INSTITUTE OF INFORMATICS
LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

Experimental Examination of Distributed Conflicts in Software Defined Networks

Nicholas Reyes

INSTITUTE OF INFORMATICS
LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

Experimental Examination of Distributed Conflicts in Software Defined Networks

Nicholas Reyes

Supervisor:	PD Dr. Vitalian Danciu
Advisors:	Cuong Ngoc Tran (LMU München) Reinhard Gloger (LRZ)
Submission:	October 5, 2021

I hereby confirm that I have written the accompanying thesis myself, without contributions from any sources other than those cited in the text and acknowledgements. This applies also to all graphics, drawings, maps and images included in the thesis.

Munich, October 5, 2021

. . . . . .
(Candidate's signature)

Abstract

This thesis explores policy conflicts in Software Defined Networks (SDN) between multiple applications and across multiple nodes in a network. The research questions comprise a concept for reproducible experimental infrastructures to examine conflicts and the development of new applications that enforce policies. In experiments, the new implementations are deployed in combination with existing applications in order to discover new conflict classes. For any discovered conflict classes, detection mechanisms are required. The role of presented implementations while taking into account security in SDN is evaluated. We examine existing work and the technical background to the presented approach in order to answer these topics. A software architecture for experiment automation is devised, and a set of designed and random network topologies serve as input for the experimental process. The results include new conflict classes based on the tested networks and applications, detection algorithms for the conflicts and an experimental architecture that facilitates further research. Three approaches to generate random networks that display realistic properties and three new control applications are presented. Insights on the validity of discovered distributed conflicts and the soundness of presented detection methods as well as experiment automation are gained by a second set of experiments on additional, untested networks. The results advance the understanding of distributed conflicts in SDN and promote future approaches to conflict resolution and interpretation.

Zusammenfassung

Diese Arbeit untersucht Policy-konflikte in Software Defined Networks (SDN) zwischen mehreren Anwendungen und mehreren Knoten in einem Netz. Die Forschungsfragen umfassen ein Konzept für reproduzierbare experimentelle Infrastrukturen zur Untersuchung von Konflikten und die Entwicklung neuer Anwendungen, die Richtlinien durchsetzen. In Experimenten werden die neuen Implementierungen in Kombination mit bestehenden Anwendungen eingesetzt, um neue Konfliktklassen zu entdecken. Für alle erkannten Konfliktklassen sind Erkennungsmechanismen erforderlich. Die Rolle der vorgestellten Implementierungen unter Berücksichtigung der Sicherheit in SDN wird evaluiert. Wir untersuchen bestehende Arbeiten und die technischen Hintergründe des vorgestellten Ansatzes, um diese Themen zu beantworten. Eine Softwarearchitektur für automatisierte Experimente wird entwickelt, und eine Reihe von entworfenen und zufälligen Netztopologien sind die Basis für den experimentellen Prozess. Die Ergebnisse umfassen neue Konfliktklassen, Erkennungsalgorithmen für die Konflikte und eine experimentelle Architektur, die weitere Forschung ermöglicht. Drei Ansätze zur Generierung von zufallsbasierten Computer-Netzen, die realistische Eigenschaften aufweisen, und drei neue SDN-Anwendungen werden vorgestellt. Erkenntnisse über die Validität entdeckter verteilter Konflikte und die Zuverlässigkeit der vorgestellten Erkennungsalgorithmen und die Automatisierung werden durch eine zweite Reihe von Experimenten an zusätzlichen, ungetesteten Netzen gewonnen. Die Ergebnisse fördern das Verständnis verteilter Konflikte in SDN und zukünftige Ansätze zur Konfliktlösung und -interpretation.

Contents

1 Introduction

Effective and secure operation of networks is an unresolved issue in light of IT services that require reactive, adaptive and highly automated computing environments [CSb]. In conventional networks, nodes are dedicated to a specific purpose with e.g. routing, switching or traffic filtering capabilities and need to be managed individually. Each node comprises a forwarding plane consisting of rules for forwarding network traffic, a control plane for decision processes based on the characteristics of incoming traffic not handled by existing data plane policies and a management plane that enables administration of the two previous planes and the node itself.
The concept of Software Defined Networks (SDN) encompasses a paradigm shift by devolving management and control facilities from individual nodes into centralized entities. Hardware properties can be abstracted by exploiting communication protocols such as OpenFlow [ONFb], that provide a common interface to the data plane via a central controller node. Furthermore, controllers can be interfaced to an arbitrary number of applications that address the need for enforcing network functions such as traffic filtering and load-balancing concurrently. Consequently, a network is transformed to a modular, programmable architecture and networking policies are implementable on an application level by separate teams or vendors. While SDN enable adaptable, pluggable but centralised decision making, control applications are agnostic to each other and can exhibit conflicting intents as shown by Tran and Danciu [TD20]. Moreover, the assumption itself suggests that an open interface can be exploited for malicious intents.
A study conducted by Cisco Systems in 2019 [CSa] shows that network strategists and overseeing IT leaders across the world consider network automation (25%) and SDN (23%) as high impact topics for future network strategies. The following report in 2020 [CSb] states that 44% of traffic is expected to be processed by systems based on SDN or Network Function Virtualization (NFV) by the year 2021, highlighting the necessity of automated conflict detection and mitigation on general principles. NFV is not dependent on but benefits from centralization, further indicating the importance of SDN. State-of-the-art implementations and current research on conflict detection and resolution are lacking in their ability to address the need for generality. Problem scenarios and their resulting conflict patterns are engineered for distinct use cases. Detection and resolution are realized by comparing existing policies in the data plane [PNC+19][LCL+18][PSY+12] or by confronting existing policies with predefined network invariants [KZCG12] and management models [DSNW15][SMR+14].
The findings by Tran and Danciu [TD19] exemplify hidden conflicts through side-effects that disable the expected behavior of the application layer. Race conditions let one application enforce rules to handle traffic, such that other control applications interested in the same traffic patterns are not notified. The authors conclude that detection mechanisms should not be based on simple examination of the data plane state but also on predictive measures and a thorough understanding of conflict patterns to realize conflict resolution mechanisms. Tran and Danciu [TD20] introduce a framework [TD20] to define and detect conflicts based on a generalized scope and parameter space. The parameter space includes dimensions that

are oblivious of data plane policies such as the start order of control applications and the characteristics of a network topology. The introduced methodology is based on choosing an unexplored object in the parameter space to conduct a suitable experiment. The exhibited and expected network behavior are compared and deviations are analyzed to derive conflict classes. The proposed detection engine by Tran and Danciu [TD20] requires a possibly exponential set of conflict patterns and the mentioned predictor [TD19] can also benefit from a set of known input patterns to anticipate hidden conflicts.
In this work, the introduced methodology is implemented through a software architecture with concise inputs and outputs to facilitate experiments on arbitrary, untested and complex infrastructures. The experiments comprise a set of designed and random network topologies to expose distributed conflicts. These conflicts arise due to the deployment of routing policies on multiple nodes and resulting inter-policy conflicts as defined by Hamed and Al-Shaer [HA06], as well as the side-effects elicited by independently running multiple control applications and a missing mechanism for their coordination. Detection methods for newly discovered conflicts complement an existing prototype by Tran [Tra22]. The deployment of control applications processing the same traffic characteristics and targeting multiple SDN switches is the foundation for examining previously unknown conflict classes. They are analysed and validated through a prototype for conflict detection. The detection engine resides in the control layer and is built into the controller processing chain as a means of passively intercepting conflicts. The prototype and applied methodology demonstrate the importance of detection and interpretation, based on a common set of parameters.

1.1 Distributed Conflicts

Policy conflicts and their causes are usually defined in a specific, limited context as explained in Chapter 2. A conflict is given if the data plane enters an inconsistent state, i.e. flow entries are contradicting, invariants such as bandwidth and availability are violated or performance degrades. Distributed conflicts are based on the data plane of all nodes in a topology. The definition is not bound to a use case or a specific topology shape. Figure 1.1 depicts an occurrence of a distributed conflict and the base requirements for the network setup. Any topology that comprises at least two switches and a minimum of two control applications that are deployed on separate switches is viable. Figure 1.1 shows a local conflict based on ambiguous policies within one switch for forwarding packets from host A to host B. On the right side, two conflicting policies residing in separate switches constitute a distributed conflict with an equivalent outcome of a blocked network connection. The two patterns are also defined as intra-policy and inter-policy conflict by Hamed et al. [HA06].

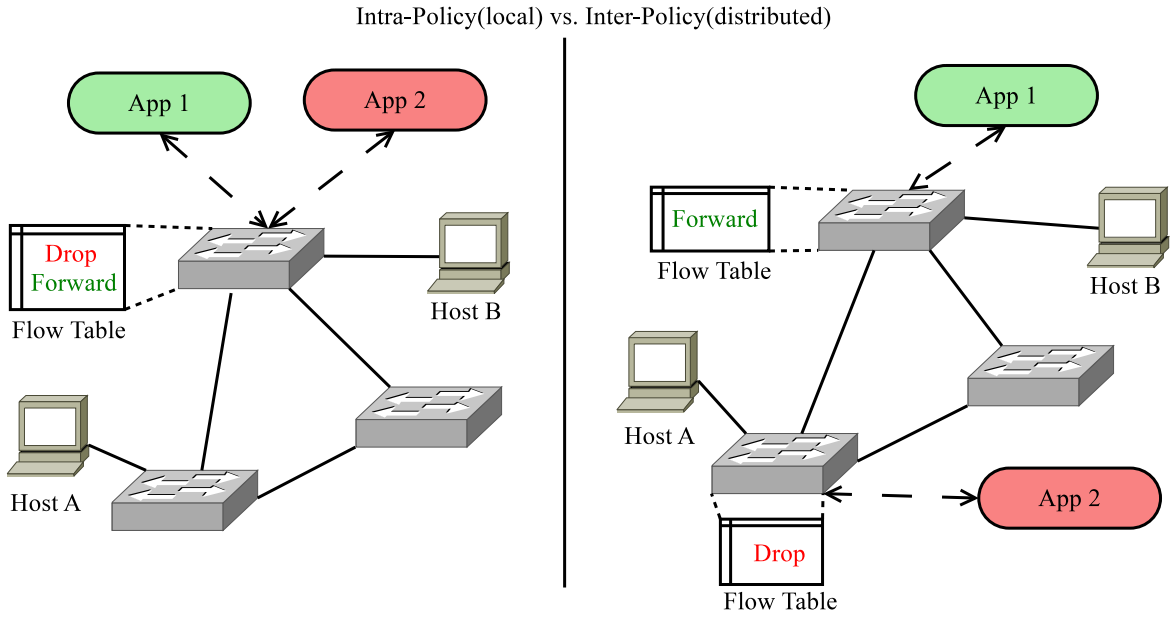

Figure 1.1: Scenario for a distributed conflict with the minimal environment of at least two switches and two individually deployed control applications, compared to a local policy conflict within one switch. The example is based on the definition of intra- and inter-policy conflicts by Hamed et al. [HA06].

1.2 Motivation

One critical aspect of operating network infrastructure is the identification of problems that encompass contradicting policies, inefficiency and unsolicited side-effects. Some of these issues can only be observed in production and others are detectable in test environments. Gaining insights from experiments and their validation is the key motivation for this work. OpenFlow [ONFb] provides the protocol for obtaining insights on these problems, but lacks an implementation for analyzing relevant data. As discussed in Chapter 2, current research and technologies analytically engineer conflicts and present an implementation that solves them. The scope of these conflicts is limited to a confined use case or a specific topology. In current work exemplified policy deviations usually reside within one node and are purely theoretical in some cases. Chapter 2 demonstrates the need to identify and classify conflicts between nodes and between applications through an empirical approach. In SDN an interface between control applications does not exist and asynchronous communication from controllers to applications can lead to race conditions and overlapping decisions [TD19]. With regard to flexibility and independent life cycles for control applications, such an interface could even prove undesirable. The centralization in SDN allows for negotiation between control functions, provided that the cause for conflicts is well-defined and conflict resolution does not inhibit SDN principles. Continual expansion of the currently incomplete set of conflict classes is necessary to advance the means for analysis as well as interpretation, and to provide for reproducible results across use cases, networks and implementations. Network engineers can benefit from a set of validated patterns to understand issues. Since automated resolution of conflicts can lead to unintended data plane states, mitigation and manual res-

olution is considered a sound approach. Once the source of a conflict is clear, constructive resolution is possible and control applications stay free of defects. Generating and analyzing data on distributed conflicts supports this effort.

1.3 Problem Statement

This work examines distributed conflicts in generic and automated testbeds, since current research is mostly focused on local conflicts and specialised conflict environments. To achieve this the following topics need to be explored.
Currently unconnected and specialized implementations for creation, definition and deployment of network topologies, configuration of the SDN context and software stack, as well as monitoring of network state need to be interfaced to achieve a sophisticated level of SDN experiment automation and a generic implementation to allow for integration of new control applications. A consolidating architecture for conflict exploration and detection requires a common approach for analyzing data plane state, independent of specific control applications. The implementation needs to accommodate input and automated generation of application configurations, which are then used to determine application intent as an input for conflict detection. Since applications are the source of SDN conflicts we will contribute new implementations and evaluate them in co-deployment with those devised by Tran [Tra22]. Furthermore, an interface for experiment deployment that abstracts from irrelevant technical details and ensures reproducible experiments and accessible results needs to be determined. Next to applications, network topologies are an important dimension for experimental exploration of conflicts. We examine an approach to generate random topologies that display relevant and realistic features. For newly exposed conflict classes automated detection mechanisms are devised and evaluated. We examine existing work on security in SDN an evaluate the presented architecture in light of the defined requirements.

1.4 Methodology

The procedure for achieving the goals in the problem statement are based on introducing a formal model for experiments in SDN. The model addresses the parameters of the executed experiment and serves as a common definition for creating its technical infrastructure. From the model the nodes and edges of a topology are created and the configuration of an experiment is derived, including control applications and expected invariants as input for the detection algorithm. The parameter space [TD20] depicted in Figure 2.5 is the point of origin for each iteration in the applied methodology for experiments. The existing framework [Tra22] for examining conflicts in SDN is augmented towards enhanced automation for setup of the technical infrastructure and processing of generated data. An experimental run is started automatically, any activity and it's output is processed by a conflict monitoring and detection prototype, and the resulting dataset is analysed. The proof-of-concept includes the implementation of a detection algorithm to reaffirm known conflicts and mark the observed data plane state in consideration of the expected state. Identifying relevant sections immediately through the prototype is key for more efficient inspection. The existing applications used for the experiments, conceived and described by Tran [TD20], are complemented by three new control applications. This provides for a diverse set of control applications to elicit distributed conflicts. The motivation for this work calls for examining

the topology definition process. Designed networks based on common or intentional concepts serve as a blueprint and can be both realistic or synthetic. Furthermore, randomized topologies are inspected, too. This approach provides for a balance between exploitation of known domains and their characteristics as well as exploration of possibly unexpected conflict domains. The devised experimental architecture for gradual coverage of the parameter space is paramount for reproducible experiments and results. Formal criteria are presented for describing distributed conflicts, which serve as a basis for detection in the proof-of-concept and provide a common format for formulating new classes. The collected data is reviewed based on the flagging by the prototype and any derived conflict classes are added to the prototype. The classes are presented and evaluated to determine if they adhere to the definition of distributed conflicts. New conflict classes, their characteristics and the composition of the experimental parameters are evaluated based on the generated data.

1.5 Results

A concise summary of the results is given by the following points:

- an experimental architecture for examination of SDN policy conflicts
- an approach for generation of random topologies for the experimental process
- new distributed conflict classes
- detection algorithms for the conflict classes where feasible
- an evaluation of the introduced detection mechanisms and the detection prototype
- an evaluation of security in SDN which is applied to the implementations

The contribution of this work supports and extends the approach of Tran and Danciu [TD20] to identify and define distributed conflicts in SDN through experiments and a standardized methodology. The results include the definition of new conflict classes and a dataset for further processing. The introduction of a model for generating a dedicated, virtualized infrastructure increases the level of automation, which promotes reproducible experiments. Similar experiments on the parameter space can be conducted based on the given instructions. Conflict detection based on a set of well-defined criteria is demonstrated by the implemented prototype. Uncovering symptoms of unknown conflict types by comparing expected and observed data plane state between application deployments proves the utility of the methodology for this work and it's feasibility for future work. The methodology, experimental design and parameter space is agnostic to technical implementations and serves as a guideline for generating, defining and interpreting conflicts in SDN. Figure 1.3 depicts the components of the proof-of-concept. The detection engine resides in the control layer and registers new modifications of the forwarding plane. The impact of new policies on the current data plane state is evaluated based on a set of given invariants or manually to identify other inconsistencies. It facilitates detection based on known conflict classes as well as manual interpretation, where necessary. The prototype as an element of the methodology proves the value of conflict detection over engineering conflict patterns and scenarios. An evaluation of flow characteristics used for conflict detection promotes a continuous and generalized expansion of conflict classes. The necessity of an approach based on interpretation

of application intent is outlined, be it for the definition of conflicts, or their resolution and monitoring. The evaluation of the examined parameter space and relevant flow characteristics helps narrow the focus on promising criteria for conflict definition and detection. This work entails a stable and reproducible experimental process with well-defined, coherent inputs and outputs.
The experiments elicit known distributed conflict classes in the SDN context as well as new conflicts that promote further research on detection and resolution of distributed SDN conflicts. The taxonomy of conflict classes is shown in Figure 1.2. The taxonomy includes Spuriousness and General Multi-Transform conflicts in the SDN context, which are inspired by the work of Hamed and Al-Shaer [HA06].

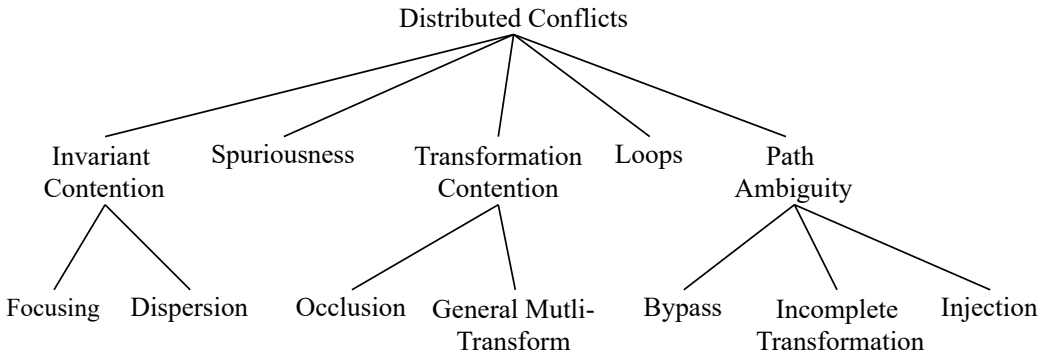

Figure 1.2: Taxonomy of detected distributed conflict classes in the SDN context. Conflict classes are grouped under a parent node if their conflict patterns relate.

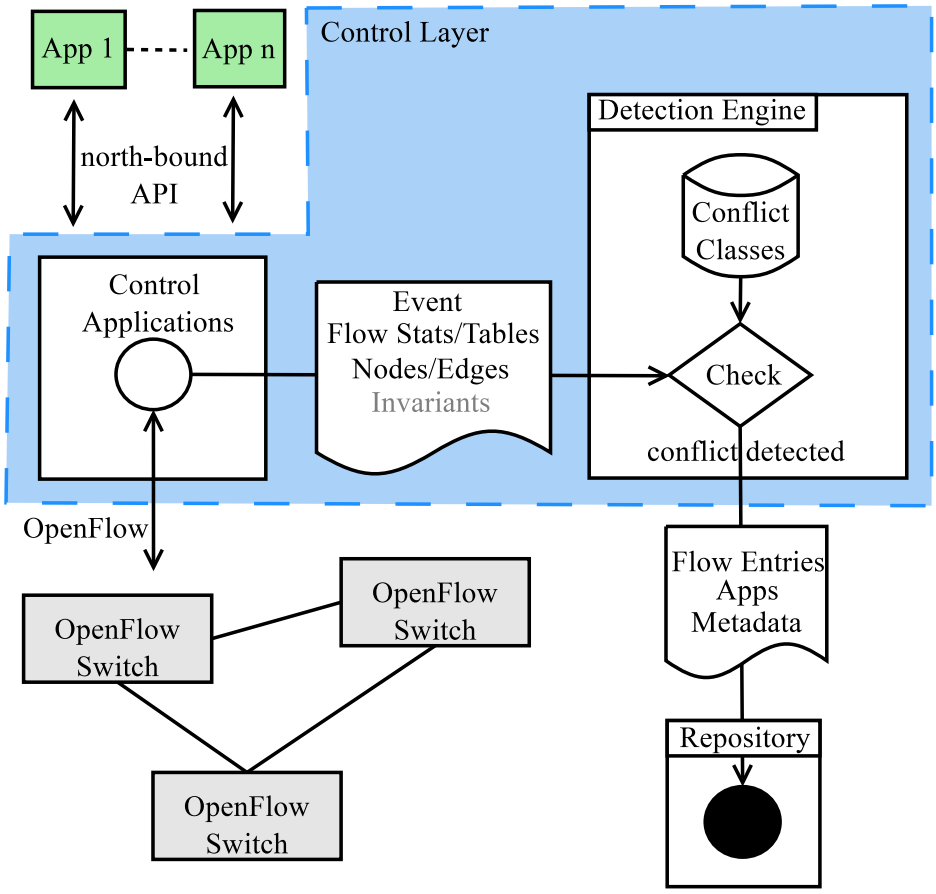

Figure 1.3: Components of the proof-of-concept that detects discovered distributed conflicts. The implementation attaches a conflict detection engine notified of flow table modifications to determine violations of expected network state. The generated data is stored for manual interpretation.

1.6 Structure

This thesis is structured into seven chapters. The first chapter covers the problem statement and motivation as well as the methodology for their accomplishment. An overview of the results and ensuing benefits is given. Chapter 2 provides an overview of relevant background knowledge on technologies and current work. In addition to the concept of SDN and OpenFlow [ONFb] as an implementation, the researched scenarios for SDN conflicts and existing conflict classifications are described. Details of the applied experimental design, the characteristics of examined topologies as well as individual features of the newly developed and adapted control application and the software architecture of the experimental process are illustrated in Chapter 3. Chapter 4 comprises the implementation of the presented architecture for experiment automation. An experiment model serves as the input for the experiment building process and the following experiment execution. The chapter includes step-by-step instructions for an example deployment. Results on distributed conflict classes are presented in Chapter 5, starting with the required definitions, followed by details and examples on the conflicts. Technical details of the proposed detection prototype and detection algorithms for

the conflict classes presented in Chapter 5 are explained in Chapter 6. The reasoning behind the algorithms is given and consequences are outlined. Chapter 7 comprises a qualitative evaluation of the experimental architecture compared to similar work. The utility of the conflict classes and detection algorithms are reviewed. The detection prototype, which is based on a rule graph 6.1, is evaluated. Traffic flow characteristics and their value for detection are presented. Furthermore, the parameter space as a basis for conducting experiments is assessed. The final Chapter 8 includes an overview of future improvements and closes with a summary of the conducted research.

2 Background

As a first step the concept of Software Defined Networks are explained to highlight the differences to traditional networking. Said differences, especially centralized decision making, serve as a basis for the implementations in this and related work and implicate the possibilities and limits for conflict detection and resolution. The experimental approach for discovering conflicts is independent of SDN in theory, but the technical infrastructure and detection algorithms are closely related to characteristics that are specific to SDN and its implementation. This chapter conveys the common terminology of related work as well as the utilized communication interfaces and data structures in detection prototypes. Outlining the existing approaches shows why the approach in this work is different. It serves the understanding of requirements for the automation process and the current shortcomings. The insights on security in SDN through the discovered results is put into a broader context by examining security related aspects of current research.
Recent and older work generally limits the context for the definition of conflicts to a certain type of control application, use case, conflict criteria or topology. Consequently, a comprehensive study of conflict patterns incorporating and complementing the following approaches is necessary. This chapter gives an overview of the scope of conflicts, how they can be classified, which dimensions influence them and the existing technologies for handling them.

2.1 Software Defined Networks

Understanding the characteristics and definition of SDN requires knowledge on the modeling of conventional networks. The functions of a network are assigned to three network planes as depicted on the left side of Figure 2.1, the data plane, control plane and management plane. Connected and closely coupled, the planes are implemented in vendor firmware with distinct commands and interfaces. They are isolated within each node that performs a specific network function, such as routing, switching or traffic engineering.
Also referred to as the forwarding or user plane, the data plane satisfies the function of transmitting data packets that enter the network. In order to determine the destination of a packet, its protocol headers are dissected and processed according to currently installed policies. The policies are cached in data structures referred to as forwarding information bases (FIB) or routing information bases (RIB) and are again vendor specific. The policies for the data plane are provided by the control plane. While the data plane is a snapshot of how the user packets flow through the infrastructure, the control plain administers the state of the network itself. It comprises the algorithms for decision-making processes and the topology state is captured in routing tables, a cache for media access control (MAC) or Internet Protocol (IP) addresses or other dedicated structures. Decisions are communicated to the data plane and inserted as policies to forward, modify or block packets. Also the decisions and information of a nodes view on the network might be propagated consecutively to adjacent nodes. This enables a common view on the network to find optimal paths, avoid loops and other functions. Decisions are reached within a node and can only be shared

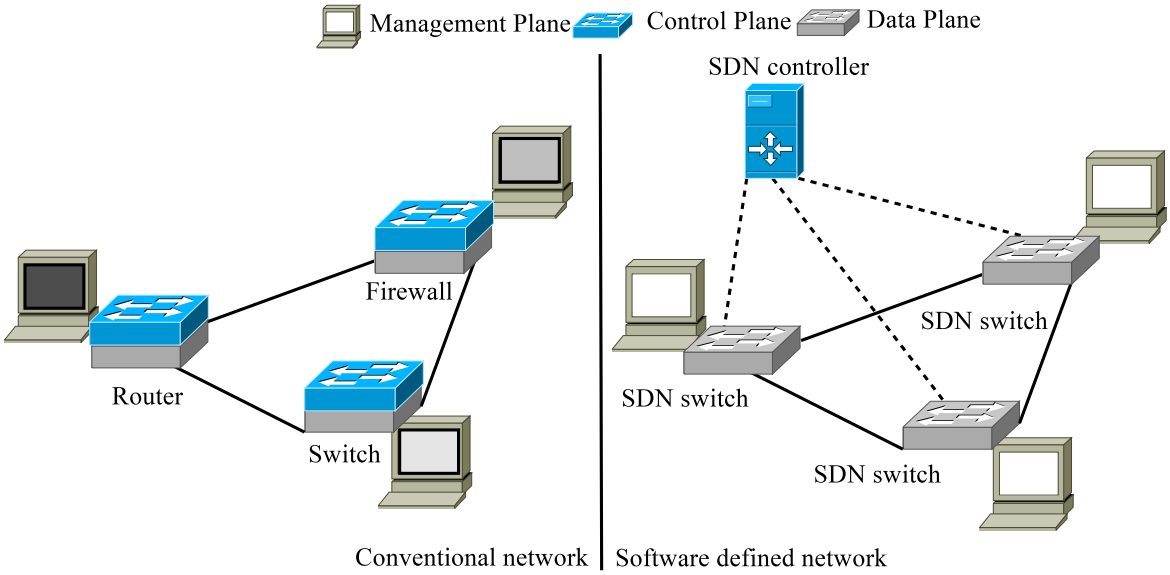

Figure 2.1: In traditional networks, management, control and data plane are contained within each node and decisions are reached individually. In software defined networks, the control plane is detached from individual nodes, decisions are centralized in a controller node and the management plane is based on a common interface.

with adjacent nodes, without a central instance. An administrator configures hardware on the management plane. Access is usually provided by a user interface that provides custom features between vendors and based on proprietary configuration parameters. Nevertheless, this often results in an implementation of the same network functions and represents an unnecessary layer of complexity for the administrative process.
The concept of SDN is a decoupling of the three network planes as shown in Figure 2.1 on the right side. In order to implement centralized decisions, dynamic handling of topology changes and hardware agnostic network functions, the planes are not implemented in distinct firmware, but in a common software stack that is deployed on generic nodes. The data plane still persists in individual nodes, but they are connected to a central controller node. When lacking a policy for processing and forwarding a packet, it is sent to the controller along with additional information. Data plane structures are not bound to a specific network function. In SDN a node can serve several of these and facilitates convoluted policies across Open Systems Interconnection (OSI) layer protocols. The most relevant part of the OSI layering model for this work comprise layer addresses, such as IP (layer 3) and MAC (layer 2) addresses as well as Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) ports (layer 4). Controllers implement the control layer for the entire network and handle the decision-making process. Decisions are communicated to all affected nodes and translated into hardware specific data plane policies through the common software interface. This avoids the need for propagating control layer information between adjacent nodes. Nodes are still administrated individually but through a common interface and based on the same set of commands for configuration. The software abstracts from the type of hardware or specific operating systems, reducing complexity and promoting a programmatic approach.

The characteristics of SDN are more appropriately described by a layering model depicted in Figure 2.2, which accounts for the workings of the centralized control layer and the programmable, dynamic networking features. The decoupling of network planes necessitates an interface to communicate decisions. A south-bound application programming interface (API) connects control and infrastructure layer, the later corresponding to the data plane with its generic SDN switches. The term switch does not describe a nodes function and is simply a naming convention.
The control layer is usually made up of a monolithic controller. A multi-controller environment is viable but counteracts centralized decision-making. Controllers can implement base functions such as loop detection or topology discovery. One novel characteristic of SDN is the north-bound API that provides an interface to third party control applications and allows for programmable network functionality. The application layer can comprise several independent applications, contributing to the decision-making process simultaneously for similar or completely different types of network traffic. They can be added or removed without disrupting the network.
SDN entail a regression due to the increased administrative network load and the bottleneck represented by controller channels. Controllers are single points of failure and can be subjected to high loads in terms of bandwidth and processing. The distribution of policies from the controller to the target nodes is paramount to a functional network. In traditional networks, nodes can reach decisions independently according to their function but might fail to propagate information to their neighbors. Thus broken connections pose problems under both paradigms.
The benefits of SDN are given by the dynamic process for adding network functions and adapting the network structure. Nodes in the infrastructure layer need not be changed physically to perform a new role. Controllers or control applications can query a node to determine its capabilities and formulate their policies accordingly. The state of the network is monitored and adaptable by the control layer. Therefore, SDN permit for programmatic, transitory realignments and reconciliation of contradicting functions. No physical machines need to be replaced or new software needs to be added to realize any new network function or structure, and in a virtualized environment nodes can be inserted dynamically.

