, Vietnam, 2019, pp. 127–134. DOI: 10.1109/ACOMP.2019.00027.

[110] Cuong Ngoc Tran and Vitalian Danciu. "On Conflict Handling in Software-Defined Networks". In: *Proceedings of the 2018 International Conference on Advanced Computing and Applications*. Ho Chi Minh City, Vietnam: CPS, 2018, pp. 50–57. DOI: 10.1109/ACOMP.2018.00016.

[111] Cuong Ngoc Tran and Vitalian Danciu. "Privacy-Preserving Multicast to explicit agnostic Destinations". In: *Proceedings of the Eighth International Conference on Advanced Communications and Computation (INFOCOMP 2018)*. Barcelona, Spain: IARIA XPS Press, 2018, pp. 60–65. ISBN: 978-1-61208-655-2.

[112] Andrzej Uszok, Jeffrey M. Bradshaw, and Renia Jeffers. "KAoS: A Policy and Domain Services Framework for Grid Computing and Semantic Web Services". In: *Trust Management*. Ed. by Christian Jensen, Stefan Poslad, and Theo

Dimitrakos. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 16–26. ISBN: 978-3-540-24747-0.

[113] Anduo Wang et al. "Ravel: A Database-Defined Network". In: *Proceedings of the Symposium on SDN Research*. SOSR '16. Santa Clara, CA, USA: Association for Computing Machinery, 2016. ISBN: 9781450342117. DOI: 10.1145/2890955.2890970.

[114] David J Wetherall, John V Guttag, and David L Tennenhouse. "ANTS: A Toolkit for Building and dynamically Deploying Network Protocols". In: *1998 IEEE Open Architectures and Network Programming*. IEEE. 1998, pp. 117–129. DOI: 10.1109/OPNARC.1998.662048.

[115] René Félix Jacques Wies. "Policies in Integrated Network and Systems Management: Methodologies for the Definition, Transformation, and Application of Management Policies". PhD thesis. LMU, 1995.

[116] Dan E. Willard. "New Trie Data Structures Which Support Very Fast Search Operations". In: *J. Comput. Syst. Sci.* 28.3 (July 1984), pp. 379–394. ISSN: 0022-0000. DOI: 10.1016/0022-0000(84)90020-5.

[117] L. Yang et al. *Forwarding and Control Element Separation (ForCES) Framework*. RFC 3746. IETF, Apr. 2004. URL: http://tools.ietf.org/rfc/rfc3746.txt.

[118] R. Yavatkar, D. Pendarakis, and R. Guerin. *A Framework for Policy-based Admission Control*. RFC 2753. IETF, Jan. 2000. URL: http://tools.ietf.org/rfc/rfc2753.txt.

[119] Peng Zhang et al. "APKeep: Realtime Verification for Real Networks". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 241–255. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/zhang-peng.

Generating priority combinations for experiments

Priority combinations form a dimension in the parameter space for experiments (see Chapter 3.2.1). We notice that the combinations depend on the number of control applications involved in the experiments. For example, two applications a and b yield four combinations:
2 2 (application a has the priority of 2, so is application $b \rightarrow pri_a = pri_b$)
2 3 (application a has the priority of 2, application b has the priority of 3 $\rightarrow pri_a < pri_b$)
3 2 (application a has the priority of 3, application b has the priority of 2 $\rightarrow pri_a < pri_b$)
3 3 (application a has the priority of 3, so is application $b \rightarrow pri_a = pri_b$)
The last combination of "3 3" is a repetition of "2 2" since these two applications have the same priority.

Note that the choice of 2 and 3 for priority assignment on the two applications is not important. One can choose 1 and 2, or 1000 and 1001, it does not matter as long as we have two different numbers for two applications. Likewise, we need three different numbers for three applications. We choose 2, 3 and 4 in the below case containing three control applications.

For 3 applications a, b and c, there are 13 combinations:

```
#     Main combi.   Pattern          Duplicated combi.
1     2 2 2         # a=b,a=c,b=c    3 3 3   4 4 4
2     2 2 3         # a=b,a<c,b<c    2 2 4   3 3 4
3     2 3 2         # a<b,a=c,b>c    2 4 2   3 4 3
4     2 3 3         # a<b,a<c,b=c    2 4 4   3 4 4
5     2 3 4         # a<b,a<c,b<c
6     2 4 3         # a<b,a<c,b>c
7     3 2 2         # a>b,a>c,b=c    4 2 2   4 3 3
8     3 2 3         # a>b,a=c,b<c    4 2 4   4 3 4
9     3 2 4         # a>b,a<c,b<c
10    3 3 2         # a=b,a>c,b>c    4 4 2   4 4 3
11    3 4 2         # a<b,a>c,b>c
12    4 2 3         # a>b,a>c,b<c
13    4 3 2         # a>b,a>c,b>c
```

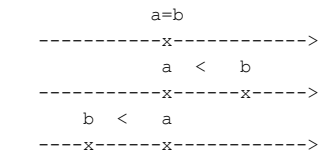
For 4 applications, the total number of combinations is 75, it is 541 for 5 applications and 4863 for 6 applications.