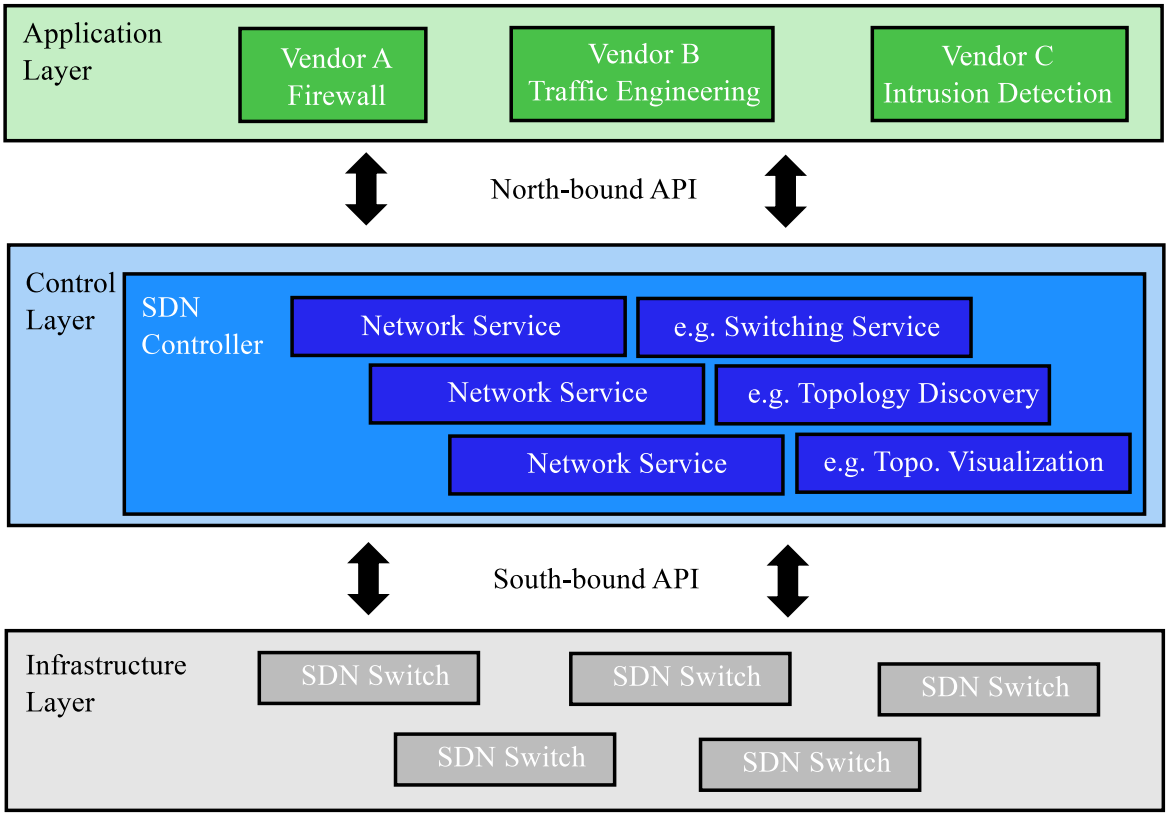

Figure 2.2: Layering model of software defined networks. The figure depicts application, control and infrastructure layers which are interfaced via the south- and north-bound application programming interface to exchange information. The figure is an adaption of Figure 1 in a white paper [ONFa] of the Open Networking Foundation. Additional input was taken from RFC 7426 [IRTF] and the survey by Kreutz et al. [KRV+15] on SDN.

2.2 OpenFlow

OpenFlow comprises a specification for hardware and software capabilities of network devices as well as a protocol for communication between SDN layers. The following information is based on Version 1.3 [ONFb] of the specification. The application layer is not explicitly mentioned. Its capabilities are comparable to control layer functions but dependent on the communication interface between control and application layer. The relevant components of an OpenFlow switch and their relations are depicted in Figure 2.3. OpenFlow switches may be virtualized over hardware.
Network packets enter or leave the switch through physical or logical ports, which again are categorized as standard and reserved ports that facilitate required and optional functions. Physical ports can be one-to-one mappings to network interfaces or a slice of an interface in case of a virtualized instance. Logical ports may map to several physical ports and can represent abstractions unknown to OpenFlow, such as loopback interfaces or tunneling devices. Reserved ports enable forwarding functions such as flooding, local management and most importantly must include one instance that represents the controller channel.

The controller channel acts as an output and input port for information that is sent to and from controllers. Controllers act in three different roles, at most one in the role of a *master*, as well as any number with the role *slave* or *equal*. These roles are mainly used as a fallback mechanism for broken connections to a master controller or for load balancing and scalability. A switch is required to implement at least one flow table, representing the FIB or RIB of an OpenFlow device and the start of the OpenFlow packet processing pipeline. Flow tables contain flow entries that identify specific streams of packets through the switch. Flow entries are ordered by priority for matching against packets. They contain instructions, flow entry timeouts and match fields. Instructions allow for rate limiting through meters, adding metadata and forwarding to subsequent tables or groups and adding or performing actions to alter, output or drop packets. Groups are addressable by multiple flow entries from any table and facilitate the aggregation of forwarding mechanisms, e.g. multicast. OpenFlow packets are data structures derived from incoming network packets. The contained packet headers are compared to a flow entries match fields, e.g. source and destination addresses or ports. Queue structures are related to ports and provide for simple scheduling of packet output for quality of service (QoS) operations. Counters record statistics for data traversal through any of the mentioned structures. They are implemented differently based on ports, flow entries, groups, meters and queues. OpenFlow mandates a mapping to standard ethernet packets and consistency with processing them in ports, flow tables and groups. For example, hardware that implements a mechanism for loop detection outside of OpenFlow must flag ports for the underlying network interfaces to prevent their use for SDN processing and possible instabilities. OpenFlow describes messages exchanged between the control and infrastructure layer, which will always include a unique datapath-id of a switch. Asynchronous messages are unsolicited, subscribed messages by a controller and synchronous messages follow a request-reply pattern. Messages for establishing and maintaining the relationship between controller and switch are always synchronous. After establishing a TCP connection, the controller and switch perform a handshake to negotiate an OpenFlow version. The controller then queries the switch for its capabilities in order to determine the statistics a switch communicates and how IP fragmentation is handled. The switch capabilities also indicate if optional structures of the OpenFlow [ONFb] specification are implemented. Furthermore, controllers can update their role within a switch, which changes the set of messages they receive. Topology discovery is implemented by handshake and echo messages, as well as asynchronous port statistics to register links. They are sent from a switch to the controller as shown in Figure 2.2. The most relevant messages are depicted on the boundary of control and infrastructure layer. Message delivery is guaranteed, unless the connection enters a failed state and switches process packets according to existing flow entries. Hybrid switches containing capabilities outside of OpenFlow can optionally proceed by updating tables independently, although this entails reconciliation of state once a controller reconnects. A switch must process every message from controllers and is required to send update messages for changes on ports and flow entries, to ensure that controllers maintain a consistent state. In addition, controllers can request aggregate or individual statistics from a switch or flush all flow entries after reestablishing a connection. The use of queues and meters is encouraged to provide minimal protection against denial of service attacks and overburdening the processing pipeline in a controller. Message and packet processing is not ordered and subject to QoS operations.
Packet processing in a switch is triggered by an incoming network packet through one of its ingress ports or the arrival of a *packet-out message* on a controller channel. An incoming packet is inserted to the matching pipeline. Along with the ID and metadata of its ingress

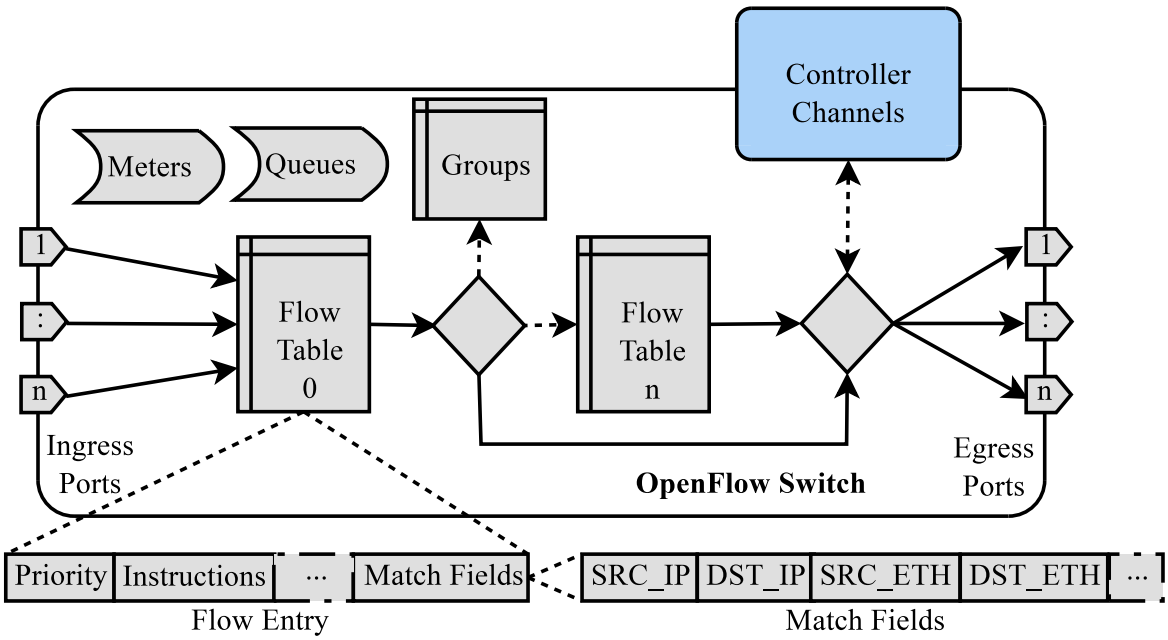

Figure 2.3: Depiction of an OpenFlow switch device including hardware and software components. The FIB of the device is implemented in flow tables that consist of flow entries. Flow entries contain match fields for packet headers. Meters and queues are implemented to facilitate quality of service (QoS) operations. Network packets enter and leave the switch through ports, logical or physical mappings to network interfaces. Controller channels connect the switch to at least one SDN controller.

port, the contents of packet headers are compared to the match fields of flow entries in the first flow table. A flow entries match as shown in Figure 2.3 includes header match fields that are validated on a packet and pipeline match fields for processing in subsequent tables. For multiple matching entries, the priority of the flow entry decides which is applied. In case of a match, the flow entry's instructions are executed. Instructions can pass packets to a table or group, apply meters, add pipeline metadata and update, apply or remove an action set. The instruction set is implemented to fit the needs of a switches processing pipeline with a specific order of execution. Actions change packet headers, send data to queues, push and pop tags to a packet and output packets to egress ports. An empty action set implies that a packet will be dropped without further processing. Usually some table will contain a wildcard table miss entry to invoke the controller channel, otherwise a packet is dropped directly. If a table miss entry is hit or an action leads to the output on a controller port, a *packet-in* message is generated in the switch. The message includes the match fields as well as data on the flow table, a packet-id to identify the packet in a switches buffer and the cookie data of the flow entry as shown in Figure 2.3. Cookie data is irrelevant for pipeline processing and a method for controllers to insert and read additional information for managing flows. A controller should react to a *packet-in* message by replying with either a *packet-out* or *modify-flow* message or both. The *packet-out* includes the buffer-id of the packet, which may or may not still exist in the switch, and an action set. The action set can again lead to reinsertion into the matching pipeline of a switch. Table entries are deleted,

modified and added by a controller through *modify-flow* messages that include cookies and timeouts for the entry. The switch will update respective tables or replies with an error message if it does not support necessary functions.

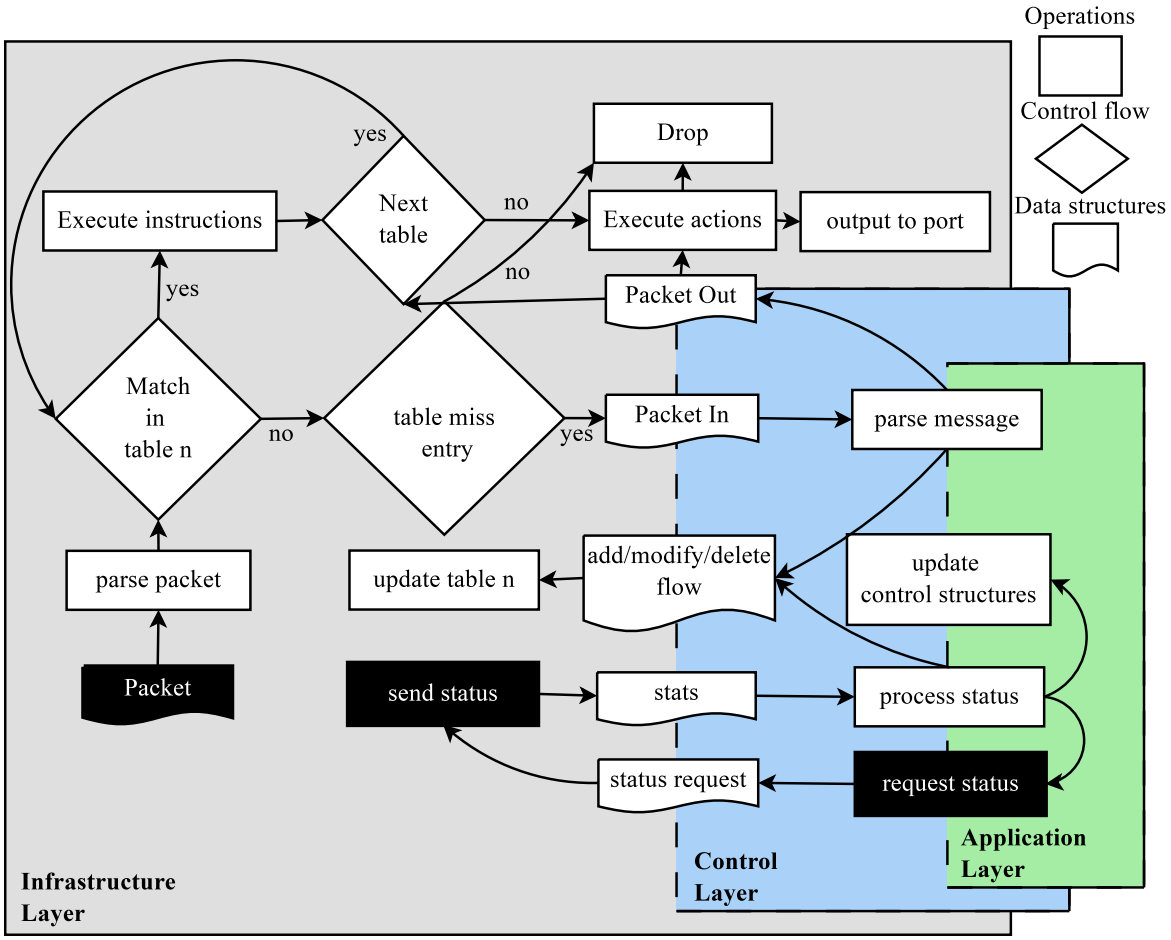

Figure 2.4: Flowchart depicting packet handling in an OpenFlow switch and the communication of its statistics through the OpenFlow [ONFb] protocol. Exchanged OpenFlow messages are shown on the boundary of infrastructure and control layer, control processes are shown on the boundary of control and application layer. The figure depicts an excerpt of the OpenFlow [ONFb] protocol, relevant to this thesis.

2.3 Conflicts in Software Defined Networks

The existing work on conflicts in SDN is based on three general paradigms, namely network management, the intent concerning applications or adversaries and data plane state. Most approaches outline a concept for conflict detection and resolution, complemented with a tool as proof-of-concept. Dwaraki et al. [DSNW15] argue that exposing application intents based on authorization, recording of state changes and resolution of conflicts, as well as management of related metadata is necessary for reliable SDN operation. This entails in-

sights into data plane state to determine application intent and also requires applications to consider state management outside of their own scope. Although the suggestion contradicts the programmable and dynamic nature of SDN, it implies that intents can only be deducted from the mentioned requirements, which in turn is necessary to interpret conflicts and sound resolution strategies. Below is an examination of current approaches for detection and resolution in light of the requirements of Dwaraki et al. [DSNW15]. The existing classification of conflicts is presented as a basis for the current work and to illustrate where it might be lacking.

2.3.1 Conflict Classification

Hamed et al. [HA06] provide the first extensive classification of conflicts based on the interpretation of forwarding plane state. The context are firewall rule conflicts, but as shown by Pisharody et al. [PCH16], the classification can be directly transferred to SDN, and the scope is not limited to security applications. The important distinction is that while Hamed et al. [HA06] observe and evaluate policies that are administered manually by administrators, the later work deals with conflicts of dynamic SDN environments. Where in SDN the rule priority determines precedence, for conventional firewalls, the same effect is given by rule order in the RIB and FIB data structures. Access-list conflicts and map-list conflicts are the two main conflict categories, as defined by Hamed et al. [HA06]. Access-list conflicts emerge between filtering rules and entail malfunctioning traffic flow. Map-list conflicts result from an inconsistent order or superposition of security transformations on packets, such as encryption and encapsulation. Weak protection, unnecessary overhead or clear text transmission of data is the consequence. While map-list conflicts are application bound and usually arise due to configuration errors, access-list conflicts occur between any SDN application and despite correct configuration. Another important definition is the difference between intra- and inter-policy conflicts. The first are based on comparing rules within one node in the topology. The latter are given by rule conflicts between several nodes and are of most interest for this work.
First of all, the set of relations between policies is defined based on pairs of related policies..Two rules exactly match if all of their attributes are equal. In terms of SDN this implies equal match fields. For intra-policy conflicts, the priority is the main distinction between two matching rules. A correlation between rules as defined by Hamed et al. [HA06] is most easily conceived by considering the rules with indices three in Table 2.1. The address spaces overlap, but none is a superset of the other. An inclusive match is given if one rule is a subset in at least one match field and at most equal in all other fields compared to a second rule. Inclusive matches are shown independently by the rules with indices one and two in Table 2.1. Completely disjoint rules satisfy none of the above criteria and partially disjoint rules are given by a completely disjoint rule with a relation in a match field that does not constitute an ordering, such as a TCP socket port. The previous rule relations where defined on the address space of rules only. When taking the priority and the action of a policy into account, the conflict patterns become apparent.
For inter-policy conflicts the patterns comprise Redundancy, Correlation, Shadowing and Exception. As already mentioned, conventional RIB policies are oblivious of rule priorities, but since the rule ordering implies a priority, the concept will be applied in the following explanation. A higher priority implies a precedence in rule application. Redundancy is given in case of an inclusive or exact match and equal actions, illustrated by rules with index one

in Table 2.1 if A = B. While Redundancy does not pose a issue for traffic flow, an excess of redundant rules can impede performance. The rules with indices 3 show a Correlation conflict if A $\neq$ B, i.e. the address space is a correlated match and the actions differ. A Correlation conflict is interpretative, due to the undetermined effect of the correlated match on different traffic profiles. Shadowing is given in case of an inclusive or exact match and differing actions, shown by rules with index one in Table 2.1 if $i > j$ and A $\neq$ B. This can lead to security issues where filtering policies are not applied. Exceptions can be intended conflicts for fine grained control of traffic flow. An Exception is given by rules with indices two if $i > j$ and A $\neq$ B or for rules with indices one if $j > i$ and A $\neq$ B.

Index	Priority	Source address	Dest. address	Action
1	i	10.0.1.*	10.0.1.*	A
1	j	10.0.1.*	10.0.1.1	B
2	i	10.0.1.1	10.0.1.2	A
2	j	10.0.1.*	10.0.1.*	B
3	i	10.0.1.*	10.0.2.2	A
3	j	10.0.1.1	10.0.2.*	B

Table 2.1: Illustration of address space relations and rule conflicts as defined by Hamed et al. [HA06]. Rules with index one and two show an inclusive match, while index three is a correlated match. The choice for priorities and actions constitute the conflict patterns Redundancy, Shadowing, Exception and Correlation.

When considering inter-policy conflicts, there are only two main types, namely Shadowing and Spuriousness, and the definition and effects are not as specific. Shadowing corresponds to the definition for intra-policy conflicts, although with an added effect. The shadowed rule on the corresponding device never gets applied and traffic enters the network that will ultimately never reach it's expected destination. Complete Shadowing is given when the address spaces exactly match and partial Shadowing results in functional traffic for some cases due to an inclusive match. Spuriousness is linked to a security context and occurs if filtering polices are applied too late. With an exact match, complete Spuriousness is given and partial Spuriousness for an inclusive match. If a firewall is placed at a intermediate position within a topology, spurious traffic is never meant to reach a destination but still enters part of the internal network. As stated by Hamed et al. [HA06] this creates vulnerabilities to network reconnaissance and intrusion.
Pisharody et al. [PCH16] reached a classification in context of SDN and based on the flow table state within one node in the infrastructure layer. The criteria for the classification is again given by comparing rule priorities as well as their address spaces and actions. Since the mentioned patterns Generalization, Overlap and Imbrication are simply a new convention for the before mentioned patterns Exception, Correlation and partial disjoint matches, the novel contribution is the distinction between the possible choices for the priority. Ultimately it is arguable, especially when considering OpenFlow as the SDN framework, if

the definition of patterns with equal priority is sensible at all since the processing pipeline behavior is not specified in that case and implementation dependant, e.g. by favoring the older policy. Therefore we will look at the definition for ordered priorities only. The two new conflict patterns are Generalization and Overlap. An Overlap as stated by Pisharody et al. [PCH16] is a specialised form of Correlation where actions A = B. When looking at rules with indices three, an Overlap is given for $i < j$ or $i > j$. A Generalization conflict is in effect the same as a Shadowing conflict. The effect is that a rule with lesser precedence will not be applied and the address space of the later rule is a subset of the first rule. A Generalization conflict would be given by rules with indices one, where $i > j$ and A $\neq$ B and rules with indices two, where $j > i$ and a also A $\neq$ B. Pisharody et al. [PCH16] assign the conflict patterns to two main categorizes that indicate the resolution strategies. Shadowing, Redundancy and Overlaps are *intelligible* conflicts that can be resolved so that all resulting rules cover the interest of the conflicting rules. Generalization, Imbrication and Correlation are *interpretative* conflict types that require adaption depending on the given environment, and resolution strategies might lead to subsequent violations and inconsistent data plane states.
In order to detect and define distributed conflicts, we need to correlate rules between switches in a network topology to form a path from a traffic source to it's destination. Therefore match overlaps are not part of distributed conflict classes but the basis for discovering them. Although we need to compare policies across data plane entities to build such paths, the mechanism is the same as with finding contending match fields for local conflicts. Furthermore, actions are key to defining a distributed conflict class. With local conflicts the conflict patterns depend on action equivalence, in a distributed scenario we will have to examine the accumulated effect of all involved actions. Accumulated effects of actions across switches concerns the described Multi-Transform and Spuriousness classes by Hamed et al. [HA06]. We expect to encounter these in the experiments. Considering the definition of inter-policy, or distributed conflicts, the priority is irrelevant, which serves as the first insight for the approach of this work and the relevant criteria for distributed conflicts. Nevertheless, the precedence of policies decides which rules across network nodes connect, implying a direct effect of local conflict resolution on any distributed conflicts. The similarities between local and distributed conflicts are given by how candidates are generated through match correlation and the fact that actions dominate the effects of each conflict type. Spuriousness and Multi-Transform conflicts form the foundation of a distributed conflict data base to build upon. The experiments show that faulty configuration can be the reason for other conflict classes too. As with local resolution, resolution in distributed cases can evoke new conflicts that need to be resolved in an iterative manner. The concept of intelligible and interpretative conflicts [PCH16] when applied to distributed conflicts has two implications that are different to local conflicts. Firstly, the existence and negative effects of some distributed conflicts can only be determined by monitoring endpoints, and others require the input of application intent. This seems infeasible when applied in a productive environment. Secondly, the resolution strategy for some of the presented conflict classes will always be interpretative and dependent on manual resolution.

2.3.2 Conflict Domains

A popular topic for SDN conflicts are security applications, concerning attack vectors, misconfiguration and overlapping responsibilities. Covert channels, established by exploiting

Authors	Scope	Criteria	Detection	Resolution	Tool
Shin et al. [SPY+13]	Security applications	Threat models	✓	✓	FRESCO
Porras et al. [PSY+12]	Security applications	Application intent	✓	✓	FortNox
Li et al. [LCL+18]	Covert channel attacks	Threat models	✓	✓	CCD
Lu et al. [LFX+19]	Multi Controller	Traffic monitoring	✓	X	-
Khurshid et al. [KZCG12]	All conflicts	Static invariants	✓	✓	VeriFlow
Sun et al. [SMR+14]	Resource conflicts	Static invariants	✓	✓	Statesman
AuYoung et al. [AMB+14]	Resource conflicts	Network state	✓	✓	Athens
Pisharody et al. [PCH16] [PNC+19]	Local conflicts	Data plane state	✓	✓	Brew

Table 2.2: Overview of research on SDN conflicts and corresponding tools.

conflicts between applications, are researched by Li et al. [LCL+18]. The authors define threat models and their attack vectors that exploit filtering applications in SDN. Conflict patterns are derived from the threat models, and a means for active prevention by inserting automatically generated policies to ensure a correct data plane state or blocking them entirely is presented. The detection mechanism is based on comparing candidate flow entries to the existing data plane state. Conflict resolution is automated and no feedback is provided for interpretation by an operator. The authors introduce a novel data structure for efficient comparison of flow entries through equivalence classes, which results in an overhead of 18.7% for processing packet-in messages [LCL+18]. The resulting prototype is the Covert Channel Detector.

Another approach is given by Porras et al. [PSY+12]which additionally considers flow entry priorities and defines three authorization levels. Manually installed rules by administrator are given the highest priority, followed by security applications and ending with other applications in the third tier. For matching fields in policies that intersect, redundant and malformed ones are ignored automatically. Resolution of all other rule intersections and policies with the same authorization level are presented to an administrator for manual inspection. With FortNox Porras et al. [PSY+12] present a mechanism that prevents rule insertions that affect performance but addresses the need for interpretation for other conflicts and the ambiguity of automated resolution. Invariants and authorization levels can be defined and are upheld by the system. Porras et al. [PSY+12] implemented the prototype by detecting conflicts by rule comparisons but do not provide explicit definition of conflict patterns. The initial effort by Porras et al. [PSY+12] was further persued by the work of Shin et al. [SPY+13] and resulted in the tool Fresco. The detection and resolution mechanism is the same, but the focus is put on debugging and development of security applications. Fresco adds a layer of configurable security modules on the security enforcement kernel of FortNox.

As stated in Section 2, an SDN environment with multiple controllers is feasible, although prone to issues due to controllers that cannot reconcile responsibilities. Lu et al. [LFX+19] investigate conflicts that arise due to multiple controllers in a topology and discern between weak conflicts that merely handicap traffic flow and strong conflicts that prevent traffic flow entirely. Again this work relies on theoretically designed patterns based on flow rules, and in effect shows that the scope of the two conflict patterns capture their general repercussions, but no tangible interpretation. Furthermore, the benefits of deploying multiple controllers of a lesser load for single instances is replaced by the new bottle-neck of the proposed shim layer between control and data plane to coordinate the controllers.
Sound network state also depends on network resources, such as available forwarding paths. A network management model is formulated by Sun et al. [SMR+14], which mediates interests between resource control applications. The model captures the state at three phases: the currently observed state, a new state as proposed by an application and a target state that emerges from safely consolidating all proposed states. In Statesman the two invariants that are the basis for determining a target state are reachability and network capacity. The authors [SMR+14] introduce the concept of *Controllability*. Resource control applications are supposed to work on information that is relevant to their functionality only but cannot propose states that can be consolidated without any knowledge on their interdependence to other applications. Controllability is a metric computed in the control layer based on all application invariants and communicated to applications in context of the network invariants they influence.
The work by AuYoung et al. [AMB+14] apply the concepts of Statesman in a SDN context and suggest a voting scheme based on the proposed network states. The framework involves an SDN controller and a separate resource controller. The introduced resource controllers possibly demand for conflicting network invariants such as high or low bandwidth or additional nodes and links. Consequently, this model aims at keeping infrastructure consistent with invariants. Voting on proposals from different modules, e.g. a switching, fault tolerance, or bandwidth module results in choosing the proposed state with highest implied benefit. The authors state that the value of a proposition is hard to quantify and differs across modules [AMB+14]. This might result in an inferior target state and impaired safety from the standpoint of some applications. AuYoung et al. [AMB+14] developed Athens to resolve SDN as well as resource conflicts by automated comparison of proposed network states and a voting mechanism. The intention of control and resource applications and the individual value of proposed changes is subject to interpretation. Two key concepts are introduced, *Parity* and *Precision*. Precision implies if an application can accurately determine the value and impact of its own proposals. Parity denotes if applications are able to rate their proposals on a commonly known metric. Depending on the given combination of precisions and the level of parity, different voting mechanism are viable and an administrator needs to interpret application effects to e.g. introduce weights to influence the impact of applications in the voting process.
Examination of data plane state and their alignment with defined invariants is implemented in VeriFlow by Khurshid et al.[KZCG12]. The framework is agnostic to applications and ignores temporary network state, i.e. only fully processed packets, and inserted rules are relevant. The approach is based on monitoring state changes and generating forwarding paths for updates to determine the entities in the network that will be affected. This decreases the regions of data plane state that need to be considered.
Brew is a tool implemented by Pisharody et al. [PNC+19] for conflict detection and resolu-

tion in SDN environments. It is agnostic to topology characteristics and handles intra-policy conflicts only. It provides automated resolution of intelligible conflicts and a visualization for interpreting and resolving interpretative conflicts manually.
There are again several key implications that can be expanded to distributed conflicts. Conflict resolution in most projects is either done by heuristics, e.g. voting or approximation of functional network state, by definition of priority levels of applications on top of rule priorities, or simply by handling intelligible conflicts and presenting interpretative conflicts. Any heuristics imply unpredictable consequences for network security. Since distributed conflicts other than loops produce interpretative conflict classes, resolution as of now appears arbitrary at best and in the worst case more harmful than the conflict itself. Tran and Danciu [TD20] emphasize the need for discovering more conflict classes and the need of application intent to collect data that can serve as a basis for developing sound resolution strategies. The existing tools show that added latency due to detection and resolution can be optimized but is inevitable. While the trie data structure used in several implementations is an example for successful optimization, it is based on layer 3 destination addresses only. Adding bi-directional addresses or addresses from additional layers defeats the optimization. Another difference is that detection of distributed conflicts can only be performed once a newly generated policy is connected to a subsequent policy. The subsequent policy comes into existence only if the current is accepted. Any form of resolution that alters the predecessor policy can prevent the emergence of the subsequent policy.