There are different ways to solve this priority generation problem. We choose the recursive solution based on the observation of the combinations on an axis.

For 1 application, there is 1 point on the axis

```
            a
  ----------x------------>
```

Adding application b, there are 3 possibilities



$\rightarrow$ there are 1 case of 1 point and 2 cases of 2 points.
Similarly, adding application c, there are 4 possibilities
+ from 1 point $a = b$, 1 other point $a = b = c$ and 2 other points: $a = b > c, a = b < c$ are generated,
+ from each case of 2 points, e. g., $a < b$, there are 2 other 2 points $(a < b = c,\ a = c < b)$ and 3 other 3 points $(a < b < c,\ a < c < b,\ c < a < b)$ generated. We obtained 2 cases of two points in the beginning.
$\rightarrow$ in total, there are:
1(1point) + 2(2point) + 2(2(2point)+3(3point)=1(1point)+6(2point)+6(3point)=13 cases.
This method can be applied recursively for n applications.
Denote $f(n, k)$ as the number of cases having k points when n applications are considered. For instance, $f(2, 1)$ represents the number of cases of 1 point in the axis when considering 2 applications.
We have

```
f(1,1)=1
f(2,1)=1, f(2,2)=2
f(3,1)=1, f(3,2)=2[(f2,1)+f(2,2)], f3,3)=3[f(2,2)]=3!
f(4,1)=1, f(4,2)=2[f(3,1)+f(3,2)], f(4,3)=3[f(3,2)+f(3,3)], f(4,4)=4[f(4,3)]=4!
```

The total number of combinations for n applications is

```
g(n)=f(n,1)+f(n,2)+...+f(n,k)+...+f(n,n)
```

We can generalize the recursive functions as follows

```
f(n,1)=1
f(n,n)=n!
f(n,k)=k*(f(n-1,k-1)+f(n-1,k))
g(n)=f(n,1)+f(n,2)+...+f(n,k)+...+f(n,n)
```

The above functions can be written in the R programming language[1] as follows

```
fac <- function(n){ if (n == 1) return (1)
                    else return (n*fac(n-1)) } #calculate n!
```

1 https://www.r-project.org/about.html

```
f <- function(n,k){ if (k==1) return (1)
                    else if (n==k) return (fac(n))
                    else return (k*(f(n-1,k-1)+f(n-1,k)))}
g <- function(n) {s=0; for (i in (1:n)){ (s = s+f(n,i)) }; return (s) }
                #e.g., g(3)=13
```

The script *pri_gen.py* in our published codebase[2] can be used to generate each of these combinations. Further instructions on the arguments are provided in the beginning of this script.

2 https://github.com/mnm-team/sdn-conflicts

Acronyms

ACL	Access-Control List
API	Application Programming Interface
ARP	Address Resolution Protocol
CIM	Common Information Model
COPS	Common Open Policy Service
CORBA	Common Object Request Broker Architecture
DAG	Directed Acyclic Graph
DAML+OIL	DARPA Agent Markup Language + Ontology Inference Layer
DMTF	Distributed Management Task Force
EpLB	End-point Load Balancer
FTP	File Transfer Protocol
FW	Firewall
HS	Host Shadowing
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
ICMPv6	Internet Control Message Protocol version 6
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPv6	Internet Protocol version 6
IRTF	Internet Research Task Force
ISO	International Organization for Standardization
ITU	International Telecommunication Union
JSON	JavaScript Object Notation
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
LLDP	Link Layer Discovery Protocol
MEADcast	Multicast to Explicit Agnostic Destinations
MIB	Management Information Base
NDP	Neighbor Discovery Protocol
NETCONF	Network Configuration Protocol
NFV	Network Function Virtualization
OASIS	Organisation for the Advancement of Structured Information Standards
ONF	Open Networking Foundation
OSI	Open Systems Interconnection
OWL	Web Ontology Language
P4	Programming Protocol-Independent Packet Processors
PBNM	Policy-Based Network Management

PCIM	Policy Core Information Model
PDL	Policy Description Language
PE	Path Enforcer
PLB	Path Load Balancer
POF	Protocol Oblivious Forwarding
PPLB4D	Destination-based Passive Path Load Balancer
PPLB4S	Source-based Passive Path Load Balancer
RDF	Resource Description Framework
REST	Representational State Transfer
RSVP	Resource Reservation Protocol
SDN	Software-Defined Networking
SNMP	Simple Network Management Protocol
SPF	Shortest Path First
TCP	Transmission Control Protocol
TE	Traffic Engineering
UDP	User Datagram Protocol
UML	Unified Modeling Language
VALID	Virtualization Assurance Language for Isolation and Deployment
XACML	eXtensible Access Control Markup Language
XML	Extensible Markup Language

Glossary

Many definitions are reused from RFC 7426 [42] and from the OpenFlow Switch specification [78]. Those frequently mentioned in this book are presented here for the sake of easy reference.