2.4 Security Requirements in Software Defined Networks

While several tools listed in Table 2.2 address security issues in SDN [SPY+13] [PSY+12] [LCL+18] [PNC+19], the criteria and motivations for detection and resolution as well as concepts of security vary. The only work on SDN firewalls exclusively, therefore excluded from the overview on general tools, is given by Hu et al. [HHAZ14] and the author's tool FlowGuard. FlowGuard requires a set of firewall rules as input, which are enforced globally in the entire network. Another limitation is the requirement that typical optimization strategies, based on the order of firewall rules, the correlation of targeted packets and differing actions, are exempt. No policy that allows access is allowed to overlap with target packets of a blocking policy. All named tools enforce their security policies across the entire network and operate as an intermediate layer between controller and applications, or between controller and data plane. Dixit et al. [DKZ+18] review filtering applications in SDN, with a focus on FlowGuard, and define several characteristics firewalls must have in SDN. This includes centralized flow tracking, i.e. maintenance of connections between flow entries across devices, centralized policy enforcement across the entire network, tracking of connection states, automated conflict resolution, robustness against concurrent policy insertion and the need for scalability. The notion of a traditional firewall is not applicable to these requirements, and the tools, including FlowGuard, represent a security enforcement layer that prevent control applications from inserting violating policies or aim at resolving conflicts. Conventional firewalls usually enforce policies within one device and are unaware of other applications and the complete network state. There are three distinguishable circumstances that lead to security conflicts. Packet transformation related conflicts that circumvent firewalls [HA06] [LCL+18] [HHAZ14] [PSY+12], role-based scenarios [PSY+12] and the deployment of malign applications or policies [HHAZ14], and the location of conflicting policies [HA06] [HHAZ14].

Transformation of packet headers disguise packets from filtering rules targeting specific protocols such as the Internet Control Message Protocol (ICMP) used for network diagnostics, or layer 2 and layer 3 address ranges. Authorization roles provide an extra layer of information outside of the control channel to verify the origins of deployed policies and can be implemented by digital signatures. Lastly, the deployment of applications within a network for network intrusion detection, traffic quarantining, filtering and traffic shaping defines what partitions of a topology can be reached by the affected traffic, before it is processed by said applications. We will utilize these notions on security contexts and requirements for security applications to evaluate new conflict classes and the presented detection mechanism.

2.5 Hidden Conflicts

Hidden conflicts in SDN as defined by Tran and Danciu [TD19] occur if a control application fails to receive a packet-in event with flow criteria that would trigger a decision by the application. The reason is the deployment at a time after another application has already enforced a policy based on such a packet-in event. This is a direct consequence of the OpenFlow [ONFb]standard, which specifies that packets handled by an existing flow entry in an switch conforming to the protocol refrain from generating another message to optimize bandwidth on the control channel. As a consequence, data plane state needs to be cleared whenever a new application enters the decision making process, which counteracts the optimization efforts and entails guaranteed downtime in the network. Tran and Danciu present an approach based on extraction and consideration of application intent to generate artificial packet-in messages and prevent the mentioned shortcomings [TD19]. While application intent can be undecidable and open for interpretation, the work on Hidden Conflicts [TD19] demonstrates it's importance for local conflicts in SDN. The results in Chapter 5 show that an overlap in application intent can be a primary cause for distributed conflicts, where a similar approach is necessary for detection. Tran [Tra22] implements a rule graph based on correlations between flow entries that fulfills a fundamental role for detection of distributed conflicts. The current implementation, as well as adaptions and extensions from this work are described in Chapter 4.

2.6 Experiment Parameters for Software Defined Networks

The conflicts in the forwarding plane are the combined symptoms of several underlying causes. Tran and Danciu [TD20] define a parameter space that encompasses the dimensions that lead to apparent conflicts in the data plane as well as hidden conflicts [TD19] for control applications interested in the same traffic flows. The parameter space shown in Figure 2.5 consists of three types of dimensions that relate to traffic characteristics, topology shapes and SDN configurations.
Data flowing into a network can arrive in a variable, bursty or constant bit rate. The traffic profile can be consistent over time or changeable and mixed. Data rate and its patterns can affect applications such as rate limiters. Transport protocols and a combination of such can directly affect SDN applications that enforce policies on certain match fields. Also control policies can cause problems if they fail to address certain transport types. Networks exert a specific endpoint combination depending on its use case. If fault tolerance or a balanced load in the network is a priority, the nodes and the established links between them can provide

alternative or multiple paths. This will result in one-to-many, many-to-many and one-to-one combinations. There are common, realistic shapes for network topologies, e.g. mesh or star topologies that can be considered. Synthetic networks are a mix of realistic shapes or are explicitly designed to produce conflicts unseen in realistic setups. Control applications or control functions of a SDN controller generate policies with a certain priority. The priority greatly influences which policies actually become active in handling the traffic flow. Applications working on a mixed set of priorities are prone to other conflicts than those that operate on the same priority. Security levels and optimization can be reasons for the choice of priorities. Control applications can be configurable with invariants, e.g. for quality of service settings, and might implement variable or conflicting configurations. Control functions in the controller are deployed at the same time, but the start order for applications working via an asynchronous interface through the north-bound API results in unpredictability due to the start order. Applications that get to insert flow entries first, deprive other applications interested in the same traffic of their influence [TD19]. Applications can target specific or all nodes in the infrastructure layer, depending on their function. The choice of target switches might imply overlaps in their responsibility.
This work introduces enhancements of the technical aspects devised by Tran [Tra22] and evaluates the parameter space dimensions in regard of any exposed distributed conflicts. The currently application dependent comparison of data plane state and application intents is generalized with a concept to include new control applications dynamically. Experiment inputs and outputs are abstracted from technical details and the existing implementations are adapted and fitted into a modular software architecture. Key dimensions are application and topology related. Consequently, we need to devise the missing approach to make the exploration of the topology dimension easier and incorporate random topologies. Additional control applications facilitate the expansion of application related dimensions. We will demonstrate the discovery of distributed conflicts and validate the methodology of Tran and Danciu [TD20].

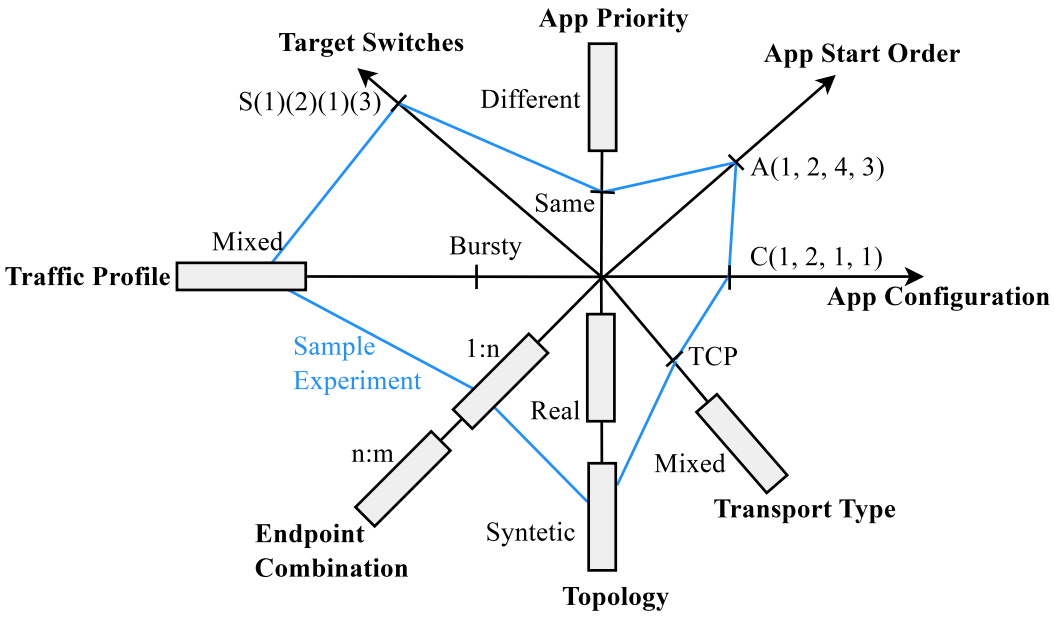

Figure 2.5: Parameter space for SDN environments defined by Tran and Danciu [TD20]. The dimensions relate to traffic, topology and SDN characteristics and serve as a data space for an empirical approach to examining conflicts. The figure is a simplified version of Figure 4 in [TD20] by Tran and Danciu.

3 Experimental Design

There is an infinite amount of data points in the parameter space due to an arbitrary number of nodes in a topology. The methodology for an iterative completion of the classification is illustrated in Figure 3.1. It comprises tasks at different levels of automation and is based on the methodology introduced by Tran and Danciu [TD20].
The selection of parameters as well as modeling a topology for the data point and building the technical infrastructure is the first step for conducting an experiment. A topology model is created manually, which serves as blueprint for generation of a virtual infrastructure and as reference point for future experiments. The deployment of the chosen control applications and simulation of network traffic follows. All control applications are run in isolation to determine their expected behavior. Once these isolated executions are complete, the entire power set of application combinations is executed to compare the observed behavior to the expected data plane state. The work by [Tra22] automates these steps, yet application input is processed individually and lacks a generalized approach. The developed prototype is incorporated to validate already seen conflict patterns in repeated or new experiments. Utilizing the detection prototype at this step helps reduce the amount of data that indicates already seen conflicts. Furthermore, validation of patterns is important to avoid the impact of singularities and to demonstrate the value of conflict detection in an empirical approach. Any remaining inconsistencies between expected and observed state usually require manual inspection and interpretation. By comparison of the observed data plane states and the expected as well as other notable inconsistencies, conflict patterns are deduced if present. The prototype provides a user interface to compare and analyze conflict classes. The figure intentionally lacks a terminating state, since the data space has an infinite complexity as mentioned above.

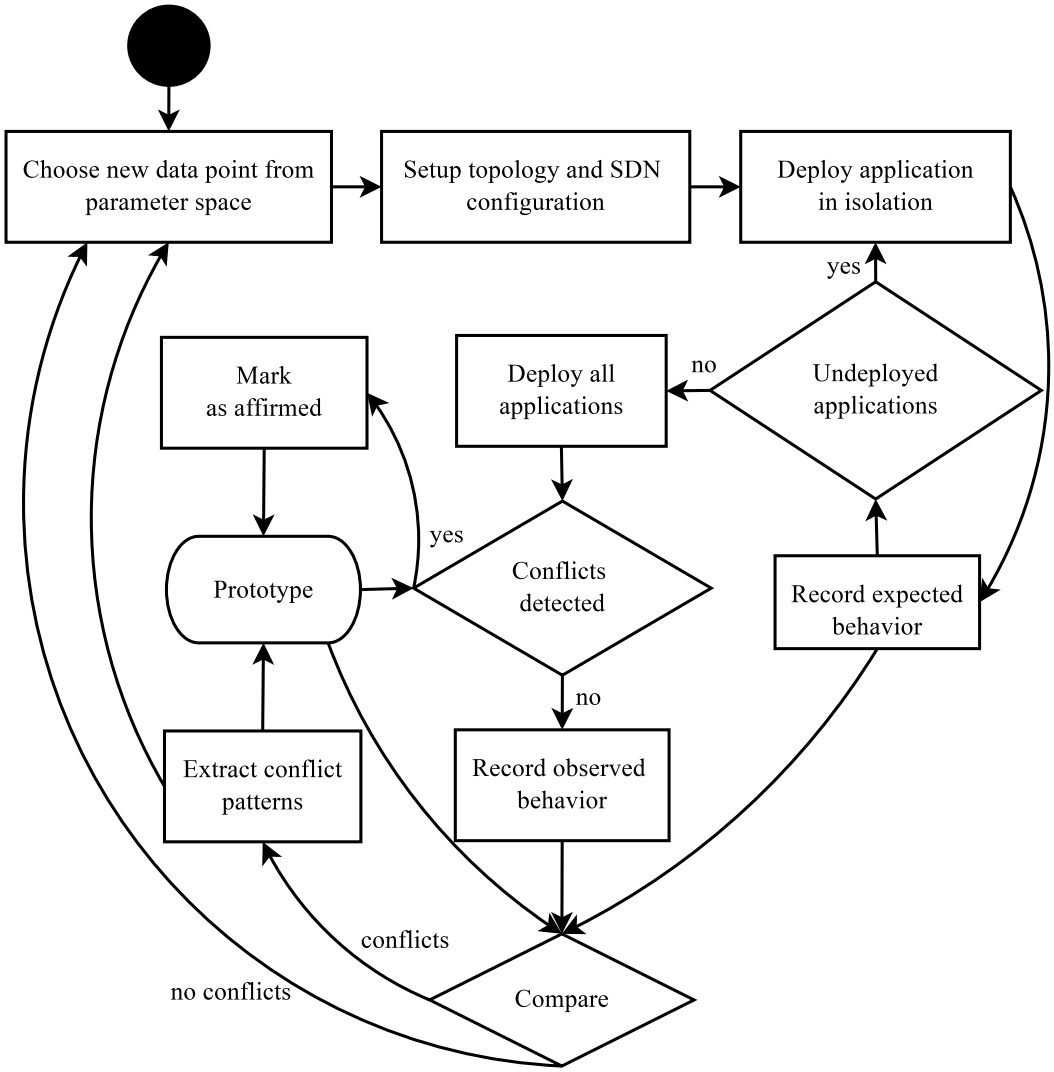

Figure 3.1: This figure depicts the major steps for discovering new conflict classes, topology setup, traffic simulation and conflict deduction, and is an adaption of the methodology of Tran and Danciu [TD19].

3.1 Network Topologies

In order to examine the presented parameter space, different categories of network topologies will be tested. One category will be intentionally designed networks and the other random, to avoid a bias towards certain shapes and configurations. These two categories do not correspond one to one to real or synthetic topologies as defined by Tran and Danciu [TD20]. Especially random instances can be both real or synthetic, and a deliberate design of a topology can also yield both types. The networks can be seen as graphs with the following properties.

Nodes are the computing and routing resources in the network. They can be SDN switches in the context of this thesis and are depicted as blue programmable switching devices. Hosts are the nodes that initiate or answer to network traffic in the experiments and are depicted as desktop-like icons. A nodes degree is given by the number of edges it connects to.

Edges are the connections between the nodes and represent virtual data links.

Degree distribution is a property of the entire network and signifies, the inter-connectivity of a topology. It is a metric of the shape and can imply a clustering in certain regions of the network.

Shapes are common structures in graphs that correspond to rings and stars as well as bus topologies. A ring topology will connect a node to its neighbours and the deficiency of a node will impair the connectivity of the entire network, although reachability is still given. A star topology exerts central hub nodes for connectivity between the edge nodes. While the deficiency of a single node is usually of low impact, a dysfunctional hub node will split the topology into its individual branches. A mesh topology in its most complex form exemplifies a fully connected graph, where all nodes are connected to each other. Mesh structures are most commonly utilized to keep the average distance between computing resources small and decrease latency, e.g. in computing centres. Large scale networks such as backbone infrastructures across research facilities usually exhibit a hybrid shape to provide redundant paths for resilience and a balanced load on the links, while exploiting the properties of more specific shapes in smaller clusters.

3.1.1 Designed Topologies

The following networks where designed for examining conflicts on the basis of realistic or existing topologies. A simple network design is devised for providing example topologies to explain and test the deployed control applications. The concepts are based on literature on SDN as well as publicly available resources.
The first and simplest topology is devised for testing. It includes four switches and four hosts, to provide for all topology characteristics that are needed for simply application testing and is still light-weight to build and deploy. The characteristics include multiple alternative paths between hosts, a loop in the topology and multiple hosts to test endpoint load-balancing scenarios. Figure 3.2 depicts the described topology.
The second topology depicted in Figure 3.3 is conceived as a simple abstraction of several use cases. It includes several hosts to initiate traffic, two endpoints that represent resources

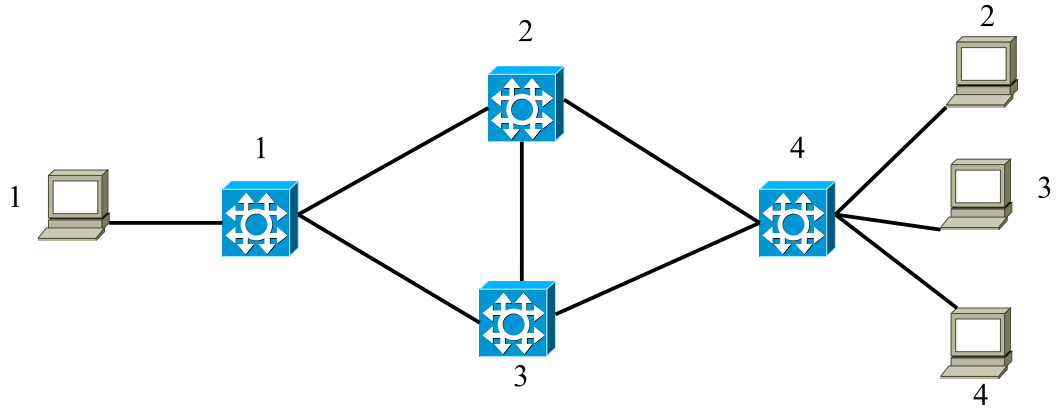

Figure 3.2: Simple network topology for application tests. Switches are illustrated by blue nodes and hosts by the desktop resembling icons. The lines between the nodes depict the established virtual connections between the nodes.

for web services and several tightly meshed computing resources with high availability and redundant paths to each other. It exerts a cluster of nodes in a mesh and a ring, unified by a bottleneck node that serves as the hub of a connecting star topology.

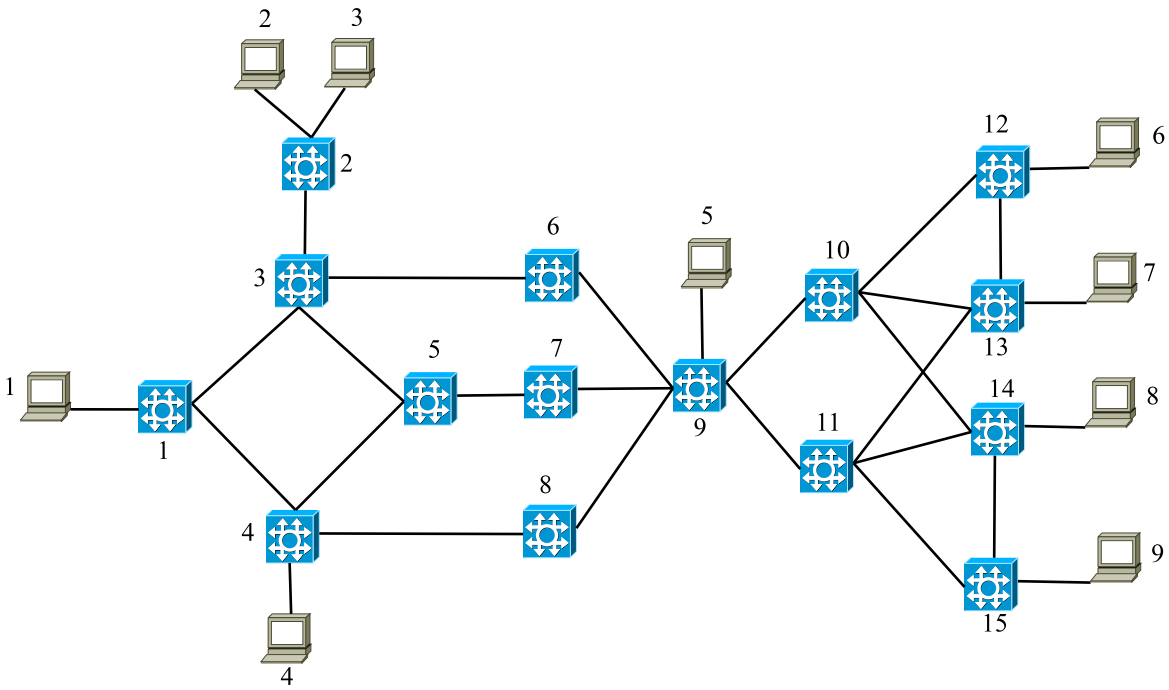

Figure 3.3: Network architecture feature a simple mesh and ring topology, connected by a bottleneck node and several redundant paths.

Asadujjaman et al. [ARAM18] devise the network illustrated in Figure 3.4 and use it for research on SDN controller resilience. The authors test for failure recovery of controller communication and connectivity and the network that contains core features that are of interest

such as loops, bottleneck nodes, parallel and fallback routes as well as a nested cluster in the core backbone. Most nodes have an average degree and three central switches display a high degree.

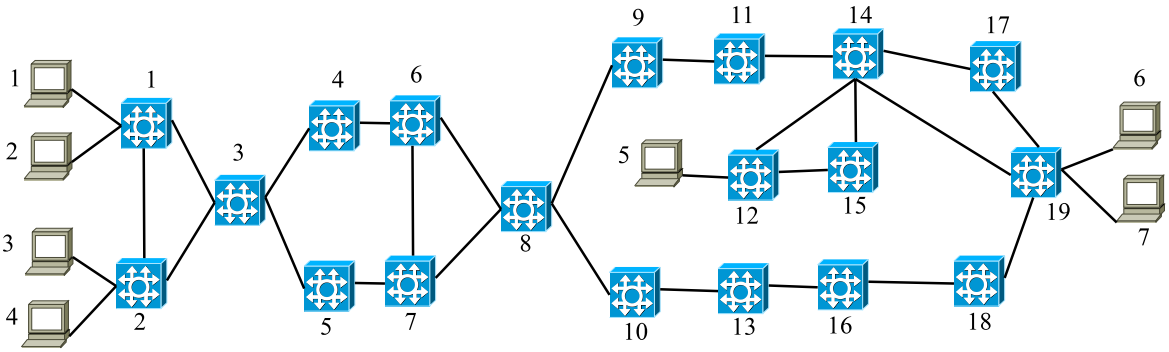

Figure 3.4: SDN topology used by Asadijjaman et al. [ARAM18] for research on failure recovery of the control channel.

The Münchner Wissenschaftsnetz (MWN) is a network between the three main universities and other research facilities in Munich. It connects all locations to the Leibniz-Rechenzentrum (LRZ) computing centre and provides access to the internet via two internet service provides. The topology in Figure 3.5 abstracts the backbone net of the Munich universities and research facilities and their inter-connectivity. The shape consists of redundant links that connect the individual sites and internet providers, as well as a star topology centralised around the campus next to the computing centre with a central node of very high degree.
The abstraction of the Nippon Telegraph and Telephone backbone, illustrated in Figure 3.6, is the network of one of Japans main internet service providers listed in the Nikkei index. It serves as a test topology for various SDN implementations on the Ryu controller framework [Nip]. The network consists of 55 switches and is the largest topology tested in this work. Its shape corresponds to the geographical outlines of the island state and consists of connected ring topologies in various sizes. The hosts are placed for simulating long range connections across the island. Most nodes have average degree, with the exception of the central connecting switches.
Pisharody et al. [PNC+19] use the topology depicted in Figure 3.7 for scalability testing of their work. It simulates the backbone switching structure of a computing centre. Hosts are placed below the access level switches to simulate traffic between computing resources. Switches in the backbone layer exert the highest degree at the core of two connected star topologies. Their child nodes in the distribution layer are part of a hybrid shape which yields a network tree. The distribution and access layer form a mesh with nodes of lower degrees.

3.1.2 Random Topologies

To avoid a bias towards conflicts that arise in constructed scenarios and to validate conflicts that arise in designed settings, random topologies are used as testbeds. There are several undesirable properties for the testbeds that randomly generated graphs can possess, which include unconnected nodes or simple and trivial shapes such as a pure bus topology. Therefore it is paramount to choose suitable algorithms for graph generation that address these

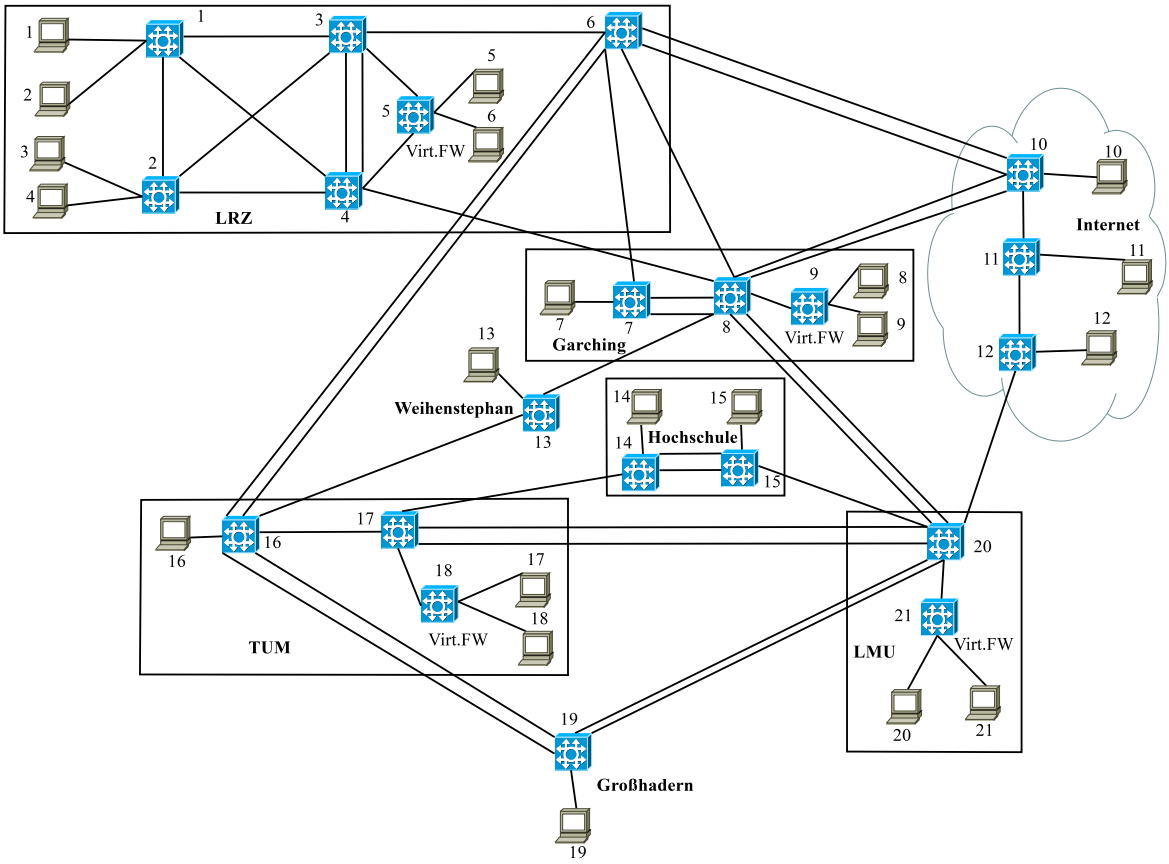

Figure 3.5: The MWN network connects multiple university campuses, several research facilities and the LRZ computing centre.

constraints. The following algorithms differ in their properties for exhibiting realistic shapes and degree distributions as well as their characteristic for ensuring connectivity.
The choice of the Erdős–Rényi model for the first set of random graphs is due to the three main criteria the model is based on, i.e. the property that edges are generated independently edges and with equal probability and the study on the connectivity of graphs of such graphs. Erdős and Rényi [ER60] as well as Gilbert [Gil59] independently devised the *G(n,p)* model and described its properties. While n denotes the number of nodes, p is the probability for creating an edge between two nodes. The graph is constructed by connecting labeled nodes randomly in a process that corresponds to percolation theory, with a binomial degree distribution of the nodes. The most important property discovered by Erdős and Rényi [ER60] is that if $p > (1 - \epsilon)ln(n)/n$, graphs will usually be connected. In other terms, there is a critical percolation threshold for maintaining a robust connected graph with $p_c = 1/ < k >$, where $< k >$ is the average degree of a node [ER60]. This yields connected networks without independent components and as the network grows, depending on the choice of p and the size of n, moderate clustering and leaf nodes become more probable. A sample graph is depicted in Figure 3.8.
A study of graphs that relate to realistic applications is conducted by Newman et al. [NSW01] and explicitly contains a comparison to the structure of the World Wide Web. The graphs

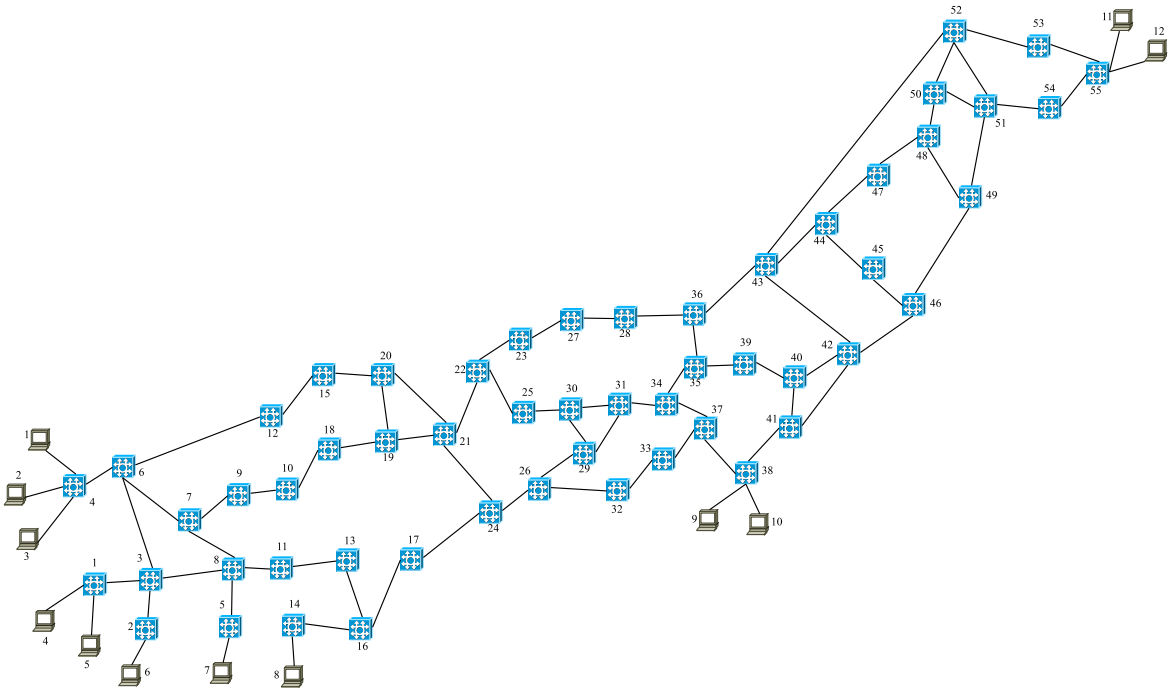

Figure 3.6: Illustration of the core backbone of the Nippon Telegraph and Telephone network in Japan. Host nodes are inserted for experiments on distributed conflicts.

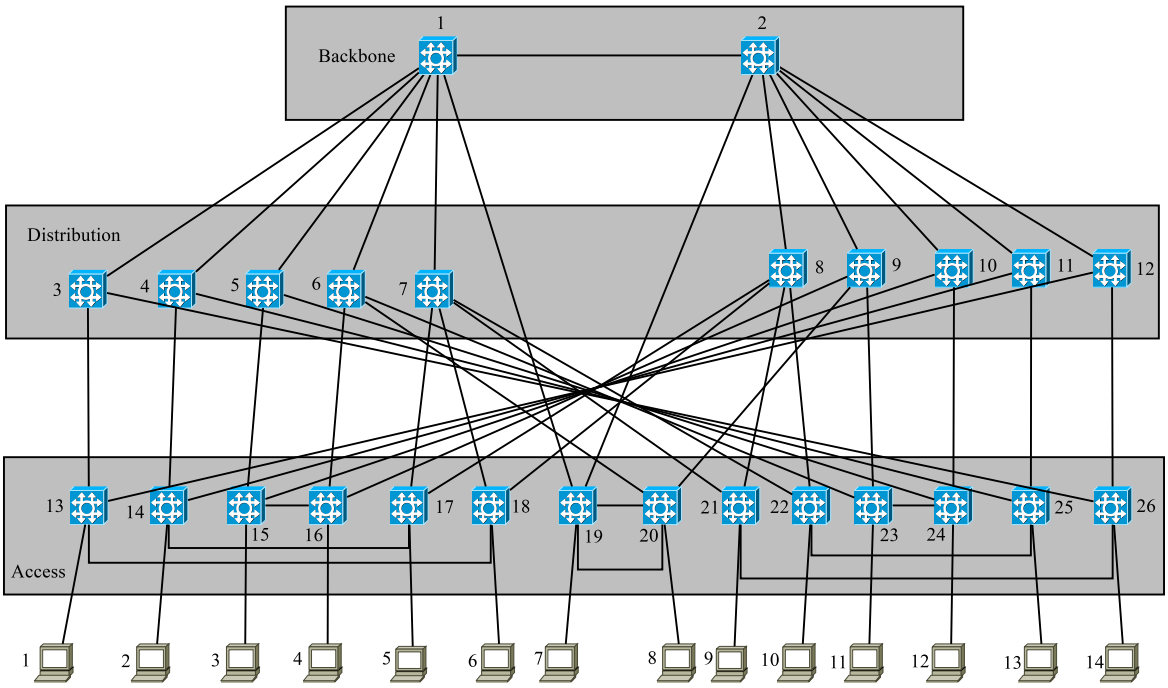

Figure 3.7: This figure approaches the network design of a computing centre backbone and is used by Pisharody et al. [PNC+19] for scalability testing of their SDN application. The figure is an adaption of a depiction by Pisharody et al. [PNC+19].

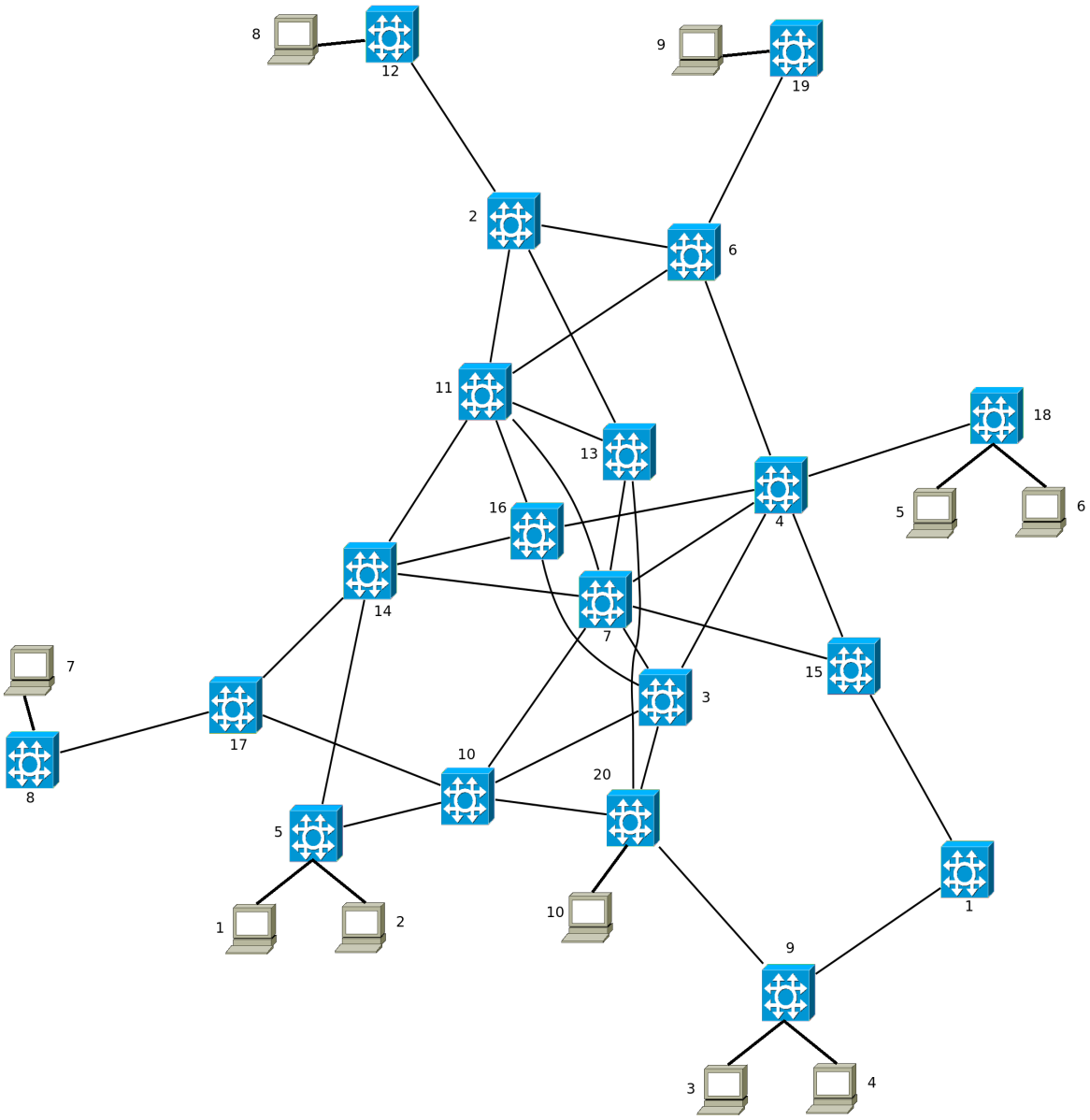

Figure 3.8: Random graph generated based on the Erdős–Rényi model [ER60], with binomial degree distribution and moderate clustering.

generated with the Newman-Watts-Strogatz algorithm tend towards a collection of circular, connected ring topologies. The network shape is constructed based on principles that relate to triadic closures mentioned first by Georg Simmel [Sim09]. The graphs show a high clustering coefficient and short average path lengths. The defining parameters are the number of nodes n, the number k of nearest neighbors for a node and the probability p for creating new edges. The initial step is to create a ring shape with n nodes. Then the nodes are connected to k nearest neighbors and finally for each existing edge between two nodes *(u,v)*, a new edge *(u,w)* to a third node is generated with probability p. As depicted in Figure 3.9, by choosing $k > 1$ we can avoid pure ring topology, while keeping p and the number of random edges low, independent of the size n. The basis is still a graph that relies on properties of the Erdős–Rényi model, while displaying small world properties often seen in realistic settings.

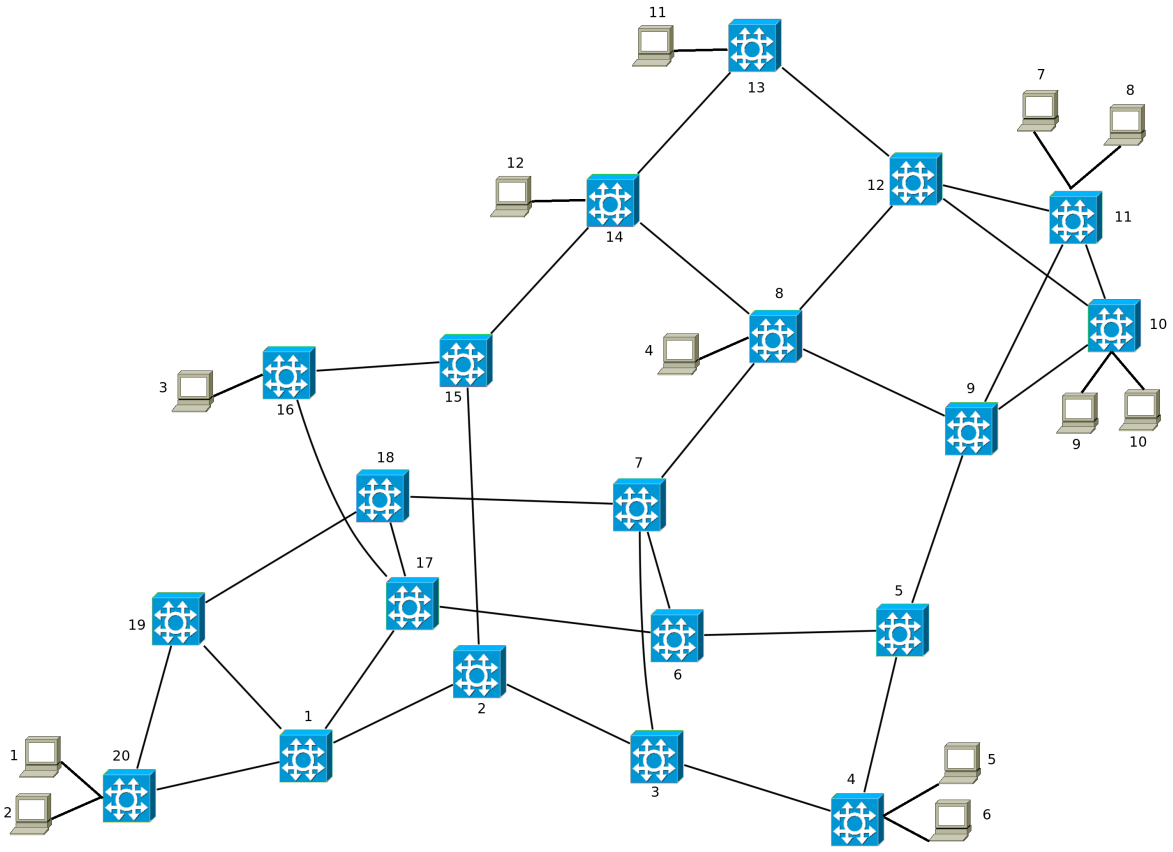

Figure 3.9: Random graph generated with the Newman-Watts-Strogatz algorithm [NSW01] that displays small world properties with short average paths and a tendency towards clustering.

The concept of scale-free networks that possess a power-law distribution concerning node degrees was the result of a study by Barabasi and Albert [BA99]. Social networks like the internet infrastructure are supposedly subject to self-organising mechanisms that yield power-law distributions and densely connected hub nodes. While a study by Amral et al. [ASBS00] shows that networks rarely show a true power-law distribution, the Barabasi-Albert model is accepted as a good approximation of realistic topologies. Two natural mechanisms in

the evolution of networks are growth and preferential attachment and Albert and Barabasi [AB01] argue at a later point that both are necessary to attain small-world networks with scale-free properties. One element of study are the dynamics behind citation networks and their growth as well as preferential attachment to central studies. The concept can be conceived by putting this thesis into the context. A new thesis, i.e. a new node, enters the citation network and since the work by Barabasi and Albert [BA99] is a central and widely applied concept in generating scale-free networks, it attracts new nodes more easily than other references. The Extended-Barabasi-Albert algorithm has four parameters. n is the number of nodes, m is a fixed number of edges that are randomly added with probability p, and q is the probability that m existing edges are rewired. Consequently the algorithm allows for growth and reconnecting, both on the basis of preferential attachment, i.e. more connected nodes will receive more reconnected edges or new edges and evolve to become hubs. Interestingly, the critical percolation threshold is $p_c = 0$ for large n, since the removal of a central node usually leads to disconnected islands in the graphs and degeneration to multiple components, although the removal of most nodes does not affect network stability. Figure 3.10 illustrates a graph based on the Barabasi-Albert model [BA99], with an obvious high edge degree and clustering around a small number of hub nodes.
The caveat of the described algorithms is that they expose their distinct properties for graphs more strongly with a growing number and especially large numbers of nodes. This is due to the binomial and power-law distributions for node degrees and can be see when comparing Figures 3.11, 3.12 and 3.13. The small graphs generated with these algorithms tend towards a more uniform shape.

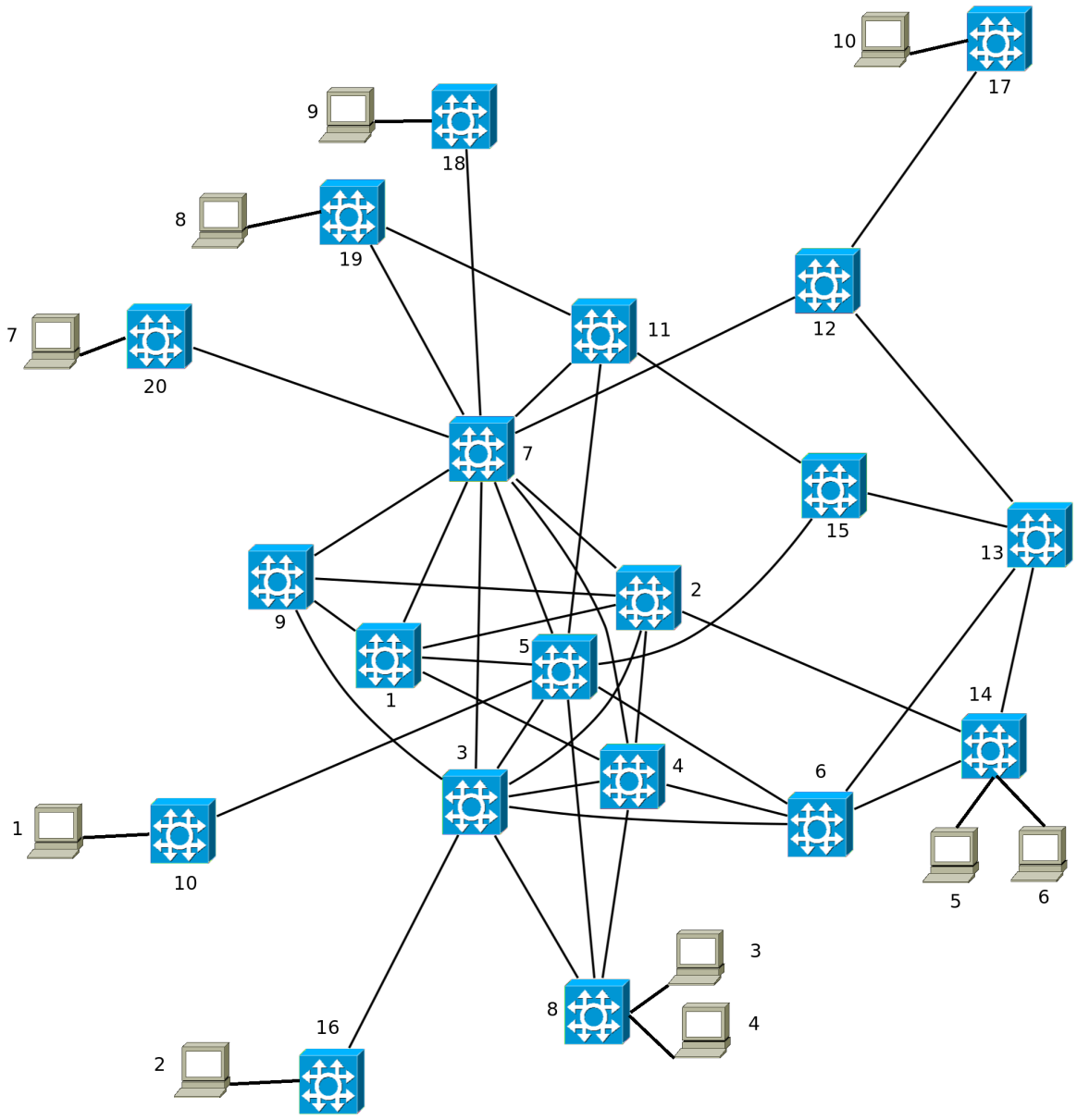

Figure 3.10: Random graph generated with the Barabasi-Albert model [BA99] through growth and preferential attachment and additional random re-connections of edges, therefore called Extended-Barabasi-Albert algorithm.

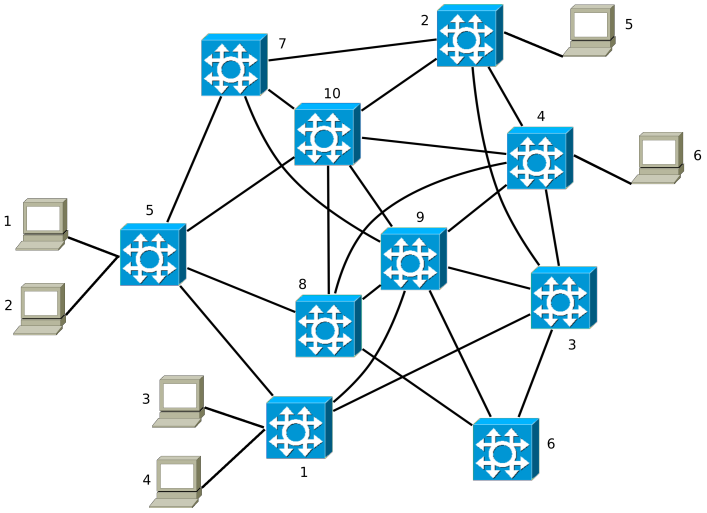

Figure 3.11: Random graph generated based on the Erdős–Rényi model [ER60], with binomial degree distribution

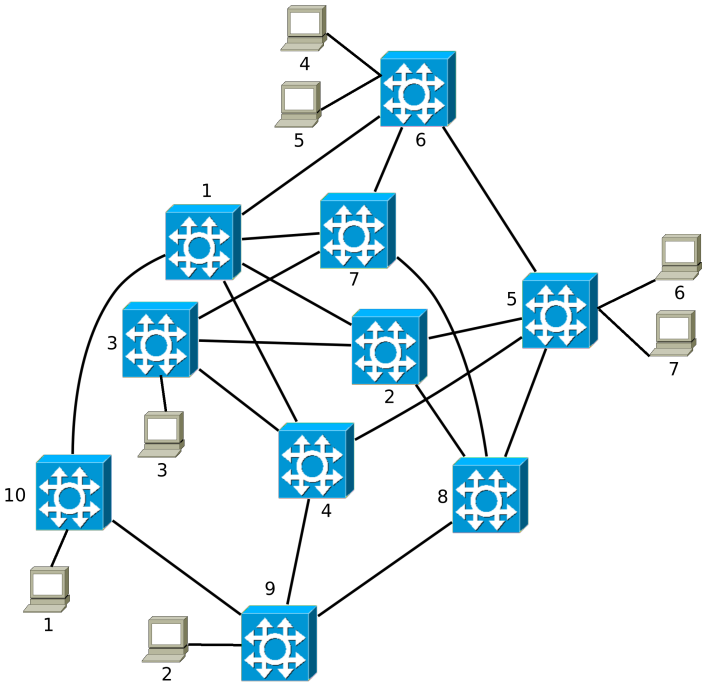

Figure 3.12: Random graph generated with the Newman-Watts-Strogatz algorithm [NSW01] that displays small world properties.

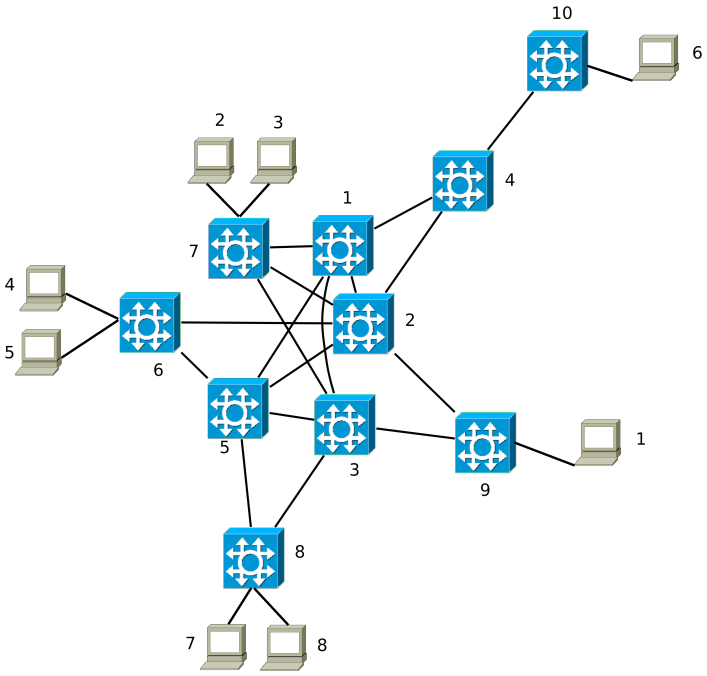

Figure 3.13: Small random graph generated with the Barabasi-Albert model [BA99] through growth and preferential attachment.

3.2 Control Applications

SDN conflicts implicate a conflict of interest in controlled domains and resources. Therefore it is necessary to observe the cumulative effects of several control applications with different intents for a given network and its traffic profile. The aim is usually filtering, throttling, redirecting, balancing of flows and determining efficient paths between two endpoints. In this experimental approach seven control applications with varying functions are considered as well as a base routing implementation. Integration of new implementations in the application or control layer is supported by the flexible experiment model 4.1 and a Hypertext Transfer Protocol (HTTP) interface based on the OpenFlow [ONFb] standard that abstracts from the Ryu controller framework [Nip]. Figure 3.14 depicts the components that form the Representational State Transfer (REST) [Fie00] interface and allow for querying of topology and data plane state by exposing status messages, as described in Figure 2.4, as well as flow modification messages. The decision to exempt packet-in messages from the REST interface is based on the assumption that the additional latency that comes with HTTP connections affects performance of the control channel and controller, since delayed decisions will result in an increased amount of packet repetitions and bandwidth on the control channel. Implementations in the control layer are limited to the scope of the Ryu controller framework [Nip] and several custom components of the experiment testbed. The module *utility_detector.py* provides methods for flow entry insertion. Their employment is mandatory for enabling the detection prototype. Current state of topology components, i.e. the available datapaths, links, and known endpoints are obtained via the module *topology.py*.

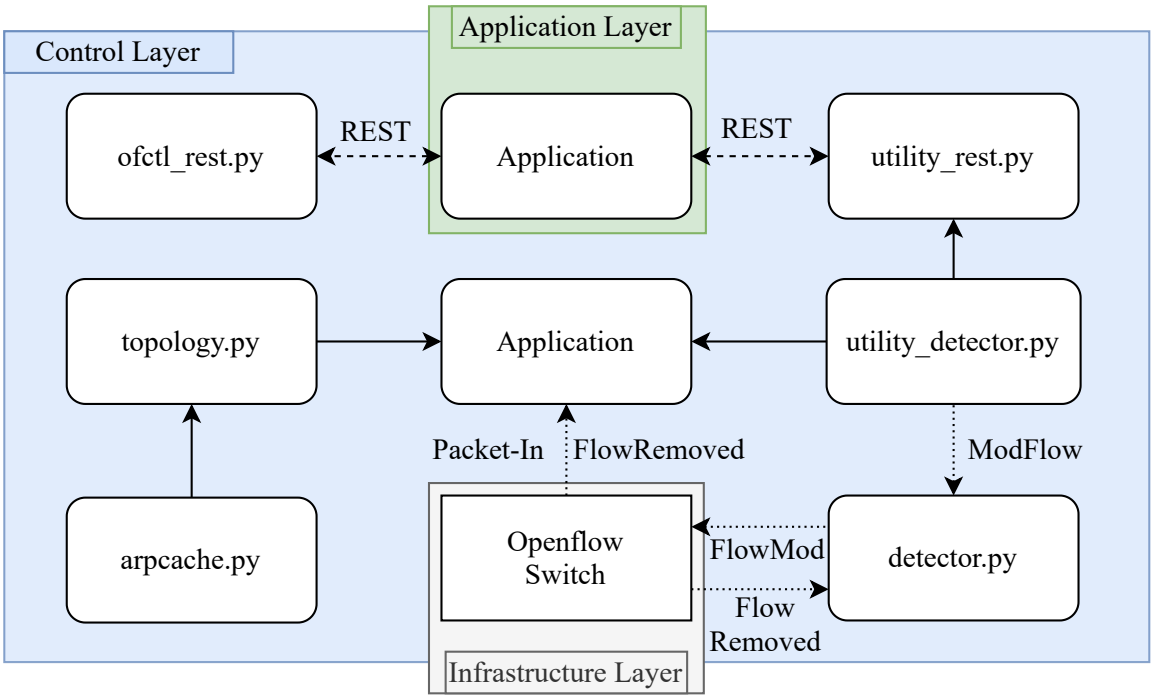

Figure 3.14: Control applications within the Ryu controller framework [Nip] subscribe to and request controller messages, insert flow entries via the module *utility_detector.py* and access topology information through the module *topology.py*. REST interfaces conforming to the Openflow standard expose flow table and topology statistics through the module *ofctl_rest.py*. The implementations are part of the work in [TR].

3.2.1 Firewall

Operating in the application layer, the firewall bandwidth manager implements the REST communication pattern described in Figure 3.14. Configuration options comprise a maximum bandwidth for switch ports, a maximum per flow bandwidth and the interval of the monitoring cycle. It functions as a reactive blacklist targeted at individual flow entries without limitations to OSI layers and match fields. The amount of data for each flow and port is monitored in megabyte (MB) per second by requesting flow table and port statistics of each target switch. The delta between two measurements is defined by the monitoring interval and the total runtime of the update algorithm on reach interval. To avoid permanent stress on the REST endpoints of the controller, the monitoring interval is necessary to set a minimum time difference between sending requests. The cached statistics for ports consist of the amount of transferred bytes, a timestamp and the calculated bandwidth, which are compared to newly requested values and mapped to a switch's port number. Flow entries are mapped to a switch's egress port and the string representation of the match fields, which ensures uniqueness for flow entries that are functionally equivalent, i.e. equivalent for every match field. Below code snippet 1 demonstrates how flows are filtered depending on port and flow bandwidth limits. If a flow's bandwidth is higher than the threshold, the actions of the flow entry are flushed, effectively dropping the flow. The priority is incremented to

enforce the block policy and it's timeout is increased exponentially. The blocking policy is active until the timeout is reached. Connections that repeatedly violate the threshold, induce the exponential increase for the mentioned timeouts. Once all flow statistics are processed and filtered, the port statistics are checked against the port threshold. Port bandwidths exceeding the limit lead to a random choice of connections that are blocked, although with static timeouts. In order to reduce the runtime complexity and avoid cumulative latency, REST requests are sent concurrently. As soon as all results arrive, the data structures are updated and individual timestamps are determined. Nevertheless, updated bandwidths are snapshots, and the latency between request of port and flow statistics is only addressed by the individual timestamps.

```
input: cached port metrics, cached flow metrics, target switches, flow bandwidth
       limit, port bandwidth limit
foreach switch in target switches do
    flowStats ← getFlowStatsForTargetSwitch;
    portStats ← getPortStatsForTargetSwitch;
    blockCandidates ← emtpy list;
    foreach flowStat in flowStats do
        bandwidth ← calculcateFlowBandwidth(flowStat, cachedFlowMetrics);
        if bandwidth > flow bandwidth limit then
            update cached flow metrics;
            block flow;
        else
            update cached flow metrics;
            update blockCandidates with flowStat;
        end
    end
    foreach portStat in portStats do
        bandwidth ← calculcateFlowBandwidth(portStat, cachedPortMetrics);
        while bandwidth > port bandwidth limit do
            candidate ← random element from blockCandidates;
            remove candidate from blockCandidates;
            subtract flow bandwidth from bandwidth;
            block flow;
        end
        update cached port metrics;
    end
end
```

Algorithm 1: Core algorithm of the REST Firewall Application

3.2.2 Path Enforcer

Enforcing specific routes in a network is a form of traffic engineering that allows for controlled flow through nodes where further processing, packet inspection and filtering is implemented. It also allows for confining flows to a less used or prioritized route. The application is configurable to discern between OSI layer 4 protocols. The topology depicted in Figure 3.3

provides a suitable complexity to showcase an example configuration. Given switch 9 as a target switch and switches 7 and 5 as jumps, in this specific order, the app would route the flows of hosts 5-9 to hosts 1-4 through switch 7. Flow entries will only be installed on switches 9 and 7, from there the packets are sent to switch 5 for further handling. Traffic originating from hosts 1-4, with hosts 5-9 as destination, will be routed directly to switches 10 and 11, since traversing the jump nodes 7 and 5 yield would result in a self-induced loop when the packets reach switch 9 repeatedly on their way to the destination.
Any target switch of this application is given a number of other switches in the topology and the implementation will try to send traffic along this route. The configuration can be a series of directly connected switches or any series of switches within the topology with an unspecified route between them. The implementation interpolates any unspecified links in the jump path using a shortest path algorithm. To enforce the entire route, flow entries are installed on all but the last node. A flow entry for the backward flow towards the incoming link is installed on the target switch. Thus, from the perspective of a specific target switch, we can only route one direction of a bidirectional communication over the jump path and avoid ambiguous configurations. There are several edge cases that need to be managed when forcing traffic on a specific path to avoid traffic loops. Let us consider a scenario where packets are sent upstream from a client to a server. If the traffic flow traverses a bottleneck node, such as switch 8 in topology 3.4, and the desired path leads over switches that are downstream from that node, the path cannot be enforced for the upstream flow, since it will create a self induced loop. In this example, the client is endpoint 1 in topology 3.4 and the server is endpoint 6. The desired path from switch 8 leads to switch 7 and then switch 6. A simple method to prevent this occurrence is a check of the route from the last jump node towards the destination of a packet flow. If the target switch is part of the determined route, the edge case is confirmed. In this scenario, the implementation ignores the upstream flow and installs a flow entry on switch 8 that outputs traffic to switches 9 or 10. The returning downstream packets can be routed over the specified jumps, and a flow entry towards switch 7 is installed and from there to switch 6. The next edge case is the interaction with routing decisions of other applications. The path enforcer does not install any flow entries on the last node on it's jump route. Therefore it is paramount to check the quality of the route from this last node towards the destination endpoint. If it is not a shortest path route, the risk is too high that another application will reverse the decision of the path enforcer and routes traffic back over one of the jumps. This will result in a distributed conflict and a shortcut traffic loop. While this simple heuristic cannot prevent loops in general, it still prevents trivial loops from emerging. Another trivial edge case is a sanity check for a valid path from source to destination. If the specified jumps on the path are not valid, or a route cannot be determined, the path enforcer ignores the packets and does not install any flow entries.