Action *"an operation that acts on a packet. An action may forward the packet to a port, modify the packet (such as decrementing the TTL field) or change its state (such as associating it with a queue). Most actions include parameters, for example, a set-field action includes a field type and field value"* [78].
Bug an anomaly in the network behaviour caused by errors pertaining to a control application, not by the interference between two or more control applications.
Conflict an anomaly in the network behaviour caused by the interference between two or more control applications.
Control Application (or simply application) *"a piece of software that utilizes underlying services to perform a function"* [42], for example, topology discovery is a common service employed by control applications.
Control Plane *"the collection of functions responsible for controlling one or more network devices. The control plane instructs network devices with respect to how to process and forward packets"* [42].
Data Plane (or forwarding plane) *"the collection of resources across all network devices responsible for forwarding traffic"* [42].
Datapath (specific to OpenFlow) *"the aggregation of components of an OpenFlow logical switch that are directly involved in traffic processing and forwarding"* [78].
Flow a sequence of packets between a source and a destination, which share the same attributes. The mentioned attributes include fields in the protocol header, such as IP addresses, MAC addresses, status bits, they can also be the ingress port of a packet arriving at a network device. A flow is expressed as a rule (or a flow entry) in the rule table of each network device on the path from the source to the destination, and can be reflected differently in these devices.
Forwarding *"deciding the output port or set of output ports for a packet, and transferring that packet to those output ports"* [78].
Header *"control information embedded in a packet used by a switch to identify the packet and to inform the switch on how to process and forward the packet. The header typically includes various header fields to identify the source and destination of the packet, and how to interpret other headers and the payload"* [78].
Header Field *"a value from the packet header. The packet header is parsed to extract its header fields which are matched against corresponding match fields"* [78].

Match Field *"a part of a flow entry against which a packet is matched. Match fields can match the various packet header fields, the packet ingress port, the metadata value. A match field may be wildcarded (match any value) and in some cases bitmasked (match subset of bits)"* [78].
Matching *"comparing the set of header fields and pipeline fields of a packet to the match fields of a flow entry (or rule)"* [78].
Network Device *"an entity that receives packets on its ports and performs one or more network functions on them. Network devices can be implemented in hardware or software and can be either physical or virtual"* [42].
OpenFlow Controller *"an entity interacting with the OpenFlow switch using the OpenFlow switch protocol. In most case, an OpenFlow Controller is software which controls many OpenFlow Switches"* [78].
OpenFlow Protocol the protocol used for the communication between the OpenFlow Controller and OpenFlow Switches.
OpenFlow Switch *"a set of OpenFlow resources that can be managed as a single entity, includes a datapath and a control channel"* [78].
Packet *"a series of bytes comprising a header, a payload and optionally a trailer, in that order, and treated as a unit for purposes of processing and forwarding"* [78].
Port where packets enter and exit an SDN device, may be a physical or a logical port, e. g., port Controller in an OpenFlow switch [78].
Rule (or flow entry) an element in a rule table used to match and process packets. It contains a set of match fields for matching packets, a priority for matching precedence, and a set of actions to apply. A rule in a network device can also contain a set of counters to track packets [78].
Rule Table (or flow table) a group of rules residing in a network device. When a packet arrives at the network device, rules in the rule table are searched based on a predefined matching policy (e. g., first match) to match that packet.
Software-Defined Networking *"a programmable networks approach that supports the separation of control and forwarding planes via standardized interfaces"* [42].
Traffic Path the path including all network devices that the traffic under consideration traverses from its source to its destination.

The SDN architecture facilitates the flexible deployment of network functions. While promoting innovation, this architecture induces yet a higher chance of conflicts compared to conventional networks. The detection of conflicts in SDN is the focus of this work.

Restrictions of the formal analytical approach drive our choice of an experimental approach, in which we determine a parameter space and a methodology to perform experiments. We have created a dataset covering a number of situations occurring in SDN. The investigation of the dataset yields a conflict taxonomy composed of various classes organized in three broad types: local, distributed and hidden conflicts. Interestingly, hidden conflicts caused by side-effects of control applications' behaviour are completely new.

We introduce the new concept of multi-property set, and the ·r ("dot r") operator for the effective comparison of SDN rules. With these capable means, we present algorithms to detect conflicts and develop a conflict detection prototype. The evaluation of the prototype justifies the correctness and the realizability of our proposed concepts and methodologies for classifying as well as for detecting conflicts.

Altogether, our work establishes a foundation for further conflict handling efforts in SDN, e.g., conflict resolution and avoidance. In addition, we point out challenges to be explored.

Cuong Tran won the DAAD scholarship for his doctoral research at the Munich Network Management Team, Ludwig-Maximilians-Universität München, and achieved the degree in 2022. He loves to do research on policy conflicts in networked systems, IP multicast and alternatives, network security, and virtualized systems. Besides, teaching and sharing are also among his interests.

44,00 €
ISBN 978-3-487-16326-0

www.olms.de