3.2.3 Host Shadower

Inspired by the concept of domain shadowing, the Host Shadower can be categorized as an adversarial application within a SDN. Traffic is redirected from an intended target host to a backend host, while the counterpart in the communication stays oblivious of the redirection. Next to malicious intents, the implementation allows for prioritization of servers depending on network routes and domains. Consequently, the application could be used in a configuration where a frontend host does not offer the targeted service at all. An example

configuration in topology 3.3 of this app could be switches 10 and 11 as target switches. Host 6 is the frontend host and host 7 the backend. Packets flowing from endpoints 1-4 to host 6 will be transformed in the target switches to destination IP and MAC addresses of host 7. The returning packets will be adapted again to contain the IP and MAC addresses of host 6 as source entries.
The implementation transforms OSI layer 3, layer 2 addresses of forward and backward flowing packets and routes the traffic to the backend. The flow entries for the backward flows, which rewrite the packets to make them appear like they are originating from the frontend host, are installed on all target switches of this app. A list of one-to-one mappings between a frontend server whose layer 2 and layer 3 addresses are exposed and a backend server that provides the service forms the configuration. The only edge case involved in handling the packet rewrites is constituted by a check if hosts are directly connected to a target switch and if a route to the backend server needs to be determined, in order to send packets out to the correct link and install flow entries accordingly.

3.2.4 Routing

This is an OSI layer 2-4 and universally deployed application based on a shortest path first algorithm and devised by Tran [Tra22]. It is session aware if applicable and handles TCP, UDP and ICMP flows. The application is topology aware and generates paths depending on the state of nodes and links in a network. It implements caching for layer 2 address resolution. Since the routing function is required on every switch node in the network, it is the most likely candidate to produce local conflicts, such as generalization. Consequently it is implemented to be aware of configurations of other applications in the control layer to avoid handling of flows that are supposed to be processed by respective control functions. Although this can not be considered a realistic approach for an entire application stack in a SDN topology, especially for those that implement a REST service, it is necessary to enable the unobstructed control flow that leads to distributed conflicts. Considering topology 3.2, this application will always suggest a path between switches 1-2-4 or switches 1-3-4 and vice versa when processing packet-in messages for traffic between host 1 and hosts 2,3 and 4. It will send a message to mentioned switches via the control channel to add appropriate flow entries. The configuration comprises only the target switches, which will always be all switches in the topology.

3.2.5 Path-Load-Balancer

This application aims at prevention of overloaded links in the topology and is conceived by Tran [Tra22]. Once a threshold is reached for a specified port on a target switch, the flow with the largest bandwidth is redirected on any alternative paths between two endpoints in the network. Depending on a fixed monitoring cycle and the threshold, flows are redirected. Consequently, flow paths from and to a destination might differ and paths are not optimal concerning the number of traversed nodes. The configuration contains the monitoring cycle in seconds as well as target switches with individual thresholds for port bandwidths. Considering topology 3.2 and the target switch 2, the implementation will monitor the outgoing ports to switch 4 when host 1 communicates with the other hosts. If the bandwidth on this port surpasses the specified threshold, the largest flow will by redirected to its destination via the link to switch 3 and from there to switch 4. In this work the configuration and im-

plementation is changed to allow for individual bandwidth limits on target switches, instead of a global threshold. Individual thresholds allow for configurations that are adjustable to different regions of a topology and address the fact that topologies can comprise any combination of network shapes and degree distributions on its nodes. In light of the experiments, the increased flexibility can reduce the complexity concerning experiment iterations.

3.2.6 Passive Path-Load-Balancers

Balancing network load on all available paths between two endpoints is the intent of this application, which is devised by Tran [Tra22]. Flows from specific endpoints are balanced alternately between routes, depending on the number of flows. Starting from a target switch, the traffic of a previously seen origin is fanned out. This application comes in two variations. One will balance and consider paths outgoing to a destination and beginning from the balance point, e.g. traffic to a server, and the other will balance on the return paths to an initiator of a flow, e.g. the answers of a server to a client request. Both the source and destination based version takes a list of servers as configuration. As an illustration, we will consider topology 3.3. The source based path load balancer deployed on switch 9 and given switches 6-9 as servers will distribute flows on all paths between switches 1-9. This not only includes balancing for links to switches 6-8 but also the two possible paths from switch 5. The destination based variant with the same configuration, will perform the balancing on paths facing to the right side of switch 9 to switches 10-15.

3.2.7 Endpoint-Load-Balancer

The scenario for this application is a server with several mirror servers. The application balances flows to a server and its proxies in a round robin implementation and is developed by Tran [Tra22]. The configuration contains the target switches as well as a set of servers with their proxy endpoints. Each target switch balances the set of servers that are provided in its configuration. In an example configuration we consider topology 3.3. Host 6 is defined as server for the service and hosts 7-9 as proxies. When deploying the application on switch 9, connections from all other hosts will alternate between the server and its proxies but the hosts will appear to be communicating with host 6, the main server, only. At the balance point switch 9, packets will be manipulated going out to the proxies and returning from proxies and layer 2 and layer 3 addresses will be rewritten.
The adaptions of the Endpoint Load Balancer affect the number of balancing configurations and as a result, also the configuration format. Where before only one set of proxy server and offloading servers was implemented, now a list of mappings of proxy servers to offloading instances is possible. The specification of endpoints and their mapping is loaded from a JavaScript Object Notation (JSON) file that adheres to the list of mappings. Consequently, the application features increased flexibility and allows for configurations that had to be covered by multiple experiment runs before the changes.

3.3 Experiment Automation

All presented topologies are complemented with an experiment model 4.1 that defines which region of the parameter space will be covered. This includes configuration files for control

applications and mappings for endpoint-combinations as well as target switches. All permutations of possible priorities and application configurations are iterated, each representing a data point in the parameter space 2.5. Comparison of flow insertions when running an application isolated from others to it's policy insertion in a combined run, exposes symptoms for possible conflicts. If an application behaves differently during co-deployment, it might not enforce expected policies that hint at a new or known conflict. Another indication of possible deviations between isolated and combined runs is the monitored bandwidth through link-layer virtual ports. Fluctuation in bandwidths and deviating paths in the network can again lead to unexpected policy insertions.
The current implementation of experiment automation exhibits problems in the build process of topologies, namely resilience against memory allocations for switch and host nodes as well as errors due to timing issues when generating physical machine instances for them. These might appear to be minor technical details, yet corrupted images or stale block devices break the automation process, and missing nodes in a topology render it invalid. Furthermore, a scaling of the topology also requires a scaling of the provided resources, i.e. a feasible number of processors, memory and disk space. While these can only be provided by the administrator of the host machine, it is important to give feedback and provide a means for allocation in an automated system. The experiment runs proved to need at least four processors on the host machine, while the controller node required two of them to run reliably even in small topologies and at least 512 MB memory. The results of the experiments include monitoring of bandwidth on ports and endpoint reachability as well as data plane comparison on target switches. The processes to generate these results are application bound and implemented redundantly since there is no common interface to define the input for them.
Depending on the number of applications in an experiment and the number of their configurations, the runtime complexity of the experiments lead to errors on smaller, less reliable machines and lacking conciseness renders the result output convoluted. The complexity of an experiment concerning the number of deployed applications a and the number of configurations per app c is given by Formula 3.3, where r is the current subset of applications that are deployed together. This is assuming that all applications have the same number of configurations. The first term c^r denotes all combinations of app configurations, r^r all permutations of app priorities, and the last term stands for the number of subsets that can be obtained from all apps a and a given r. This includes redundant runs for all equal priorities. An example of four applications with two configurations each requires 2440 repetitions of starting the controller process, simulating traffic, analysing the data plane and generating result files. While this does not appear as a large number, depending on the size of a topology traffic simulation takes up to several hours or days. Since checkpoints for the execution or intermediate output of results is not implemented, this could entail a substantial loss of computational resources if any failure occurs. The same insights can also be obtained by multiple experiments with less configurations per applications and will reduce the amount of repetitions to 320 by a factor over seven. However, this requires 16 individual experiments with one configuration per app to cover the same fraction of the parameter space. The benefit are individual results that can be check-pointed or transferred to other machines to avoid loss of data and quicker evaluation.

$$\sum_{1<=r<=a} c^r * r^r * \frac{a!}{(r!(a-r)!)}$$

Increasing and improving the level of automation of the experimental process addresses the need for reproducability, scalability, reliability and usability. If the inputs and outputs of the system are unclear or hard to reproduce and an interface that abstracts from technical details is missing, scalability and usability are compromised and the main task becomes the configuration of the software, as opposed to using it. This work focuses on a concise and flexible experiment configuration that produces tangible results and reproducible experiments. The goal is to drive the automation process by one source of configuration for any type of application and topology, which enables us to develop a general process of generating experiment results on a common interface even for new applications. The key component is a formal experiment model as well as several loosely coupled modules for generating the technical infrastructure, any required configuration files and for running the experiment.
The desired architecture is depicted in Figure 3.15 and shows the key components of defining a testbed and application configurations, building a topology for the testbed as well as deployment of an experiment and the generation of results. A well-defined model is the prerequisite for easy reproduction and serves as the entry point of both topology generation and application configuration. A topology builder module and asset generator prepare specification files that define the topology, including scripts for executing the experiment, technical details and other resources. Details on these steps are described in Chapter 4. Given the specification files as input, the topology builder module either generates the testbed directly on the local machine. The config generator extracts the model specification concerning the target domain of the parameter space and produces global and individual configuration files for the SDN controller. The experiment is run based on the before mentioned configurations, comprising all dimensions of the parameter space 2.5.
A common model, especially for app configuration, permits for a generalized process of app deployment, monitoring of bandwidths and specification of endpoint interest, traffic generation as well as automated comparison of relevant regions in the data plane. Results include a copy of the flow tables on all switches, a log of all detected conflicts and the input parameter space. Several reasons call for a common interface and format to structure the input for an experiment run and describe a topology. Firstly, specifying and building complex topologies with Openflow switches, endpoints and the required edges is an error prone task, while the only novel contribution of a new topology is purely its structure. Therefore, we can enforce a simple format for defining such a structure. Secondly, in order to generate experiment results that are not application bound, we need to enforce a shared, yet flexible interface to configure applications. This declarative approach will conceal the technical details of the testbed while exposing its functionality to a researcher. An explicit model is a tool for a user of the system as well as a tool to determine if a user is using a system correctly and provides transparency by checking structure and types of the experiment input. Researchers testing a new application can integrate their configuration in the provided format and use it in their app deployment. However, the most important utility is the declaration of interests, namely on target switches and endpoints, and automated analyzing or detection of conflicts. Topology generation, application configurations and experiment deployment can evolve over time, yet the input model has an independent life cycle. A modular implementation of the architecture allows for an adaption or replacement of individual components. This makes the architecture flexible for future maintenance, refactoring, improvements and updates to new technologies.

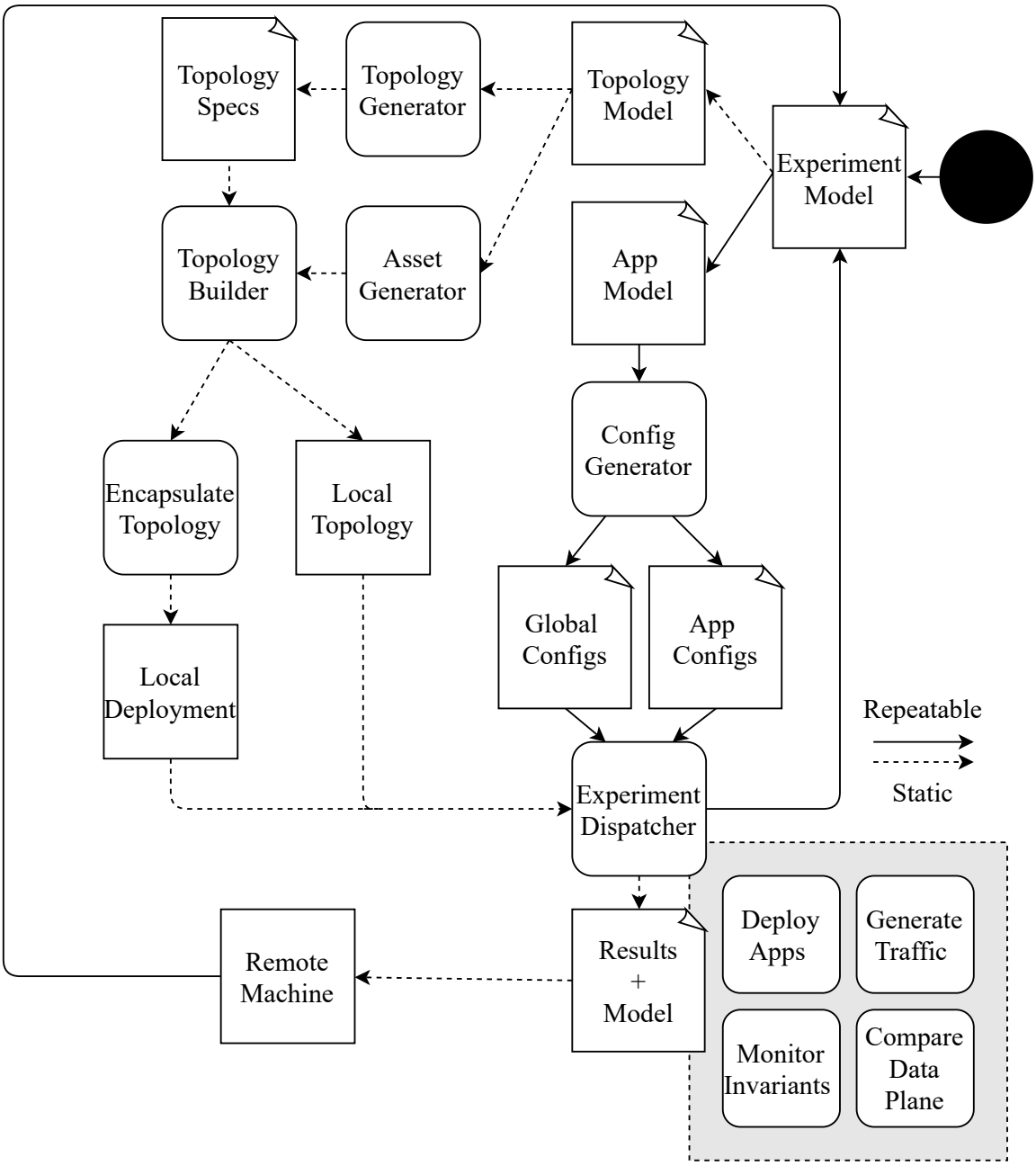

Figure 3.15: Architectural model illustrating the key components of testbed generation and experiment execution, based on reproducible inputs and results.

4 Experimental Infrastructure

Since the experiment model is the entry point for testbed automation, the structure and utility is introduced first. The following sections describe the additions to the existing code base for experiment automation and topology generation and any exposed errors, with the appropriate patches. While researchers need to administer the entire procedure in the current state of the project, decoupling of components allows for an Infrastructure as Code implementation, where only an experiment model and any new applications are needed as input and results are provided as output. Additionally, any experiments on a generated topology can be repeated, adapted and processed in bulk. Readers interested in testing the implementation should refer to Sections 4.1 and 4.5 and can avoid the technical details describing the changes of the current system and new implementations. The source code for the implementation of the experiment model and experiment building process, as well as for the practical example deployment is accessible in the public repository [TR]. The mentioned and depicted source code and files are part of the file structure shown in the Appendix 8.2.

4.1 Experiment Model

YANG is a data modeling language common in the networking domain and specified in RFC 6020 [IETF]. It features a type system and customizability for generic definition of configuration files. The following YANG model is used to generate experiment model instances in Python [PSF] as JSON files and to verify said JSON files as input for testbeds. Suitable libraries for other programming languages exist. This implies that researchers can also configure experiments by directly writing JSON instances and use them as input to build topologies and run experiments.
The model is structured into three main parts, i.e. topology configuration, app configuration and settings for bandwidth monitoring and traffic generation. The *autostart* attribute is a utility that can be used if necessary, e.g. if additional configuration before an experiment run is desired. *topologyId* is a simple identifier, provided in the result output. The *evaluationRun* attribute is added for easier tests with the automation process. The current implementation aborts an experiment, if a connection between endpoints fails during an isolated run of an application. In case of intentionally constructed conflicts, we need a mechanism to skip this check. Traffic types comprise UDP and TCP. At the current state, the attributes for e.g. bursty or variable traffic profiles are ignored. The attribute *bw_difference_threshold* is used to determine the accepted difference on port bandwidths in target switches. If the throughput on the link changes by the specified amount of bandwidth between application deployments, a warning is generated. As said before, topologies are made up of switches, hosts and links. What the model enforces is uniqueness in IDs and types for the different nodes to avoid any ambiguity. App configurations include the mandatory attributes *cookie* and *targetSwitches*. A cookie is a settable field in flow entries. If an application sets this value and provides information on its target switches, the experiment scripts will check their flow entries and compare data plane state between an isolated and co-deployment of apps. Target switches are

identified by their ID, the same type as in the topology definition. Optionally, switches can declare invariants and assets. Invariants are made up of keys and either an integer or a string as value, or a list of integers and strings. Only one type of value can be given and if multiple are provided, only one will be mapped to the key in the JSON file. This basically allows a researcher to introduce arbitrary invariants, yet in future work a specific set of keys can be defined to declare application interest and use these interests in detection of distributed conflicts. Assets represent references to endpoints or switches and the model allows for providing a one-to-many or one-to-one configuration nodes. Application's configurations and switch configurations have the same format. While application assets and invariants are converted to a flat list, switch configurations will be mapped to their switch ID. This represents a different granularity of scope that the individual configurations are applied to. An example for a set of assets is a list of switches *(jumps,router2, router3)* as used by the Path Enforcer app, a mapping of endpoints *(proxy,pc1):(servers,pc1, pc2)* as used by the Endpoint Load Balancer or a one-to-one mapping *(frontend,pc1):(backend,pc1)* for the Host Shadower. Sample invariants can be an integer representing a monitoring cycle in seconds, a set of IP protocols that an app is supposed to handle or ignore, or a bandwidth limit for ingress ports in a switch. The output and generation of a model instance can be examined in the example for configuring and running an experiment 4.5. While this appears to be an arbitrary structure for app configuration, it is inspired by the current applications but flexible enough to suit new implementations that want to share the format or simply declare interests as an input for generating detailed experiment results. The model aims at flexibility concerning configuration structures, while still providing a test harness for the experimental process and evaluation of endpoint and switch interests. The contained topology definition avoids details tied to the technical implementation of the physical network.

```
+--rw testbed
   +--rw topologyId?                  string
   +--rw autostart?                   boolean
   +--rw evaluationRun?               boolean
   +--rw bw_difference_threshold?     uint8
   +--rw apps* [id]
   |  +--rw id          string
   |  +--rw config
   |     +--rw cookie                cookie
   |     +--rw appInvariants* [invariantKey]
   |     |  +--rw invariantKey       string
   |     |  +--rw intValue?          int8
   |     |  +--rw stringValue?       string
   |     |  +--rw intItems*          int8
   |     |  +--rw stringItems*       string
   |     +--rw targetSwitches* [id]
   |     |  +--rw id                   switchId
   |     |  +--rw switchInvariants* [invariantKey]
   |     |  |  +--rw invariantKey      string
   |     |  |  +--rw intValue?         int8
   |     |  |  +--rw stringValue?      string
   |     |  |  +--rw intItems*         int8
```

```
|      |  |  +--rw stringItems*     string
|      |  +--rw switchAssets* [id]
|      |     +--rw id               int8
|      |     +--rw majorAsset
|      |     |  +--rw assetKey       string
|      |     |  +--rw assetValue     hostId
|      |     +--rw minorAssets
|      |        +--rw assetKey       string
|      |        +--rw assetValue?    nodeId
|      |        +--rw assetItems*    nodeId
|      +--rw appAssets* [id]
|         +--rw id              int8
|         +--rw majorAsset
|         |  +--rw assetKey       string
|         |  +--rw assetValue     hostId
|         +--rw minorAssets
|            +--rw assetKey       string
|            +--rw assetValue?    nodeId
|            +--rw assetItems*    nodeId
+--rw trafficProfiles*            trafficProfile
+--rw trafficTypes*               trafficType
+--rw switches* [id]
|  +--rw id     switchId
+--rw hosts* [id]
|  +--rw id         hostId
|  +--rw source?    boolean
+--rw edges* [id]
   +--rw id        uint8
   +--rw nodes*    nodeId
```

Listing 4.1: Source code of the presented experiment model implementation in YANG [IETF] syntax.

4.2 Technical Infrastructure

The presented network topologies are made up of testbeds that simulate nodes as virtual machines (VM). Host virtualization is the concept of emulating physical and software resources of a host machine for a guest operating system (OS) through a layer of abstraction, the hypervisor. The paradigm of para-virtualization requires that a guest OS is adapted to be run on a host OS through the hypervisor. This provides for a more light weight hypervisor and a more efficient execution of an individual VM. The caveat is that Xen [TLF] guest domains need to be run on a host that provides their specific Linux kernel version. To satisfy this requirement on any given physical host that cannot provide the right Linux kernel version, the setup allows generation of testbeds within an additional wrapper VM. The outer virtual machine is created with the open source version of VirtualBox [OC] and runs Debian Stretch as OS. Virtualbox [OC] is based on a hypervisor that implements full virtualization. Guest machines are agnostic of the host OS, but that requires a heavy-weight implementation in

the hypervisor. The inner VM are Xen [TLF] guest domains and the edges are connections between the their virtual network interfaces. The interfaces are bridged through the main master bridge in the outer VM. Every switch node is connected to a controller VM on one network interface to provide a management net based on one IP subnet. All other interfaces, depending on the degree of the node are layer 2 links.
Ryu [Nip] is a Python [PSF] based implementation of the OpenFlow [ONFb] protocol. Python is an interpreted, object-oriented programming language suitable for scripting and light-weight, rapid development and an extensive system set of libraries. A Ryu controller starts threads for all attached control applications, so basically apps are started in parallel. The controller main thread maintains events and exposes them to subscribing applications. Ryu provides several built-in REST applications that expose the messaging mechanism, topology state and OpenFlow 2.2 control functions to HTTP clients. Representational State Transfer (REST) is a concept defined by Fielding [Fie00] in his thesis on styles and designs of software architectures. The HTTP standard implements the suggested architecture. Key properties comprise a client-server architecture, statelessness in servers and a uniform interface between servers and clients. Since state is maintained in clients and all clients implement a common interface, a REST control application can be implemented in any programming environment and on any endpoint, complementing the pluggable nature of SDN.
Network traffic between the host nodes is generated using the tools iperf [Sofb] for testing the bandwidth limits and netcat [Sofc] as a general utility for sending data between computing nodes. Both tools provide the option to generate connection oriented TCP and connectionless UDP traffic. This simulates flows in the network that can be handled by the control applications. Monitoring, logging and analysis of the generated traffic is implemented with the tool tcpdump [Sofa].

4.3 Experiment Building

The static components depicted in Figure 3.15 are referenced in VM specification files that include a unique identifier for each VM instance, a path to a template OS image, as well as any desired files that are copied to the file system. File resources for the host machine or wrapper VM contain a copy of all experiment automation scripts from the project repository, including the apps, their configurations and setup scripts to create network interfaces and assign MAC and IP addresses. Furthermore, researchers can provide a path to a directroy that contains a cryptographic key-pair and *ssh_config* file for a Secure Shell (SSH) connection to a remote machine. Transferring intermediate results for a batch run of experiments helps reduce the risk of data loss. It is also a means for enabling an Infrastructure as Code service in the future, where researcher do not have to administer the host machine anymore and simply provide an experiment model and get results. Security is an issue in this approach but it serves to showcase the utility of the experimental process.
Template images for controller, switch and host nodes are fully configured and contain all necessary installations and resources for experiment execution to enable a deployment with or without public internet connection. Node specifications additionally contain a list of network interfaces. The first for each is bridged over a master interface in the host machine to form a management net and to establish the control channel between switches and controller. The following network interface identifiers are paired to form the edges of the network and each pair again is bridged over a dedicated master interface on the host machine. Figure

4.1 illustrates management and testbed networks as well as the control channel. The auto-generated VM IDs are used to establish SSH connections to the individual nodes through the management net. IP addresses are assigned to hosts within the testbed net. Figure

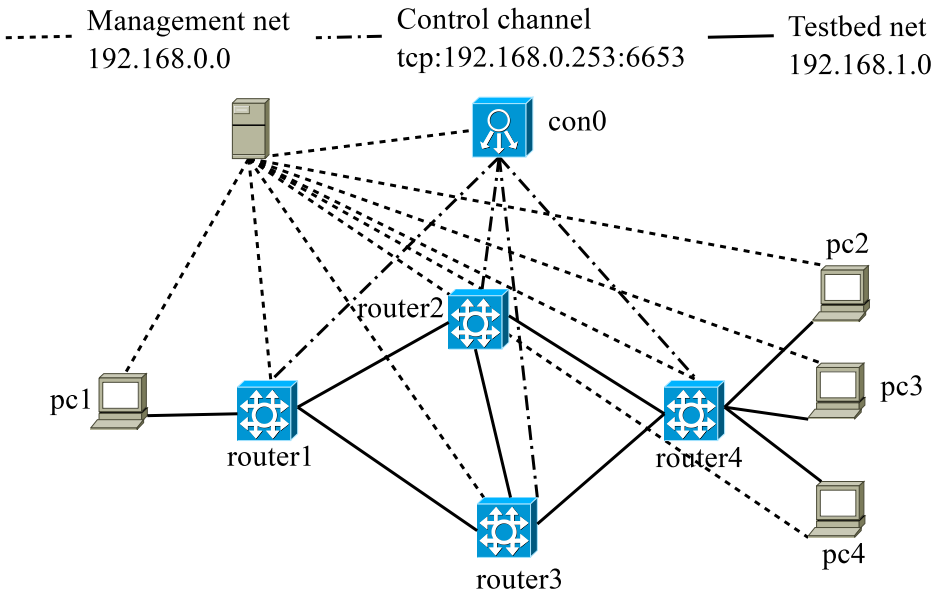

Figure 4.1: Sample topology with depiction of management, testbed net and control channel between controller and switches. All network interfaces are bridged over interfaces of the host machine to form the edges of the testbed net. Host nodes are assigned an IP address within the testbed network.

4.2 shows the resources involved in generating VM specification files and their dependencies. A requirement for automated generation of specification files is the adaption of the script *constants_generate_spec.py* to a user's settings in order to provide absolute paths to the VM template images. It features an absolute path to a directory containing an archive with scripts to create bridges and Xen [TLF] domains, and the mentioned optional SSH configuration archive can be added at this location. Depending on the desired method for defining an experiment model, a programmatic approach here exemplified by the python script *sample_topology.py* or a corresponding JSON file is the entry point for the experiment build procedure. The YANG model is translated to a python module with the shell script *setup_pyang_module*, which requires the software library *pyangbind*. The module *random_topo.py* provides several functions to generate random topologies described in Section 3 and a helper method to generate a diagram for switches and edges. Once a JSON file has been created manually or via code, the script *generate_sdn_spec.py* is executed and given a path to the model instance. The helper class *parameter_space_loader.py* loads and validates a YANG models and processes different model versions. The loaded model is used in *app_config_generator.py* to create app configurations. These two modules are reused in the experiment execution process and ensure adaptability for future changes of YANG models. The main script *generate_sdn_spec.py* integrates app configurations with the required scripts from the project repository and generates the final VM specification files that tie together all references and mentioned settings. This architecture let a user specify nodes and edges in one configuration file and abstracts the generation of specification files and details on

network interfaces. Up to this point, no VM image has been instantiated. The generated

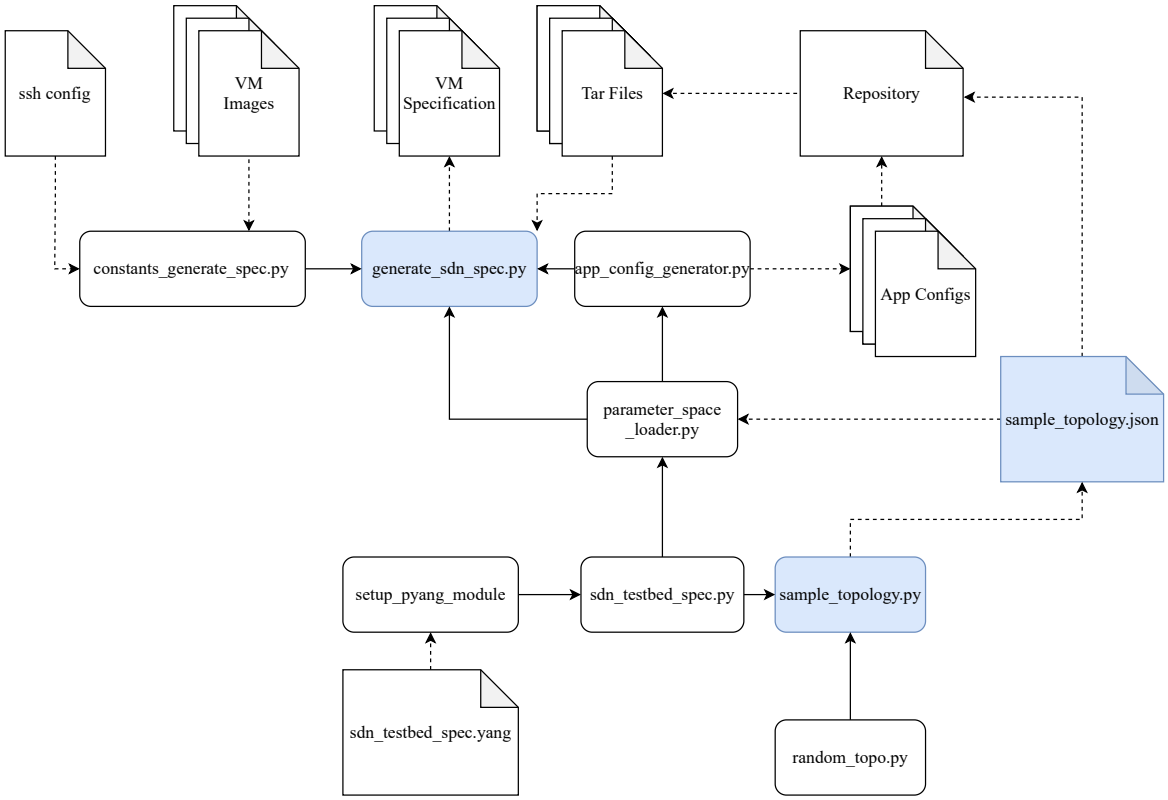

Figure 4.2: Figure depicting the scripts and static resources that generate VM specification files. The specifications aggregate all information to produce OS images, a correct setup of network interfaces of a testbed and an experiment run. Blue components mark the entry points for defining an experiment model instance or generating a specification from it.

specification files serve as input for the topology building processes depicted in Figure 4.3. The main tasks are copying and configuration of template OS images as well as the insertion of necessary resources to the specified destination in the host machine or into an optional wrapper VM. Once the machine images are assembled the topology is deployed on the local machine.

An adaption of the configuration in *constants_build_env.sh* for the local directory is a requirement for running the build process. The default configuration creates VM images in the */tmp* directory and it is important to provide absolute paths with any change of the configuration. This entails that encapsulated topologies will be deleted on any Linux system upon restart or shutdown, so depending on the host machine and the retention period of the topology, an adaption is advised. The entry points for topology generation are the two scripts *deploy_sdn_conflicts_experiment.sh* and *build_sdn_conflicts_experiment.sh*. The minimum requirement for building a topology is the software package *qemu-utils* and a Debian-based OS with a kernel version that matches the version of the template images. Another requirement for the direct local deployment is a Linux kernel version with the Xen [TLF] hypervisor extension. The script *deploy_sdn_conflicts_experiment.sh* will copy a Virtualbox [OC] VM

image and insert all resources and Xen [TLF] VM images into its file system. It calculates and prints the required memory for the topology to the command line and prompts the user to enter base settings for allocation of processors and a local port for creating a local SSH port forwarding for access to the wrapper VM. Root access is required to administer and access the Virtualbox [OC] VM. The latter script omits any steps for the wrapper VM and deploys the Xen [TLF] domains on the local machine directly. All file resources will be also created on the local machine. The auxiliary scripts exhibited several issues that entailed some unreliability, lacking portability between Linux kernel versions and inhibitions to scalability. The script *common_network_helpers.sh* produces MAC addresses for the Xen [TLF] domains and is adapted to generate up to 253 IP addresses. The implementation of the script *common_qcow_helpers.sh* is adapted to remove a portability issue with caching and asynchronous I/O mode between threads for connection of a local block device to a mounted image using the command *qemu-nbd*. While the previous image might increase efficiency with the right OS, the implemented settings are more portable and less error prone in general. Another issue was a race condition between unmounting the outer VM image and detaching the network block device, which lead to unreliable behavior, frozen network block devices and invalid images. The commands are now reversed, to ensure that a block device is detached first and an image is unmounted if the detachment is successful. Furthermore, the script *common_generate_xen_topology.sh* is a slimmed version of *common_generate_qcow_image.sh* that omits steps for the wrapper VM and deploys the Xen [TLF] domains on the local machine.

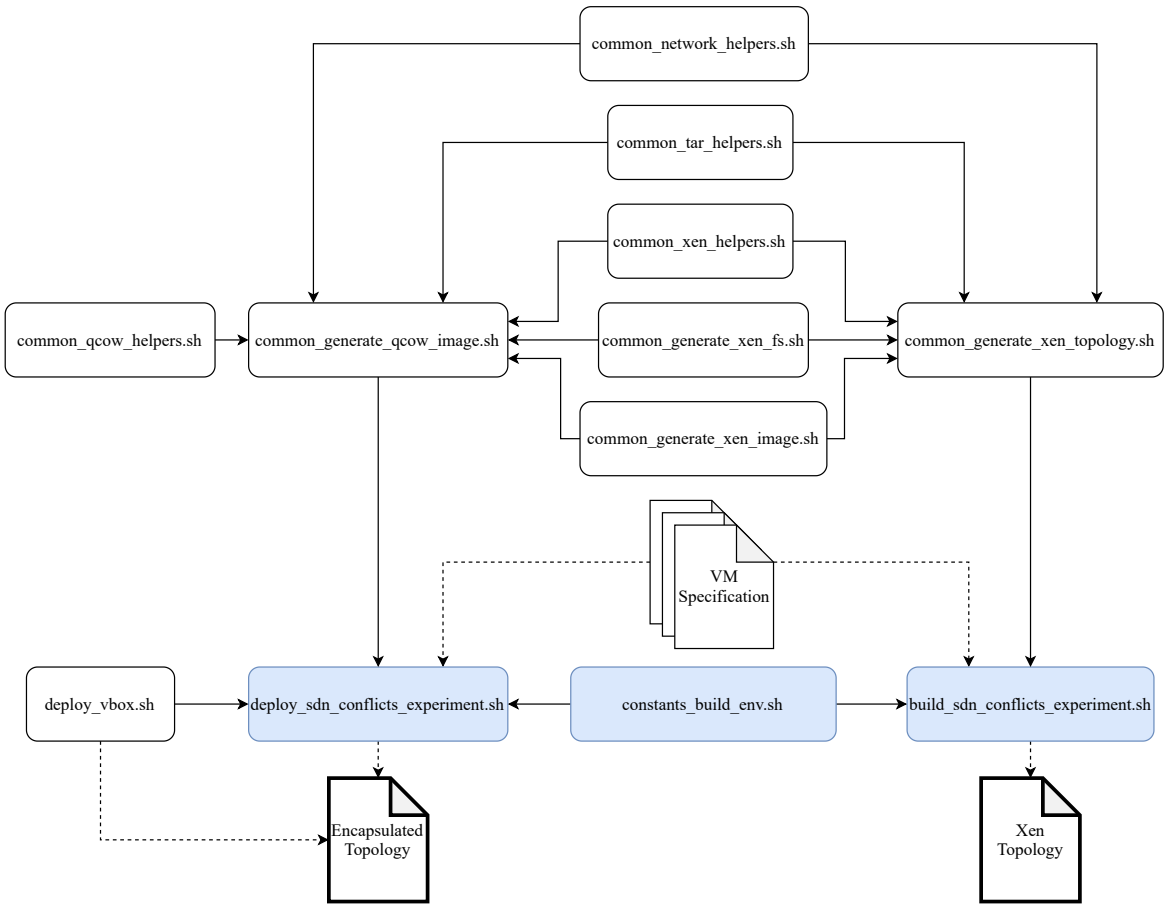

Figure 4.3: This diagram shows the dependencies and components of the topology generation system. Entry points for the topology build process are the blue components and take a path to VM specifications as input. The result is a local deployment of Xen [TLF] domains or an encapsulated topology within a wrapper VM. The scripts originate from the MNM-Team [MT]. Adaptions and additions are described in this section.

4.4 Experiment Execution

Execution of an experiment, i.e. reachability tests, traffic generation, invariant monitoring, data plane comparison and result generation, requires the following preparations within a testbed. First of all, bridges that connect the interfaces of nodes within the testbed are created. The Xen [TLF] domains that form the testbed need to be created from the generated images and configurations. Endpoints need to be assigned IP addresses within the testbed network and the switches connections to the control channel are established. The script *rnp_vms.sh* originally implemented by the MNM-Team [MT] and adapted in this work ties together all of these steps and is executed automatically once a topology has been deployed. The script is adapted to avoid errors during Xen [TLF] domain creations, a critical process for the entire automation. The error is caused by the fact that all Xen [TLF] domains are started in background processes at the same time. This leads to memory allocation errors of the Xen [TLF] hypervisor and occurs on a regular basis even for small testbeds and on any host machine. Trying to start up Xen [TLF] domains in parallel is more time efficient, but even one error had the effect of total failure of any further automation. Even when starting the domains sequentially, said memory allocation errors can occur and will occur for large topologies. Therefore, the scripts are adapted to check if domains are created successfully and repeat the process on failure. It is paramount that the process is reliable for further automation steps. For experiment configurations that set the *autostart* attribute explained in Section 4.1, the following steps are executed automatically through the script *config_and_start.sh*. Otherwise they can be performed manually. The script *config_testbed.sh* by Tran [Tra22] requires the number of switches and hosts as parameters to configure all nodes in the testbed. The script *read_parameter_space.sh* starts an experiment run as specified in the model instance, i.e. a simulation of all combinations of applications, application configurations and application priorities on the testbed. For automated starts of experiment runs, the process is started within a tmux [Mar] session. The results for each permutation include:

- a dump of the entire data plane state
- logs of detected differences between isolated and co-deployment of applications in data plane state and port bandwidths in respect to the configured bandwidth threshold
- a log of any known conflict classes that are detected by the detection prototype
- a copy of the input parameter space

The common interface for application configuration through the experiment model allows for a generalized process to produce these results. The script *read_parameter_space.bash* is adapted for this purpose. While before the implementation was application bound, now any application configured through the model will be considered. Furthermore, deviations between traffic bandwidth on switch ports is performed based on a static value defined in this script. The parameter is now definable through the experiment model and processed by the script. Concerning bandwidth monitoring, the threshold was applied to any case. This implementation fails to address the significance of a link that did not receive any bandwidth in an isolated run but is used in a co-deployed instance. Now we also take ports into consideration that exhibit bandwidth changes below the bandwidth threshold but are unused in expected state and used in observed state. Simply said, when we encounter a

bandwidth of 0.0 in an isolated run on a link and a bandwidth of any value above zero in a co-deployed run, a warning is raised oblivious of the configured bandwidth threshold. Two versions exist of *read_parameter_space.bash* for backwards compatibility. A YANG model instance dictates the version and which script is executed. If a SSH configuration for a remote host was given, all result files are uploaded to the host. Since the creation of testbeds take a considerable amount of resources to create, it suggests itself to define a number of experiment instances and process all of them automatically. The script *iterate_parameter_space.bash* accepts a directory path as argument and processes any number of model instances in the directory by loading each model, generating appropriate application configuration files and the necessary input for the script *read_parameter_space.bash*. While this process supports mass processing of the parameter space depicted in Figure 2.5 and production of experiment results for further inspection, researchers might want to exert more control and get direct feedback. Researchers can use the system as a debugger, by attaching to the created tmux [Mar] session to check the output of experiment runs in process. Furthermore, for complete verbosity and control, a user can open a SSH connection to the controller domain to run control applications manually and access switches and endpoints to generate any desired form of traffic within the testbed.

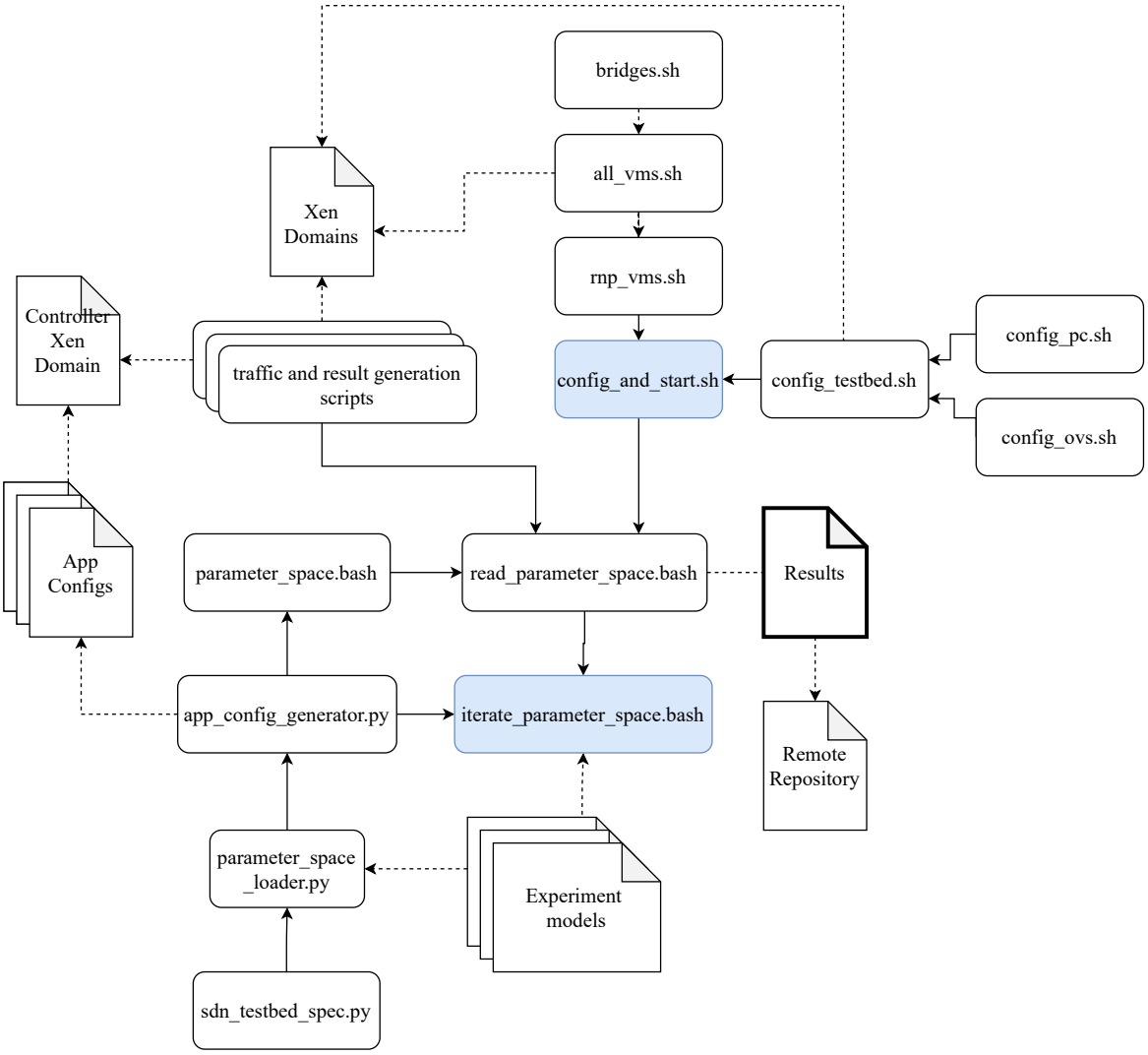

Figure 4.4: Overview of software modules required for experiment runs. The architecture consists of implementations by Tran [Tra22] and auto generated scripts (all_vms.sh, rnp_vms.sh and bridges.sh) from the MNM-Team [MT]. Adaptions and extensions are described in this section. Blue elements mark the entry points for topology deployment as well as single or batch experiment runs.

4.5 Example Deployment

This example shows how to create an experiment model, build a testbed and run experiments. Next to this step by step guide, the repository [TR] contains README files as a guide, which are also shown in the attached directory tree 8.2. The required packages for this example are the following:

- qemu-utils
- python3
- python3-pip
- pyangbind (pip3 install pyangbind)
- networkx (pip3 install networx)
- pygraphviz and utilities (sudo apt install graphviz libgraphviz-dev pkg-config && pip3 install pygraphviz)
- Linux kernel version 4.9.0-13-amd64 with Xen [TLF] hypervisor extension for local topology
- Virtualbox [OC] version 5.2.42 or later for encapsulated topologies

Unless a dedicated machine for executing this example is available, which can be downgraded to the required kernel version and provisioned with the Xen hypervisor [TLF], it is recommended to build an encapsulated topology with VirtualBox [OC]. If you choose to build a local topology, be informed that the directories */xen* and */control* will be created on the local file system and that any contents could be overwritten. References to the project repository and VM images are also required and are part of the work of Tran [Tra22]. They are accessible through the project repository [TR].

The first step is the creation of an experiment based on the YANG module. Open a command line terminal in the project repository and execute commands:

```
$: cd topogen/autogen
$: touch sample_topology.py
```

Use any desired text editor and create a model. The below code is an example experiment with four switches and four hosts as a testbed, illustrated by Figure 3.2. We deploy the Host Shadower and Path Load Balancer applications. The code shows the use of a model for a custom application that can be provided by a researcher. The *someApp* application needs to be provided by a researcher, either based on a Openflow REST interface as a Python module but independent of any controller framework, or as a Ryu [Nip] control application. In this example file has to be named *someApp.py* and copied to the directory controller/massive of the repository. In this example we assume that the application does not actually use the generated config file from the model, but it still provides a configuration to declare interest for hosts pc1 and pc2 as well as switch router1. A definition of a cookie, which should be used for flow entries as well as a declaration of at least one target switch, is mandatory. If you do not want to provide a custom application, comment out the respective lines in the code. Below the application configurations, we add hosts, switches and edges to the topology and draw a diagram.

```
import os
import json
import random_topo
import pyangbind.lib.pybindJSON as pybindJSON
from pyangbind.lib.serialise import pybindJSONDecoder
from sdn_testbed_spec_v2 import sdn_testbed_spec_v2
from constants_generate_spec import TOPOLOGY_DIR

topoId = "sample_topology"
topoFileName = os.path.join(TOPOLOGY_DIR, "".join([topoId, ".json"]))
model = sdn_testbed_spec_v2()

model.testbed._set_autostart("true")
model.testbed.topologyId = topoId

# Host Shadower config
hs = model.testbed.apps.add("hs")
hs.config.cookie = "0x440"
hs.config.targetSwitches.add("router2")
hs.config.targetSwitches.add("router3")
# add(1) is just an initializer and has no further meaning
appAssetsConfig = hs.config.appAssets.add(1)
appAssetsConfig.majorAsset.assetKey = "frontend"
appAssetsConfig.majorAsset.assetValue = "pc2"
appAssetsConfig.minorAssets.assetKey = "backend"
appAssetsConfig.minorAssets.assetValue = "pc3"

# Path Load Balancer config
plb = model.testbed.apps.add("plb")
plb.config.cookie = "0x300"
bw = plb.config.appInvariants.add("bw_time")
bw.intValue = 5
ts = plb.config.targetSwitches.add("router1")
bw_thres = ts.switchInvariants.add("bw_threshold")
bw_thres.intValue = 10

# configuration for custom app
# which does not use config file but
# provides config for automated results
someApp = model.testbed.apps.add("someApp")
someApp.config.cookie = "0x200"
someApp.config.targetSwitches.add("router4")
appAssetsConfig = someApp.config.appAssets.add(1)
appAssetsConfig.minorAssets.assetKey = "hosts"
appAssetsConfig.minorAssets.assetItems = ["pc1","pc2"]
appAssetsConfig = someApp.config.appAssets.add(2)
appAssetsConfig.minorAssets.assetKey = "switch"
```

```
appAssetsConfig.minorAssets.assetValue = "router1"

# set traffic profile and types
model.testbed.trafficTypes = ["udp", "tcp"]
model.testbed.trafficProfiles = ["cbr", "vbr", "bursty"]

# add switches and hosts, set host 1 as traffic source
for i in range(1,5):
  model.testbed.switches.add("router{}".format(i))
  pc = model.testbed.hosts.add("pc{}".format(i))
  if i == 1:
    pc._set_source("true")

# set the edges between nodes
# hosts need to be connected to one switch
# switches are connected to a switch or host
router_edges = [["router1","router2"],
    ["router1","router3"],["router2","router3"],
    ["router2","router4"],["router3","router4"]]

# draw the switches, hosts not supported yet
agraph = random_topo.create_agraph_fromedges(router_edges)
random_topo.draw_switches(agraph, topoId)

host_edges = [["router1","pc1"],["router4","pc2"],
    ["router4","pc3"],["router4","pc4"]]
edges = host_edges + router_edges

for i in range(len(edges)):
  edge = model.testbed.edges.add(i)
  edge.nodes = edges[i]

# write yang model to json in ietf format
output = pybindJSON.dumps(model, mode="ietf")
print(output, file=open(topoFileName, "w"))
```

Listing 4.2: Sample configuration for an experiment on Software Defined Network conflicts, including specifications for the network topology shown in Figure 3.2. Source hosts generate traffic, all other endpoints start mock servers for services. The current implementation supports automated generation of network diagrams for switch nodes.

To generate the experiment JSON file, run the following command within the directory topogen/autogen of the repository.

```
$: python3 sample_topology.py
```

The directory topogen/topology contains experiment models for all topologies listed in Section 3 with various app configurations and can be used for generating a JSON model instance

for this example. Adjust the constants in the file *constants_generate_spec.py* to the local file system, provide absolute paths to template images and a path to the optional SSH config and cryptographic key-pair. To generate the specification files for building the topology and any app configuration files, execute the following command within the directory topogen/autogen:

```
$: python3 generate_sdn_spec.py path/to/sample_topology.json
```

This produces specification files in the directory topogen/specs and will show the absolute path on the command line. Be sure to check for the correct path to the generated specification files. Now move to the directory topogen/scripts and execute one of the following command to build an encapsulated topology:

```
$: sudo deploy_sdn_conflicts_experiment.sh ../specs/sample_topology
```

You will be prompted for the amount of processors that can be allocated to the VM and a number to create a local port forwarding for SSH access to the VM. The required SSH command will be printed on the command line.

For deploying a Xen [TLF] topology locally, execute the alternative script:

```
$: sudo build_sdn_conflicts_experiment.sh ../specs/sample_topology
```

Root privileges are required for both scripts,since the implementation mounts images to the local file system.

For a successful deployment within a Virtualbox [OC] VM, it can now be accessed. Since the experiment model contains the autostart attribute, no further steps are required. As soon as a tmux [Mar] session is created, the experiment run has started. Switch to the root user and access the the VM via SSH with the following command if your provided port was e.g. 2226:

```
#: sudo -s && ssh -p 2226 localhost
```

Now check if a tmux [Mar] session has been created and attach to it.

```
#: tmux ls
#: tmux attach -t ID
```

For a locally deployed Xen [TLF] topology check, for any new tmux [Mar] session with the above commands and attach to it. If you provided a SSH config, any results will be uploaded to the target machine within the directory *sdn_results*. If you want to start more experiment runs, create more experiment models, and insert the JSON files into the encapsulated VM if necessary. From there move to the directory /control/automating_experiment and execute the following command:

```
#: bash iterate_parameter_space.bash path_to_json_files
```

This will only work for models that fit the generated testbed. The testbed is not recreated, but the application configurations are based on nodes in the testbed and will only be valid for these nodes. For every JSON model instance the script *app_config_generator.py* will generate app configuration files and any required global settings for an experiment run. This process is used for experiment building, as well as experiment deployment and the script will be imported automatically where needed. The script can also be used to automatically

generate configuration files from an experiment model for a manual execution of the Ryu [Nip] controller and control applications. The experiment results are saved to the directory */control/automating_experiment/dataset* or on the configured remote machine under */home-/user/sdn_results*.

When building a new testbed, be sure to tear down an established topology in order to free up needed resources and avoid deploying any remnant Xen [TLF] domains in case of a local deployment. This also applies for removing testbeds after finishing experiments. For an encapsulated topology, simply stop and delete the wrapper VM. To determine the name of a wrapper VM, try either of the following commands.

```
#: vboxmanage list vms
#: vboxmanage list runningvms
```

Be sure to collect any generated results from */control/automating_experiment/dataset* if you did not supply a valid SSH configuration. To stop and delete the VM, execute both commands:

```
#: vboxmanage controlvm sample_topology-1633001718 poweroff
#: vboxmanage unregistervm sample_topology-1633001718 --delete
```

In case of a local topology, the control folder needs to be deleted and the created Xen [TLF] domains need to be destroyed and deleted:

```
#: rm -r control
#: /xen/rnp_vms stop
```

Check if all Xen [TLF] domains have been removed:

```
#: xen list
```

If only the domain with id 0 shows in the list, all files can be deleted:

```
#: rm -r /xen
```

The presented approach promotes sharing of topology designs between researchers. It integrates a universal set of tools that maintain close control over technical details, while automating mundane tasks that result in configuration errors and increased workload for topology setup.

5 Distributed Conflicts

The results presented in this chapter are the product of 403 experiment rounds on the designed topologies 3.2 (15), 3.3 (129), 3.4 (170), 3.6 (22) and random topologies 3.10 (12), 3.8 (17), 3.11 (19), 3.12 (19). Each experiment round comprises at least three experiments for two applications and at most 541 for five applications with one configuration. This includes repetitions to verify the validity of the results. Although OpenFlow 2.2 is the SDN framework chosen for this approach, the presented results are not bound to the OpenFlow [ONFb] specification. Conflicts are determined with the procedure described in Section 3. This includes the comparison of data plane state when running applications in isolation and concurrently. The number of flow entries an application deploys is noted and compared to the number of flow entries it deploys when several applications are run together. The second criteria concerns endpoint reachability and the number of successful TCP and UDP connections established on a host. In case of a deviation between isolated and co-deployed runs, the deviation is noted. A comparison of link bandwidths is the last considered criterion, however it is important to note that the threshold for logging a conflict due to deviating bandwidths is part of the experiment configuration. All criteria represent symptoms indicating a possible yet unconfirmed conflict. This means that correlation between the criteria need to be considered to determine the existence of a conflict. While bandwidth deviation by itself might not lead to an unexpected data plane state or a change in endpoint reachability, it can still indicate a conflict. A preliminary, unproven attempt at a formal notation for conflict patterns of the presented conflicts classes is included in the Appendix 8.2 and serves as a guideline for the implementation of detection algorithms 6. Any presented conflict classes are defined in the following context.

5.1 Definitions

Packets are part of a data stream between two endpoints. In this implementation only IP packets are considered for the definition of conflict patterns.

Actions can be either packet transformations, e.g. the adaption of a specific property of coherent packets, the output to a link or subsequent flow table, or no action at all with the consequence of a packet drop. Transformations are always followed by an output action. Therefore, an empty action set denotes a drop, a single action denotes an output to a link and an action set larger than one denotes the existence of packet transformations with an output action. A transformation action can be described as a mapping $a \rightarrow a'$, where $a \neq a'$ and a, a' are instances of the same property of a packet, e.g. a destination IP address. For any action that transforms packets of a bi-directional flow, there must be a reverse transformation $b' \rightarrow b$ of returning packets, in our example a transformation of the source IP address, for successful communication between two endpoints.

Match fields as described in Section 2.2, are layer 2 and 3 addresses and layer 4 ports,

the port number assigned to an incoming link in a switch. When determining a correlation between two sets of match fields, there are three characteristic relations. An address based field can be disjunct to, equal to or a subset of another address. The relations are described in detail in Section 2.3.1 and are first mentioned by Hamed et al. [HA06]. Ports and IP protocols can be equal or disjunct. Fields like the ingress port of a rule cannot be compared for flow entries in two separate switches. Any further match fields are exempt since they cannot be covered by the experimental approach yet. Since the three types of relations still apply to further match fields, the conflict patterns apply to any related match fields.

Rules are flow entries, consisting of an indicator of it's originating application, a set of match fields and a set of actions.

Target switches are the switch nodes of a topology in the decision domain of a control application.

Paths consist of any number of connected rules in subsequent flow tables within a target switch or between target switches. That said, paths are not given by a list of switches but by a list of rules residing in the switches.

Rule graphs consist of any number of paths where the collection of unique switches are the same as in the physical topology, but the number of links are made up of connections between rules, not connections between switches, and are directed. Figure 6.1 depicts such a rule graph and the underlying physical topology.

5.2 Invariant Contention

Input by the inducing applications is required to detect Invariant Contention conflicts. The conflict class is showcased by at least two applications that enforce their policies based on the same invariant such as bandwidth. Furthermore, the conflict arises if a preceding application influences the invariant of a subsequent, even if the implementation of the prior is oblivious of the invariant. Although these are all implementations that contend on bandwidth or influence bandwidths, depending on the given topology, the pattern can apply to other invariants such as the maximum transmission unit. The shape of flow entries in the target switches disclose no interdependence of the invariant. Consequently, only the presence of each applications rules is detectable but not how app functions contradict. The flow and port statistics do provide details on bandwidths and if input is given, we could check if any packets travel on paths that are handled by both applications. As stated in the introduction of this chapter, the effect can be intended or represents a conflict. The conflict is best exemplified in the two following distinct edge cases.

5.2.1 Dispersion

The Path Load Balancer (PLB) and the Firewall (FW) application are both operating on the basis of bandwidth on ports of their target switches and elicit a Dispersion conflict.

Property. A conditional change of rule deployment by one application leads to a different decision by another subsequent application. Match fields in the decision are the same

across all conditions, but actions and egress ports can differ.

Effect. Failure to enforce application intent, e.g. packet filtering and failure to balance loads on paths as expected.

Example. We consider the topology and applications depicted in Figure 5.1. The Destination based Passive Path Load Balancer (PPLB4d) application is configured to balance traffic for the destination endpoints 1-4. The Firewall application is deployed with varying configurations for maximum port and flow bandwidths. The dashed line illustrates the single path all flows take when running an experiment with the Firewall application only. The dotted paths illustrate how bandwidth of the dashed path is dispersed when adding the Passive Path Load Balancer with said configuration.
For one thing, this clearly shows why a bandwidth based filtering application is non-deterministic at best and entirely unreliable in the worst case when being deployed with applications that influence bandwidth. In effect, traffic that was blocked without the PPLB4d is not blocked anymore, depending on port and flow threshold configuration in FW. The same conflict can occur when deploying the PLB instead of FW, since PLB alternates its routing decisions based on port bandwidth thresholds. Dispersion is also an issue when deploying the Path Load Balancer with the Firewall, both of which make decisions depending on bandwidths.
Another scenario for this conflict class is an experiment with the Endpoint Load Balancer instead of Passive Path Load Balancer, depicted in Figure 5.2. Endpoint Load Balancer (EPLB) is oblivious of bandwidth but its decisions lead to a dispersion of it on two links instead of one.

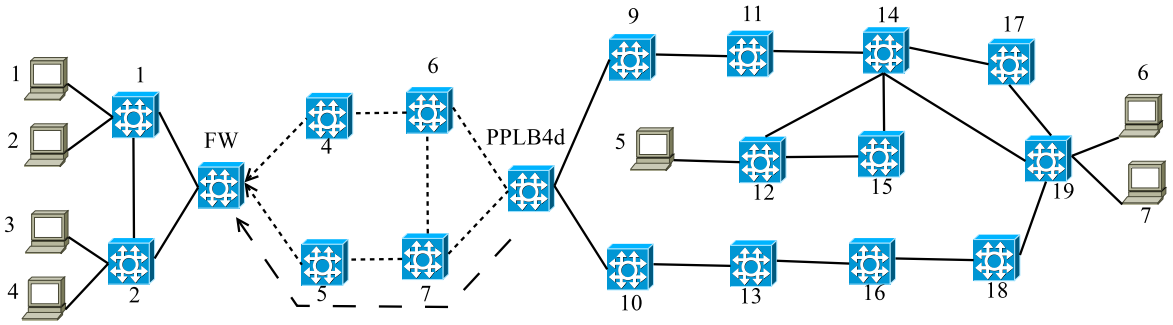

Figure 5.1: Example configuration for the Dispersion conflict class. The dashed path depicts the paths flows take when deploying the Firewall application in isolation. The dotted paths are the result of a combined experiment with the destination based Passive Path Load Balancer, where flow bandwidth gets distributed to all available paths leading to the target switch of the Firewall app.

5.2.2 Focusing

A Focusing conflict occurs when deploying the PLB or FW in conjunction with the Path Enforcer.

Property. A conditional change of rule deployment by one application leads to a different decision by another subsequent application. Match fields in the decision are the same

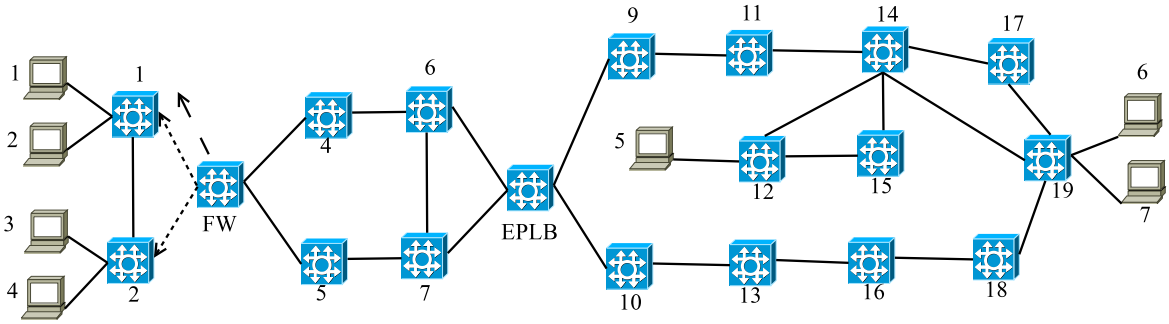

Figure 5.2: Example configuration for the Dispersion conflict class. The dashed path depicts the paths flows take when deploying the Firewall application in isolation. The dotted paths are the result of a combined experiment with the PPLB4d configured to balance traffic to endpoints 1-4, where traffic exits through two instead of one port.

across all conditions, but actions and egress ports can differ.

Effect. Failure to correctly enforce application intent, e.g. by unexpected packet filtering and balancing of loads on path multiple paths.

Example. This conflict class is caused by the opposite effect compared to dispersion. One or several applications channel traffic to a single or small number of routes. Any bandwidth oriented applications on the route are confronted with different conditions as compared to isolated deployment. The configuration depicted in Figure 5.3 is chosen to serve two purposes. The Path Enforcer directs any packets flowing from hosts 1-4 to hosts 6-9 to the central path. This decreases bandwidth on alternate routes and reserves them for backward flows to hosts 1-4. The bottleneck switch is suitable for e.g. a deployment of the Firewall or the Path Load Balancer application. Since the dotted path leads to a concentrated bandwidth in comparison to the dashed paths that are the default choice for any shortest path routing, the applications on the bottleneck node alter their behavior.
Similar to the alternative scenario for the Dispersion conflict, Focusing can also occur due to applications that influence the bandwidth of outgoing ports and any applications that are deployed on the receiving end of respective links. Figure 5.4 illustrates a conflict between the Host Shadower and the Path Load Balancer application. The Host Shadower is configured to redirect traffic targeted at endpoint 8 to endpoint 6. This results in an increased bandwidth on the dotted path, whereas the bandwidth is split equally on the dashed paths for an isolated deployment of the Path Load Balancer. Both switches connected to the target switch of the Host Shadower exert a different behavior in co-deployment of both apps.

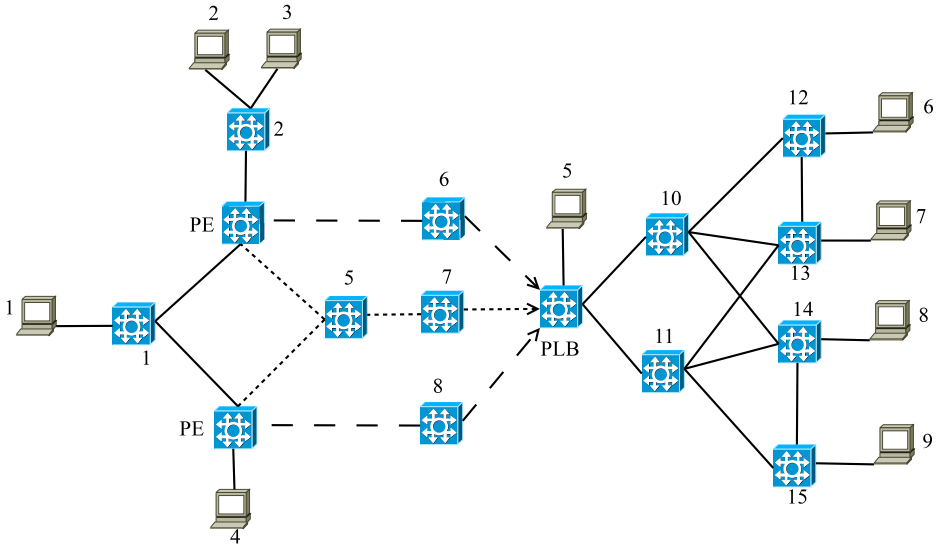

Figure 5.3: Figure illustrating a conflicting interest between the Path Enforcer and bandwidth oriented applications, resulting in a Focusing conflict. The dashed paths are the shortest routes from hosts 1-4 to hosts 6-9 and vice versa. The dotted route shows how the Path Enforcer is deployed to offload the traffic going out from endpoints 1-4 to the central path, which can inadvertently alter the behavior of bandwidth based applications in the bottleneck switch.

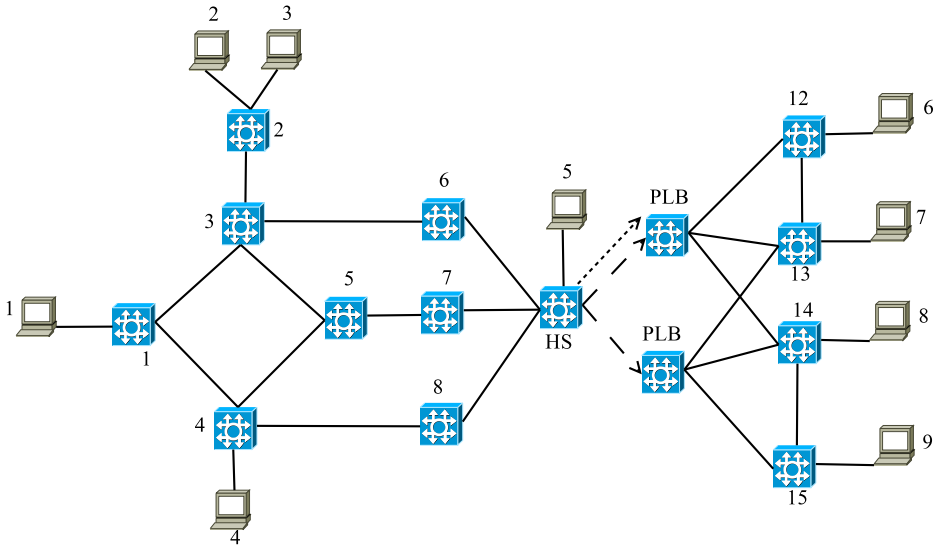

Figure 5.4: Focusing conflict caused by the Host Shadower application that channels the output bandwidth on its target host to one exit port to the dotted link. This induces a behavior in bandwidth based applications such as the Path Load Balancer, which receives bandwidth on two dashed paths when deployed in isolation.

5.3 Transformation Contention

This general conflict type relates to the transformation actions that can be performed in switches on passing packets. The observed conflicts are based on transformations of address based match fields of a flow entry and result due to multiple applications that claim interest on the same hosts or an intersecting subset of such. Again, these conflicts can emerge due to intended behavior. Input on application intent is needed to decide if these are relevant conflicts. Nevertheless, the following two conflict classes as such are relevant, and we can assume that they also apply to other transformations affecting e.g. layer 4 ports or even the Type of Service field of IP packets.

5.3.1 Occlusion

Occlusion is displayed be the absence of a rule deployed by an application on a rule path or a deviating decision, compared to an isolated deployment of said application, due to a packet transformation on an earlier node in the rule path by a different application.

Property. A previous application on a rule path transforms packets such that a subsequent application fails to enfore a policy that is seen during isolated deployment. The transformation adapts packet properties in such a way that the subsequent application fails to register the match fields and does not enforce a policies based on the original match fields, leading to a different or no decision.

Effect. Failure to enforce application intent, e.g. balance traffic on endpoints or links.

Example. Let us consider a deployment of both the Host Shadower and Endpoint Load Balancer as depicted in Figure 5.5. Both applications transform MAC and IP addresses of a packet. The Endpoint Load Balancer is configured to balance any traffic to host 2 between the hosts 1-4, indicated by the dashed paths. This happens whenever hosts 5-7 direct traffic at host 2. The Host Shadower redirects traffic destined to host 2 to host 1, indicated by the dotted path. When deployed together, this results in complete inactivity of the Endpoint Load Balancer were no flow entries get deployed or any existing entries are not applied anymore, effectively voiding the dashed paths.
The second scenario in Figure 5.6 features the Endpoint Load Balancer and Host Shadower app in reverse deployment, yet with the same configuration of endpoints-combinations. In isolated deployment, all traffic directed at host 2 flows on the dashed path towards host 1. In co-deployment, the Endpoint Load Balancer rewrites some packets of the traffic directed at host 2 to target hosts 3 and 4 instead. This will not render the Host Shadower entirely inactive, but it will deploy less flow entries than before in such a way that traffic will be distributed to all dotted paths. The main issue of this conflict is the actual lack of any artifacts that let us detect it. It is not a dispute between existing flow entries in the data plane but rather one between a present entry and a potential policy that is partially or completely suppressed. This can also be seen as a distributed hidden conflict. Considering manually engineered firewall policies in a switch, a packet transformation can disguise packets to pass a firewall that would reject the unaltered packets.

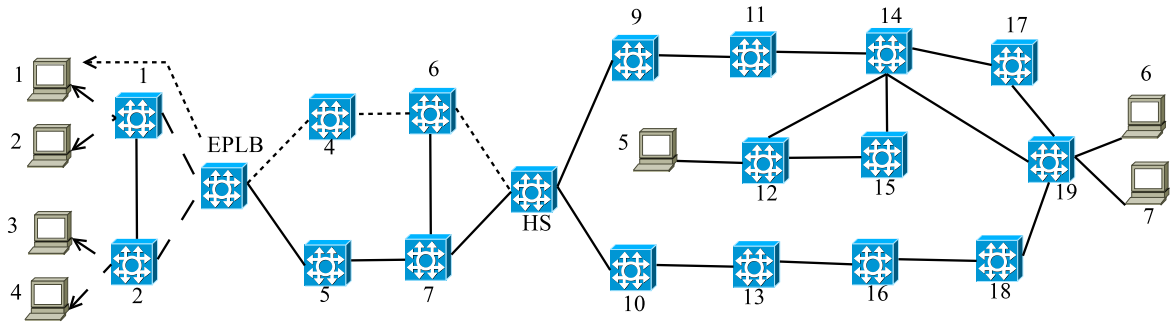

Figure 5.5: In this Occlusion conflict scenario, the EPLB application is configured to balance traffic targeted at endpoint two to all endpoints depicted on the left side. If the Host Shadower application is configured with endpoint 2 as frontend and endpoint 1 as backend node, leading to the dotted path, the EPLB does not deploy any flow entries for the dashed paths.

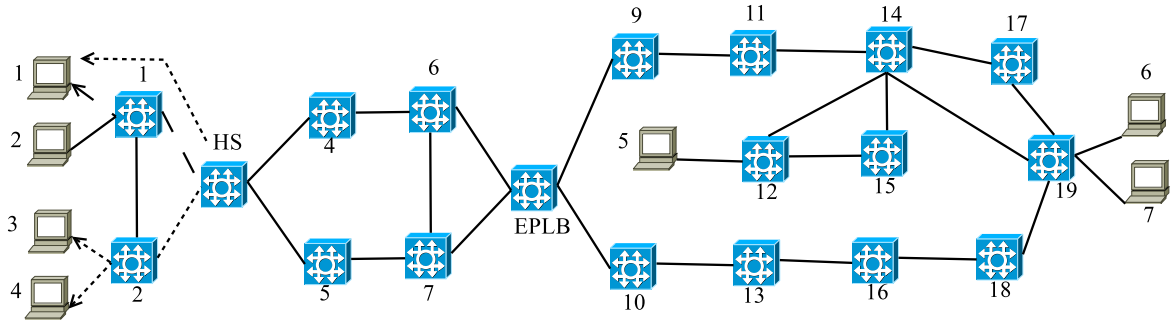

Figure 5.6: Example of Occlusion caused by the EPLB application. The Host Shadower redirects all traffic targeted at endpoint 2, show by the dotted dashed path, to endpoint one, indicated by the dashed path. The EPLB is deployed to balance loads targeted at endpoint 2 to endpoints 3 and 4, shown by the dotted routes. When co-deployed, the Host Shadowing app does not deploy any or a smaller number of flow entries for the dotted path.

5.3.2 General Multi-Transform

General Multi-Transform conflicts relate to Multi-Transform conflicts as defined by Hamed and Al-Shaer [HA06], although in a broader scope and not limited to security transformations. This can include circumvention of filtering applications and unintended superposition of packet transformations. The work on the tool FortNox describes such firewall circumvention scenarios [PSY+12] that can be correlated to this conflict class.

Property. Multiple transformations of packets in a flow through several apps. It occurs when the transformed output packet of one application is matched by a flow entry of a second app in a subsequent target switch and packets are again transformed.

Effect. Superposition and violation of application intent and e.g. ambiguous routing to endpoints.

Example. Figure 5.7 demonstrates such a conflicting configuration between the Endpoint Load Balancer and Host Shadower. The EPLB app is configured to balance traffic between hosts 2 and 3 if host 2 is targeted and between hosts 6 and 7 if host 5 is targeted. The dashed paths illustrate the flows when the EPLB is deployed in isolation. When we add the Host Shadower to the experiment and configure it to rewrite packets destined to hosts 3 and 7 to be sent to endpoints 2 and 6 respectively, indicated by the dotted paths, traffic flows as if neither the Host Shadower (HS) and the EPLB app are active. Their conflict of interest and multiple transformation reinstate the original state of the packets as sent out by the source endpoints. The same applies to the reverse transformations back to the sources.
An alternative configuration is shown in Figure 5.8, where the interests of the Host Shadower application is violated by a subsequent packet transformation of the EPLB app. The Host Shadower is supposed to channel traffic for endpoints 6 and 7 to host 6 only. The EPLB enforces a contradicting policy that balances all traffic directed at host 6 to all available endpoints on the right side of the figure.

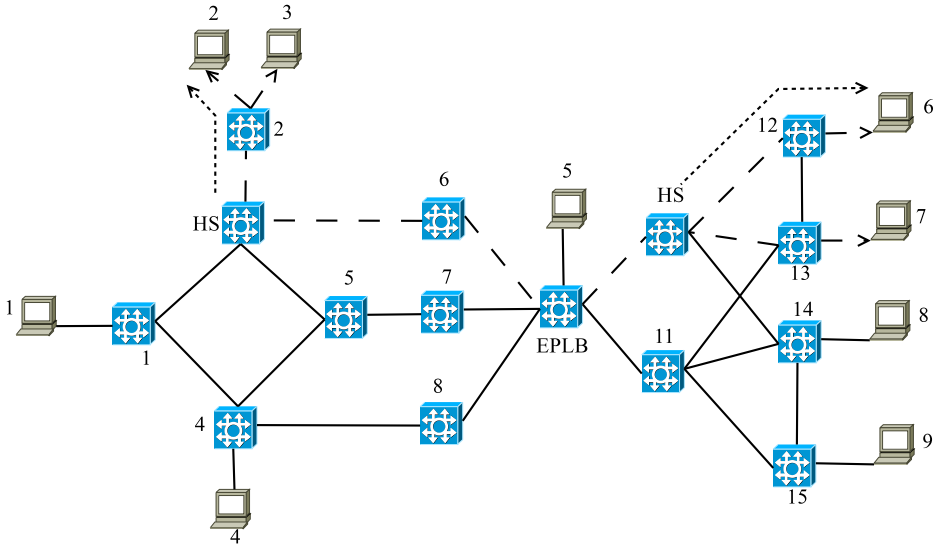

Figure 5.7: Illustration of a General Multi-Transform conflict. The dashed paths illustrate that the EPLB app transforms traffic directed at hosts 2 and 6 to be balanced to hosts 3 and 7 respectively. The dotted paths indicate where the Host Shadower adds an additional transformation, which negates the transformation by the EPLB and result in traffic that is only directed at hosts 2 and 6.

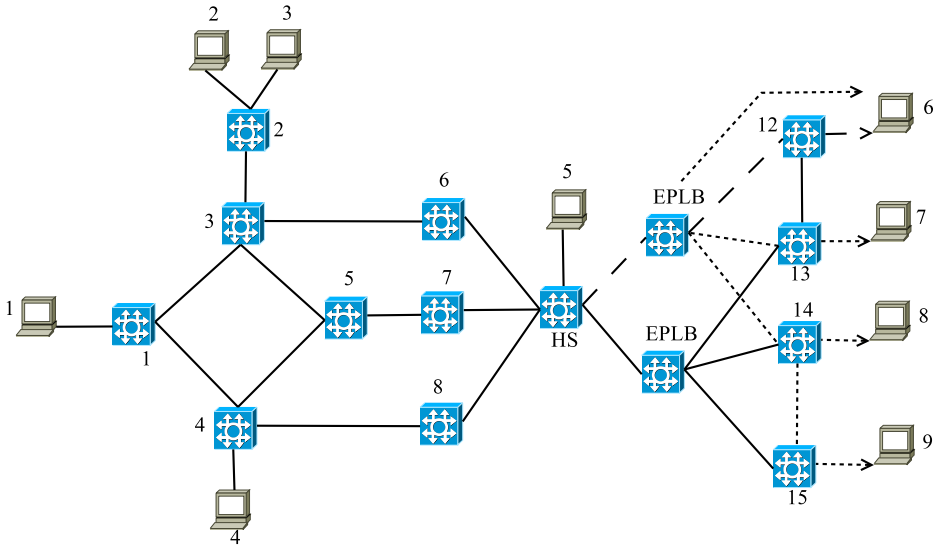

Figure 5.8: This figure presents a conflict scenario for a General Multi-Transform conflict that results from multiple packet transformations. The Host Shadower application is configured to redirect any traffic targeted at host 7 towards endpoint 6, shown by the dashed path. The EPLB is set to balance any traffic for host 6 between hosts 6-9, indicated by the dotted paths.

5.4 Spuriousness

Inter-policy Spuriousness as mentioned by Hamed and Al-Shaer [HA06] is a conflict class in SDN too. It is denoted by the following property.

Property. Packets entering a local network are filtered on an intermediate switch. A policy on one switch forwards packets to another switch where packets are dropped.

Effect. The invalid packets are discarded but infiltrate part of the local network, which affects security and unnecessarily increases network load.

Example. Figure 5.9 demonstrates two alternative configurations for deployment of the Firewall application. Endpoints 6 and 7 are used to simulate traffic from a remote network where the dashed paths indicate a connection of the target switch FW I to a public network. In absence of FW I, the placement labeled with FW II exerts the Spuriousness conflict pattern indicated by the dotted paths because it fails to prevent undesired packets from intruding the local network. A drop action on a switch at the edge of a local topology is not considered a conflict, as we have to assume that it is the intended behavior. However, drop actions that lead to Spuriousness can be seen as conflict since it creates an unnecessary network load with unauthorized packets. While the pattern can help in optimizing the deployment of a filtering app, incomprehensible bandwidth is only a symptom, and it depends on the desired configuration if this is truly a conflict. We do not discern between partial and complete Spuriousness as defined by Hamed and Al-Shaer [HA06] since there is no apparent reason to discern between the security impact of the amount of unwanted traffic that traverses a local network.

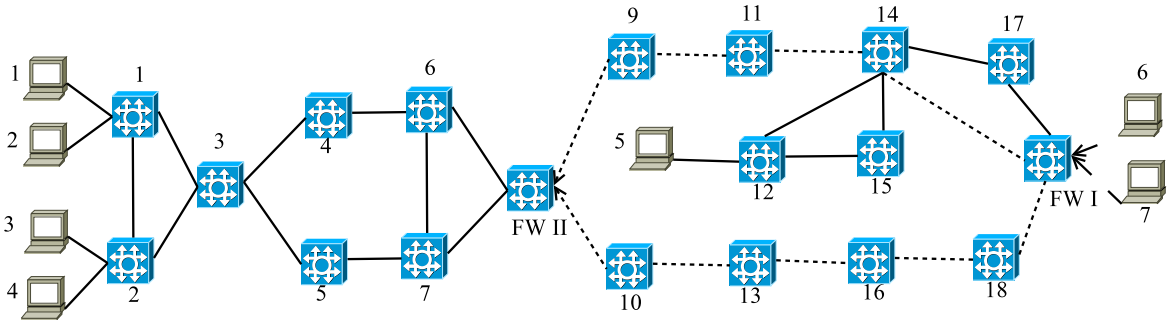

Figure 5.9: Demonstration of a Spuriousness conflict, indicated by a placement of the Firewall application on the target switch FW II and the dotted paths. Endpoints 6 and 7 represent machines that connect through a public network on the dashed paths in this experiment, and FW I is the deployment that prevents undesired traffic from entering the local network.

5.5 Loops

The Loop conflict pattern is the standard problem in routing scenarios of trapped packets between certain links. The loop is rather a symptom that emerges from an uncoordinated

routing decision by multiple control applications. The central control instance represented by the controller is the predestined mechanism of avoiding the loop, but due to the lack of repeated packet-in messages this has to be a passive implementation based on a rule graph 6.1 or other methods.

Property. Any rule that appears twice in a rule path with a path that contains intermediary rules between those appearances can be considered a loop. This also covers the definition of a short loop between two adjacent nodes in the topology.

Effect. Packets fail to reach their intended destination and are trapped within the network. Network load increases substantially.

Example. The current implementation for loop detection in the Detection Prototype successfully exposed a weakness in the Path Enforcer application in the experiment depicted in Figure 5.10. The Path Enforcer (PE) app was configured to route any TCP and UDP traffic through the node 7 and then to the target switch of the PLB app. In a combined simulation with the PLB app, this lead to a short loop between switch 7 and its target switch. This again raises the issue that the Path Enforcer fails to insert any policies on the last jump of its routes. Yet in this instance that would evoke a local conflict and depending on the order of priorities between the apps, could produce the same result. Let us consider another theoretical example involving the same applications, assuming that the Path Enforcer inserts policies on the last jump node from source to destination, as illustrated in Figure 5.11. Firstly, we can note that three applications are actually part of forming the loop since the Routing app is deployed and active on all switches, in particular on switch 7. The PLB app deployed on the bottleneck switch will alternate its routing decisions between all paths on the left side for traffic that targets host 2, and in combination with the Routing app, the dashed path is a possible result. If the PE app is deployed as depicted in Figure 5.11 and configured to route traffic over switches 5 and 7, including the only sensible choice of an outgoing port from switch 7 to the target switch of the PLB app, the loop shown by the dotted path is completed and the packets are trapped. The clearly visible loop emerges if the PE inserts policies on its last jump and also if not, since the Routing app will then establish a short loop between the PLB target switch and switch 7, as seen in the first example 5.10. Consequently, that demonstrates that the remedy is not more control by the PE app on its last jump but rather a better handling of such edge cases in the first place.

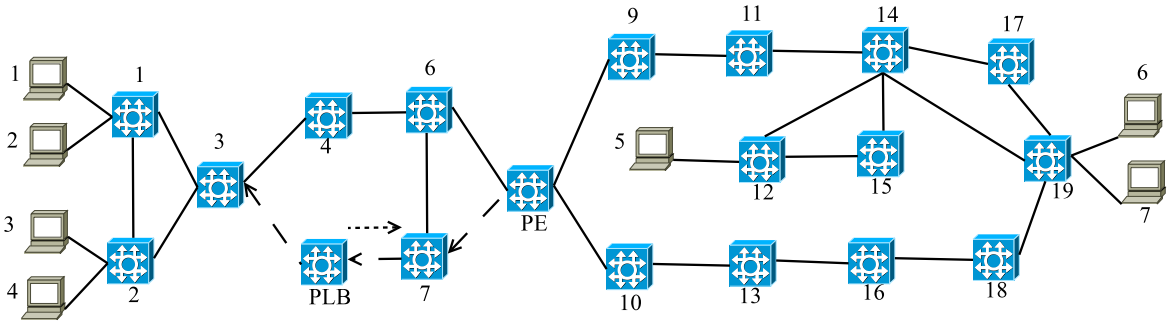

Figure 5.10: Depiction of an occurrence of a short loop when deploying the Path Enforcer and Path Load Balancer in a problematic configuration. The PE app routes traffic over switch 7 and the target switch of the PLB. The PLB alternates flows between a direct route to switch 3 and the routes over switches 7,6,4 and then 3. For some traffic, that results in the loop depicted by the dotted and dashed path going out of and back to switch 7.

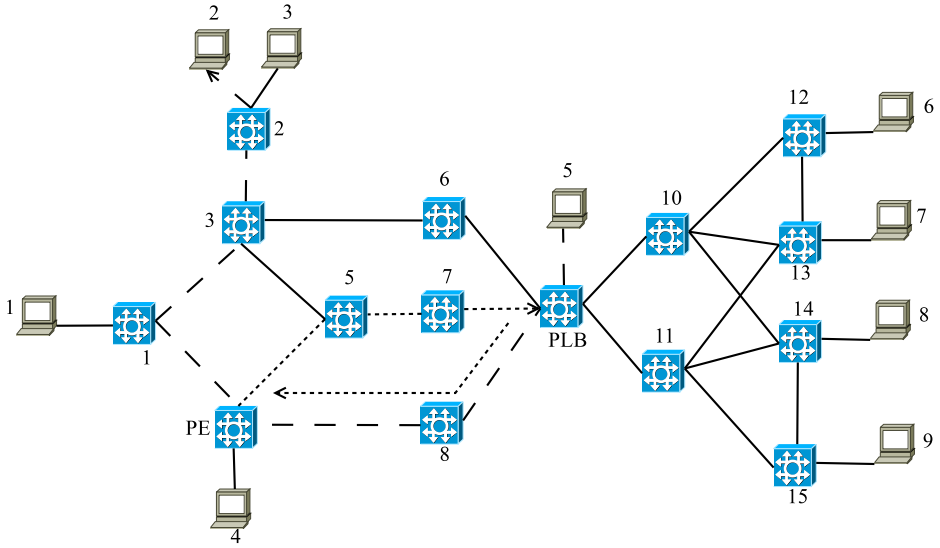

Figure 5.11: Theoretical experiment configuration that highlights a previously unhandled edge case in decision by the Path Enforcer. The traffic loop on the dotted path is a result of a Path Enforcer implementation that inserts policies on the last jump of its enforced route, here switch 7, in co-deployment with the PLB. The dashed route is the decision by the PLB and the Routing app, without the PE.

5.6 Path Ambiguity

Redundant nodes and links are part of any scalable network architecture addressing the requirement for resilience and availability. This entails the use of shortest path algorithms or other implementations to avoid loops in chosen routes. Control applications that are based on differing algorithms or decisions that ignore optimal paths can interact towards undesirable results in such topologies. In combination with applications that perform packet transformation and possibly erroneous or unsolicitous configurations, the following conflicts classes arise.

5.6.1 Incomplete Transformation

In case of multiple paths between endpoints and applications that perform packet transformations, packets on any reverse path need to be reverse transformed. For any action by these applications, several associated flow entries need to be inserted in the target switches, at least two if there is only one target switch.

Property. The Incomplete Transformation pattern arises if an application has target switches on any given path from a source to a destination and performs packet transformations and back transformations on one path but fails to perform the back transformations on all paths. When another application routes any packets on the returning flow over an uncovered path, the returning packets are not transformed.

Effects. Returning packets fail to reach their destination or are discarded at the destination due to the missing back-transformation.

Example. Figure 5.12 presents a scenario of a missing back-transformation of packets that are altered by the Host Shadower, where the forward rule for the dashed path gets applied in the target switch. The returning packets are routed on an alternate path due to the influence of the Path Enforcer and the backward rule by the HS is not applied. If the flow entries are address based, it is easy to determine what the associated entry is. If we expand this pattern to any form of transformation, it is questionable how a reverse entry can be identified and how to avoid invalid associations.

5.6.2 Injection

Applications that route packets of the same flow on multiple paths, e.g. PLB or PPLB4d as well as the PE that routes packets over a specific path, are prone to injecting said packets to transformation policies on such a path. This can be intended behavior, but especially in case of extreme routing decisions, e.g. very inefficient paths, and in combination with unintended transformations, this can indicate a deficient configuration. While this conflict class was exposed because of unintended transformations of packets it might not be limited to such a scenario. We could image an application that routes packets over a path where rate limiting or filtering is applied, too.

Property. The conflict occurs in the presence of alternating routing decisions of packets in the same flow over several possible rule paths and a transformation of these packets

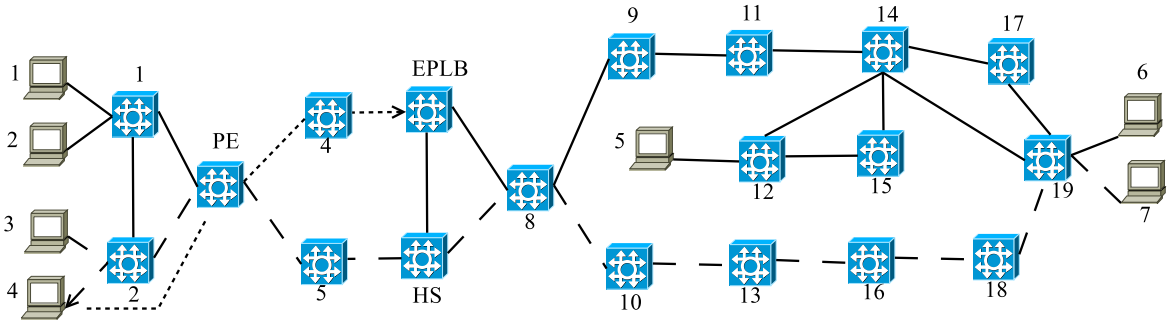

Figure 5.12: Configuration of an experiment that exerts the Injection and Incomplete Transformation conflict patterns. The Host Shadower redirects packets targeted at host 3 towards host 4 and needs to re-transform address based fields in packets on the reverse path. The Path Enforcer deploys policies that circumvent the HS on the reverse flow and induces an Incomplete Transformation. If the EPLB app is added in with a configuration that overlaps with the interest of the HS, the Path Enforcer evokes an Injection conflict in its target switch, resulting in an incorrect transformation of packets.

on only part of these paths. Consequently packets are transformed on one route, but not on the other.

Effect. Ambiguous application intents with packets possibly being dropped by the intended destination.

Example. The example depicted in Figure 5.12 is chosen intentionally to include more than two applications to highlight why an Injection conflict can occur in case of a deployment with all or only a subset of the applications. Let us consider an experiment without the HS application. Endpoint 7 establishes connections with endpoints 3 and endpoints 2. The flow for endpoint 3 will follow the dashed path, the flow to endpoint 2 passes the target switch of the EPLB app, which, given enough connections, will deploy flow entries that send part of the traffic targeted at endpoint 2 towards endpoint 3 and entries that rewrite the source addresses of packets to appear as coming from endpoint 2 on the way back. The flows on the dashed path towards endpoint 3 will be forced to the dotted path on the way back to host 7 and are transformed by policies of the EPLB app. The returning packets will be ignored and appear to be unsolicited as they now originate from endpoint 2. The scenario that includes the HS app can result in the same result, but the root cause is the Injection conflict by the PE app and not the overlapping interest of the two endpoint oriented applications. Furthermore, the same effects can be observed when adding the PLB on the target switch 8 or substituting the PE app with the PLB app on its target switch.

5.6.3 Bypass

Another conflict pattern in a topology with ambiguous routing options is the complete circumvention of target switches of an application. Traffic stops to flow through a previously used route or never actually traverses it. Applications that have inserted flow entries are deprived of bandwidth and fail to enforce their interest. This pattern does not lead to a

failure in connections, which makes it harder to detect and might be the intended behavior.

Property. Transformation actions of applications do not operate on at least one target switch on every distinct path from source to destination endpoint and another application routes matching packets over an uncovered path.

Effect. Deprivation of application intent.

Example. The experiment illustrated in Figure 5.13 exhibits a Bypass conflict. The PLB routes traffic on the dotted paths around the target switches of the Host Shadower. Any returning packets might traverse the Host Shadower's switches, but the flows will appear to be of no interest.

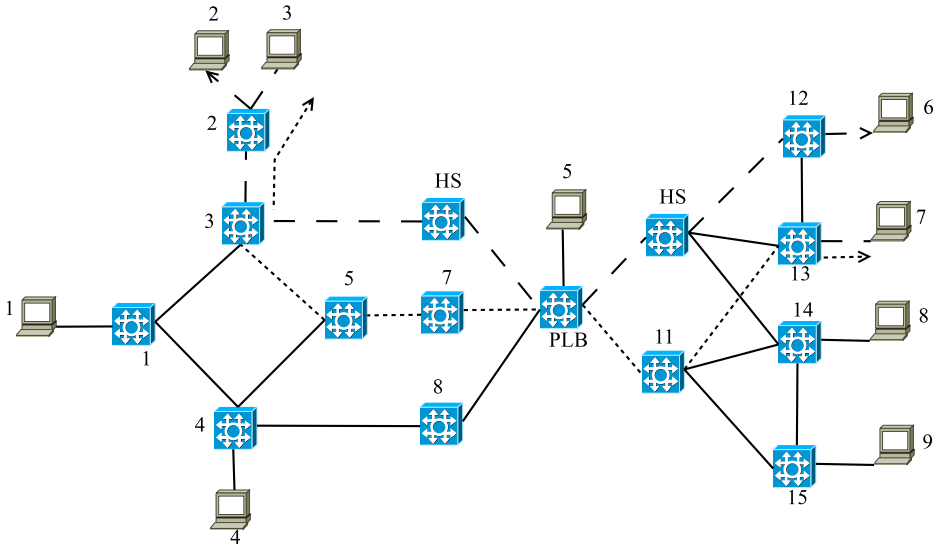

Figure 5.13: Experiment demonstrating a Bypass conflict with a deployment of the PLB application that leads to a circumvention of the target switches and/or the interests of the Host Shadower. Traffic flowing to host 3 or host 7 is transformed on the dashed paths, but the PLB routes traffic on the dotted paths that avoid such actions by the HS app.

6 Conflict Detection

Preemptive discovery of distributed conflicts is not possible without global knowledge of application intent, and communication of application intent is subject to interpretation. Detection of most conflict classes require a delay until all flow entries from source to destination have been inserted for unique flows. In case of general flow entries that are not unique to a connection, preemptive discovery is feasible, and conflict resolution can be performed if the interpretation problem of application intent is solved. Such policies include policies that target a range or all addresses and specify generic match fields only. Memorization of data plane state is necessary to determine if policies across target switches are connected, while avoiding latency for repeated statistic queries of said state. Target switches and connections imply a graph for tracking flows through a topology. Thus, a rule graph as devised by Tran [Tra22] is the data structure of choice.
Detection methods will only be applied to conflict classes that are not based on application intent, with the exception of the Occlusion class. This excludes any Invariant Contention and Incomplete Transformation conflicts. Effects of Invariant Contention conflicts as of yet are unclear and require detailed review. This review needs to include the correlations between divergence of data plane state and monitored metrics of respective invariants with possible negative effects. Also, a sound concept for communicating application interest in invariants is missing. This interest is obvious for the Firewall and path load balancing applications, however the Endpoint Load Balancer influences bandwidths too. Furthermore, in a combined experiment of the Path Load Balancer and the Passive Path Load Balancer the apps are still doing what they are supposed to do and connections are established successfully, yet data plane state and bandwidths are different for every run and every set of app configurations. This results in relevant deviations, but application effectiveness is upheld. The effects of Invariant Contention conflicts are not clear enough and a sensible interface for determining application intent is missing. Application interest in endpoints is concise, and the experiment model addresses the need for specifying endpoint interest, therefore detection of Occlusion conflicts is feasible. To detect Incomplete Transformation conflicts we also need to know application intent, since we can neither decide which direction is the up- or downstream flow or if the missing back-transformation on a specific switch is the intended implementation. Other conflict classes arising from ambiguous routing options are not dependent on application intent, and detection methods stay oblivious of any specific application.
In contrast to local conflicts, the immediate effects of distributed conflicts cannot be determined. This requires monitoring of and input by endpoints, which is not feasible in a production network. Therefore detection is based on the properties of conflict classes described in Section 5 without consideration of the effects. Most detected conflicts are *interpretative*, as defined by Pisharody et al. [PCH16] because they arise due to faulty configuration or desired inter-dependencies. Such desired inter-dependencies can only be validated by an administrator. The detection algorithms aim at providing administrators with the necessary information to discern desired effects from the unintended. The below thoughts for conflict

detection feature pseudo-code for new implementations and a description of any additional input for existing algorithms.

6.1 Prototype

The core element of the prototype is the rule graph devised by Tran [Tra22] and the following detection algorithms extend the function of the existing implementation. The experiments showed that a rule graph introduced significant latency to the process of flow entry modifications and insertions. Therefore detection is performed passively, and the prototype at this stage forwards any Openflow messages 2.2 to control applications before detecting conflicts. We choose OpenFlow [ONFb] for the implementation due to the maturity of the framework and as a means to demonstrate the correctness of the presented conflict classes. Figure 1.3 illustrates the role of the detection prototype within the SDN control layer and Diagram 3.14 and depicts the modules used by control and application layer implementations to serve the detection mechanism. The connective element for deferred detection of distributed conflicts is the module *utility_detector.py*. Next to providing methods that generate flow modification messages sent to nodes in the infrastructure layer, it additionally produces custom events to asynchronously notify the subscribing *detector.py* module. In this publish-subscribe design pattern, the custom ModFlow message sent by the utility module and flow removal messages generated by switches represent the entry point for maintaining a rule graph. This entails that a complete copy of data plane state is cached in the control layer. Diagram 6.1 illustrates an example rule graph for an infrastructure of four SDN switches. For any new rule that is inserted to a flow table, the outgoing port is determined. The link connected to the port indicates the next target switch for packets handled by the current flow entry. All entries in the flow table of the next hop are iterated to determine which ones connects to the current rule. Outgoing connections are limited to one target policy on each subsequent switch, because only one flow entry is effective in case of a local conflict. Every entry is mapped to a list of its properties, i.e. application cookie, priority, match fields and actions. Subsequent entries are compared to the previous entry after packet transformation actions are applied. Missing match fields are considered wildcard fields that match any packets. A trivial example matching anything would be an entry with only an ingress and egress port and no other match fields. Local conflict patterns described in Section 2 are computed from relationships between all match fields. Although local conflicts exert no direct implications for distributed conflicts, the outcome of the comparison is a prerequisite for determining which rules are connected for new insertions. Furthermore, relationships or correlations between match fields from connected nodes in the rule graph are determined with the same mechanism as local relations. Two nodes in a rule graph correlate, if their match fields are equal, any one is the subset of the other or if there is a partial, bidirectional overlap. Please refer to Table 2.1 for an example of match field relations.

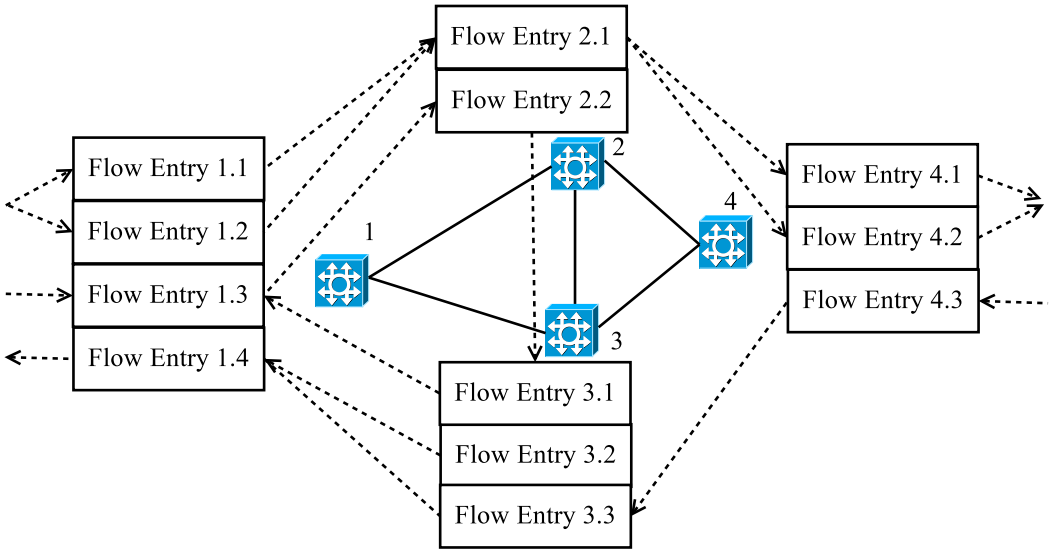

Figure 6.1: Showcase example of a rule graph for a topology with four switches maintained by the detection prototype. The graph contains a loop between entries 1.3, 2.2 and 3.1 and an unconnected entry 3.2. without an input rule or originating from packets due to an endpoint connected directly to switch 3.

6.2 Spuriousness

Policies that lead to dropping packets in general are already detected. This work makes a distinction between filtering of incoming traffic at edge a topology and inter-policy Spuriousness as described by Hamed et al. [HA06]. Filtering on an edge switch prevents malicious packets from entering the network. When packets are filtered after they have already traversed nodes within the topology, the definition of the Spuriousness conflict class is fulfilled. If any policy without actions is inserted to the rule graph there is effectively no output to a subsequent connected node. The incoming port of the flow entry is checked to determine if it is part of a link to another switch in the local network. Packets that are forwarded within the topology and dropped at an intermediate or edge node represent unnecessary, possibly harmful traffic. The current implementation by Tran [Tra22], which already detects flow entries containing drop actions, is extended to check if the flow entry is already connected to a rule in the rule graph.

6.3 Loops

Loop detection is based on the implementation by [TD20], complemented with the distinction between a loop in the topology and one within the rule graph. This allows for loop-like routes where packets pass through an already traversed switch, taking into consideration that it can be handled by a different rule that leads out of the loop on the second pass. Consequently, loops occur if a packets gets handled by a policy twice.

6.4 General Multi-Transform

A flow entry that applies any form of packet transformation is a potential candidate for a multi-transform conflict. The detection algorithm 2 is backwards oriented concerning the rule graph. For any new policy containing packet transformation actions, we determine any paths in the rule graph leading to this rule. The algorithm is applied recursively over all vertices in all possible input paths. Every vertex is checked for the existence of packet transformations and a warning for a General Multi-Transform conflict is raised for any positive cases. Since the algorithm 2 is backward oriented, we can raise a warning once we encounter two transformation actions on a packet flow. The starting vertex is passed to every recursive call of the algorithm and it starts with a check of the current ingress vertex and the root vertex. We need to break the recursion in case of a loop in the rule graph. Any additional transformations on a predecessor vertex were already considered in a previous call of the algorithm. A check for correlation of match fields in a flow entry is necessary to assure that the transformations affect the same packet flow. The necessity is exemplified by flow entry 2.1 in the sample rule graph 6.1. Let us consider flow entry 4.1 as the current rule entry that contains packet transformations and flow entry 1.1 and 1.2 as performing packet transformations, too. Flow entry 2.1 is the connective vertex, and both flow entries 1.2 and 1.1 are connected vertices. By applying the packet transformations of rule 1.1 and 1.2 on the match fields of the rules, we can determine any correlation to the match fields of rule 4.1. We want to avoid false positives for connected predecessor vertices that perform transformations but are not correlated by their match fields and transformations to the current candidate rule.

```
input: vertex in rule graph containing transformation actions, ingressVertex, app
if vertex == ingressVertex then
|  return;
end
rule ← getRuleFromVertex(vertex);
match ← getMatchFromRule(rule);
ingressVertices ← getIngressEdges(rulegraph, ingressVertex);
foreach inVertex in ingressVertices do
|  inRule ← getRuleFromVertex(inVertex);
|  actions ← getActionsFromRule(inRule);
|  inMatch ← getMatchFromRule(inRule);
|  inApp ← getAppFromRule(inRule);
|  foreach action in actions do
|  |  if action transforms match field and app ≠ inApp and match correlated with
|  |   inMatch then
|  |  |  raise Multi-Transform conflict with vertex and inVertex data;
|  |  else
|  |  |  recurse with vertex, inVertex, app;
|  |  end
|  end
end
```

Algorithm 2: Detection algorithm for General Multi-Transform conflicts.

6.5 Injection

False positives for the Injection conflict class are guaranteed in a configuration with multiple interdependent applications and without interpretation of application intent. The following implementation is based on two heuristics to avoid false positives. Flow entries with output actions only connected to policies by another application that transforms packet fields are considered. The second criteria is met if there are multiple paths between source and destination endpoint, and at least one path does not contain the switch that the considered candidate rule resides in. An Injection conflict occurs if there is an application that inserts policies on a target switch that is part of all paths and can therefore make ambiguous decisions concerning the generated path, and the two above criteria are met. A version of the algorithm is feasible, where no transformation actions are involved. This represents a general Injection conflict between two applications, e.g. the Path Enforcer application injecting flows into the Path Load Balancer. Since applications are intended to interact in SDN, this will produce a high rate of false positives, therefore we examine transformation related Injection conflicts only in the evaluation of this algorithm.
Below algorithm 3 illustrates that for any rule containing transformation actions, we traverse back in the rule graph for any input path. As with the algorithm for General Multi-Transform conflicts, we have to check if connected vertices correlate in their match fields. An actual check for the presence of a path in the rule graph and active policies that can circumvent the packet transformations is not performed, to avoid parsing of the entire data plane state of such a path in the network and the added computational complexity. Another method would be input by an application to determine if it would perform such an ambiguous routing decision. As with Multi-Transform conflicts, an early stop condition for any loops in the rule graph is necessary to avoid indefinite recursions. The justification for generating warnings for an Injection conflict without this check is given by the fact that ambiguous routes containing different action sets are likely produced by a faulty configuration. In an illustrative example, we consider flow entry 2.1 in the sample rule graph 6.1 and the connected rule 1.1. The source endpoint resides before switch 1 and the destination endpoint behind switch 4. If rule 2.1 contains packet transformations and there are no rules present in switch 3 with the same transformations, an application on switch 1 could route packets to the destination without the packets transformations being applied. The condition is that the application that produces rule 2.1 is not present on all paths to the destination and the injecting application is.

```
input: vertex in rule graph containing transformation actions, ingressVertex, app
if vertex == ingressVertex then
  return;
end
rule ← getRuleFromVertex(vertex);
switch ← getSwitchFromRule(rule);
match ← getMatchFieldsFromRule(rule);
ingressVertices ← getIngressEdges(rulegraph, ingressVertex);
foreach inVertex in ingressVertices do
    inRule ← getRuleFromVertex(inVertex);
    inSwitch ← getSwitchFromRule(inRule);
    inMatch ← getMatchFieldsFromRule(inRule);
    inActions ← getActionsFromRule(inRule);
    paths ← getAllSimplePathsFromMatch(inMatch);
    inSwitchInPaths ← 0;
    appInPaths ← 0;
    if inActions performs no packet transformations and match correlates with
     inMatch then
        foreach path in paths do
            if inSwitch in path then
                inSwitchInPaths++;
            end
            if app has target switch in path then
                appInPaths++;
            end
        end
    end
    numberOfPaths ← length of paths;
    if numberOfPaths > 1 and numberOfPaths == inSwitchInPaths and
     numberOfPaths > appInPaths then
        raise Injection conflict with vertex and inVertex data;
    end
    recurse with vertex, inVertex, app;
end
```

Algorithm 3: Detection algorithm for Injection conflicts.

6.6 Bypass

Bypass conflicts can only occur if an application tries to enforce policies in a topology with multiple paths from source to destination and fails to insert rules on switches on all possible paths. Detection is limited to flow entries with actions other than direct output to a next node. This allows for multiple routing options without raising a conflict, while considering policies that aim at filtering, packet transformation or multi-table processing within a node. Furthermore, actions of flow entries in switches other than the current target switch of a packet-in message might look different while enforcing the same policy, limiting us to a check if an application inserting respective flow entries covers all possible paths with its target switches. For any flow entry containing transformation actions, we determine all target switches of the producing application in Algorithm 4 and all routes to the destination of affected packets. If all paths contain a target switch of the application, we assume that it is able to enforce its interest. For any path without a target switch, a warning is generated. We have to note that this will produce a warning for potential conflicts and does not ensure that the actual conflict pattern, i.e. multiple paths in the rule graph for the same packet flow, is present. The reasoning for this is that uncovered paths likely represent a faulty configuration and pose a risk of bypassed packet transformations. Furthermore, checking alternative paths and each individual set of data plane policies in all switches on the path for a correlation to the packets match fields is computationally expensive.

```
input: flow entry inserted by application containing transformation actions
match, actions, switch, app ← flow entry;
paths ← getAllSimplePathsFromMatch(match);
targetSwitches ← getTargetSwitches(app);
coveredPaths ← 0;
numPaths ← 0;
foreach path in paths do
    numPaths++;
    foreach switch in targetSwitches do
        if switch in path then
            coveredPaths++;
            break;
        end
    end
end
if numPaths - coveredPaths > 0 then
    raise Bypass conflict with rule and paths data;
end
```

Algorithm 4: Detection algorithm for Bypass conflicts.

6.7 Occlusion

The transformation of a packet flow by an application can lead to ineffectiveness of a subsequent application on the route to a destination. The experiment model is designed to allow for determination of application interests in endpoints and permits detection of Occlusion

conflicts. The detection algorithm 5 is applied to any rule insertion that involves transformation actions of destination addresses. Once such a rule is inserted in the rule graph, we follow all possible paths to a destination and determine apps that specify a target switch on such a route. Apps that adhere to this condition and claim interest on an endpoint address that is affected by the packet transformation are possible candidates for Occlusion conflicts. For address based interests, we need to compare not only for equality, but also if addresses are within overlapping address spaces. Since this algorithm is forward oriented on a path in a topology and any previous Occlusion conflicts on a switch prior to the current switch have already been checked, we consider paths from the current switch to a packet destination only.

```
input: app of current rule, vertex in rule graph, actions of rule with transformation
       actions, match fields of rule
currentSwitch ← getSwitchFromVertex(vertex);
interests ← empty list;
candidateApps ← empty list;
allInterests ← getAppInterests();
foreach action in actions do
    if action transforms destination endpoint address then
        address ← getPropertyFromMatch(matchfields);
        append address to interests;
    end
end
if interests is empty then
    return;
end
foreach appId, appInterests in allInterests do
    foreach i in interests do
        if i in appInterests and appId ≠ app then
            append appId to candidateApps;
        end
    end
end
if candidateApps is empty then
    return;
end
paths ← getAllSimplePathsFromMatch(matchfields, currentSwitch);
foreach appId in checkApps do
    targetSwitches ← getTargetSwitches();
    foreach switch in targetSwitches do
        foreach path in paths do
            if switch in path then
                raise Occlusion conflict with vertex and app data;
                break;
            end
        end
    end
end
```

Algorithm 5: Detection algorithm for Occlusion conflicts.

7 Evaluation

A qualitative approach promotes understanding an interpretation of the observations in this work and serves as a first stepping stone for exploring distributed conflicts in SDN. The detection algorithms are tested in a controlled environment on designed and random topologies to examine their soundness and completeness. Since the rule graph represents the core component of the detection mechanism, its properties are reviewed. We compare the experimental infrastructure to similar projects in order to highlight the utility of this approach as opposed to other work. Conflict classes and their impacts are evaluated concerning security in SDN and put in context of known work. Lastly, the parameter space as defined by Tran and Danciu [TD20] is reassessed in light of distributed conflicts.

7.1 Conflict Detection

Given the definitions of distributed conflicts in Chapter 5, there are some assumptions that will impact the correctness of the detection results and methods. The detected classes can be valid conflicts but can also be intended configurations that are chosen for optimization purposes or any other intentional dependence between control applications. Consequently, intentional inter-dependence between application configurations that lead to conflicts cannot be discerned from conflicts arising from the erroneous or unintended. This circumstance can also be observed in the ordering of filtering rules in traditional firewall applications or even in the ordering of flow entries within a single target switch of an SDN application. An intentional optimization appears to be a conflict in objective, analytical terms. The in-explicit intentions of a network operator can therefore result in an undecidable, non-deterministic problem.
Another important aspect is how many applications make decisions that form a conflict class, concisely, if conflicts can occur between more than two applications. Let us assume that all applications are deployed on different switches. This will firstly always lead to a conflict between two applications on a route. The end product of that conflict will result either in a broken route and no other application receives a packet-in event to process the packets further, or drops which will have the same effect, or some form of structurally valid flow entry by the second application. That flow again gets processed by the next application in a third switch on the route, where any conflicts will arise between the policy of the second application and the third. Inductively the same reasoning will apply for any number of applications that make decisions on a path.n If there is no conflict between two directly successive applications, any of the two can conflict with the next decision making process on a path, but this again is the result of two competing applications respectively. So the resulting decision due to a conflict between two applications can again lead to a conflict with a following policy, but there is no reason to assume that there is any correlation between the first and the last policy in a conflict from more than two policies. The only exception to this reasoning is the loop conflict, which can be the result of an arbitrary number of applications. No applications appear to compete with each other in a distributed sense, but the overall

distributed decision is faulty.
Furthermore, we need to consider what happens if we have two applications deployed on the same target switch and a conflict pattern that arises from a policy in a previous or successive node. Due to the nature of the OpenFlow [ONFb] protocol, a packet is processed by exactly one flow entry in a switch. A switch can have multiple flow tables, but these can be seen as virtual switches when detecting distributed conflicts. This means that of the two applications that are deployed on one target switch, both might install a flow entry yet only one policy will process a matching packet. Therefore, any conflict will again be the product of two conflicting applications. Which of the two possibly locally conflicting applications wins in the decision making process, e.g. due to a higher priority, has no impact on the number of applications that are part of the distributed conflict even if the outcome of the local conflict can lead to a different result for the distributed conflict.

7.1.1 Evaluation Experiments

Conflict classes and conflict pattern in the detection algorithms have to be independent from specific points in the input parameter space 2.5. We will assess their relevance in a set of evaluation experiments. We will examine two designed and two random topologies. Each will be subject to five experiment rounds on every conflict class. Each experiment on a topology and for a given conflict class will be conducted with the same applications. The applications are configured to elicit the conflict class under question. The qualitative insights we want to gain from the evaluation runs comprise the following topics:

- revalidate the observed conflict classes on a set of untested topologies
- expose the conflict detector to the experimental procedure with unseen topology shapes to assess the quality of the prototype for detection
- asses the quality of the detection algorithms

The random topologies in the evaluation set are 3.12, 3.10, and 3.7 and 3.5 the designed. We define in total 120 experiment models and build the respective experiment with the procedure described in Section 4.5. The results are then analyzed to assess the number of occurred conflict patterns, the number and shapes of detected conflicts and the number of valid cases. Tables 7.2, 7.1, 7.4 and 7.3 show the results of the evaluation runs with occurrences, detections and valid detections from left to right in each cell. Rows 1 to 5 represent one experiment round for each individual conflict.
For loops and Spuriousness conflicts, the experiment results on data plane state failed to expose the actual number of occurrences in data plane state and we resort to assessing the number of valid cases based on the logged paths in the rule graph 6.1. Two experiment rounds on loops on the two designed topologies 3.7 and 3.5 where unsuccessful in both cases even after repeated attempts. Experiments on Spuriousness conflicts for the two designed topologies failed in every run and after repeated attempts. Invalid experiments are marked with an *X*.
The results show that detection based on a rule graph and the devised detection algorithms are subject to high latency. This leads to only partial detection of conflicts before an experiment run is stopped and experiment output is generated. Short flow entry timeouts entail that the output on data plane state is incomplete. Flow entries are deleted from the data plane before results can be generated. In order to improve experiment results, it is necessary

to store data plane state at intermediate steps and ensure that the detection prototype is not stopped before it can process all flow entry insertions to the data plane. While the automation process yields realistic circumstances in comparison to artificial or manual insertion of flow entries to the data plane, it is less controllable and not yet suited for quantitative analysis. Artificial insertion of complete paths from source to target endpoints is controllable, but it is unclear how that affects detection latency. Realistic data plane states for a given topology and timings for flow entry insertions will improve the quality of the evaluation process, but are not available.

Run	Occlusion	Bypass	Loop	Spuriousness	Injection	Multi-Transform
1	4/4/4	6/4/4	12/6	2/2	2/2/2	2/2/2
2	8/4/4	12/8/8	15/5	4/4	4/2/2	4/4/4
3	12/8/8	18/12/12	10/4	6/6	6/2/2	6/6/6
4	16/8/8	18/12/12	15/10	8/8	8/2/2	8/8/8
5	20/8/8	18/12/12	16/10	10/10	10/6/2	10/8/8

Table 7.1: Evaluation experiments on random topology 3.12 with 10 switches and 8 endpoints. Rows 1 to 5 represent one experiment run for each individual conflict. Each cell shows the number of occurred conflict patterns, the number of detected conflicts and the number of valid cases from left to right. For loops and Spuriousness, the true number of occurrences could not be assessed and are omitted.

Run	Occlusion	Bypass	Loop	Spuriousness	Injection	Multi-Transform
1	4/4/4	6/4/4	6/3	2/2	2/2/2	3/3/3
2	8/4/4	12/10/10	13/11	4/4	4/2/2	4/4/4
3	12/8/8	18/16/16	4/2	8/8	6/2/2	8/8/8
4	16/8/8	24/21/21	4/2	10/10	8/4/2	12/12/12
5	20/8/8	24/19/19	11/9	12/12	10/4/3	12/12/12

Table 7.2: Evaluation experiments on random topology 3.10 with 20 switches and 12 endpoints. Rows 1 to 5 represent one experiment run for each individual conflict. Each cell shows the number of occurred conflict patterns, the number of detected conflicts and the number of valid cases from left to right. For loops and Spuriousness, the true number of occurrences could not be assessed and are omitted.

Occlusion

Occlusion conflicts occur in all evaluated topologies 3.12, 3.10, 3.7 and 3.5. Tables 7.2, 7.1, 7.4 and 7.3 all show that the conflict pattern can be detected but that latency is an issue. Not the size of a topology is the deciding factor for detection success, but the number of occurrences. One consequence of the detection mechanism presented in Chapter 6 is that we may detect multiple conflicts for one flow entry because we check the number of target switches that are occluded and not the application itself. If an occluded application is active on two switches on a path from source to destination, we cannot decide which of the two will be occluded. Occlusion conflicts are interpretative, as defined by Pisharody et al. [PNC+19]. While we can determine competing application intents with the approach in

Run	Occlusion	Bypass	Loop	Spuriousness	Injection	Multi-Transform
1	2/2/2	4/2/2	24/12	X	2/4/1	2/1/1
2	2/2/2	4/2/2	94/47	X	4/4/1	2/1/1
3	4/4/4	4/3/3	X	X	6/8/2	4/2/2
4	4/4/4	4/4/4	152/76	X	8/6/2	4/2/2
5	4/4/4	4/4/4	X	X	10/8/2	2/2/2

Table 7.3: Evaluation experiments on random topology 3.7 with 26 switches and 14 endpoints. Rows 1 to 5 represent one experiment run for each individual conflict. Each cell shows the number of occurred conflict patterns, the number of detected conflicts and the number of valid cases from left to right. For loops, the true number of occurrences could not be assessed and are omitted. Cells that contain an *X* mark unsuccessful experiments.

Run	Occlusion	Bypass	Loop	Spuriousness	Injection	Multi-Transform
1	8/8/8	4/4/4	X	X	2/3/2	2/2/2
2	8/8/8	8/2/2	38/19	X	4/4/1	5/5/5
3	12/8/8	12/2/2	46/22	X	6/10/2	6/6/6
4	16/8/8	16/4/4	X	X	8/12/3	8/8/8
5	20/8/8	20/6/6	6/2	X	10/12/1	10/7/7

Table 7.4: Evaluation experiments on random topology 3.5 with 21 switches and 21 endpoints. Rows 1 to 5 represent one experiment run for each individual conflict. Each cell shows the number of occurred conflict patterns, the number of detected conflicts and the number of valid cases from left to right. For loops, the true number of occurrences could not be assessed and are omitted. Cells that contain an *X* mark unsuccessful experiments.

this work, automated conflict resolution is not possible without information loss and entails manual effort.

General Multi-Transform

The evaluation runs show that the conflict class is independent of the deployed topology. Tables 7.2, 7.1, 7.4 and 7.3 all show that the conflict pattern can be detected, but the size of a topology and latency can affect the success. This conflict is interpretative, as defined by Pisharody et al. [PNC+19], and cannot be resolved automatically without information loss and possible violation of application intent. It needs to be resolved or marked as valid by an administrator.

Bypass

Bypass conflicts are detected on all evaluation topologies. While Tables 7.2, 7.1, 7.4 and 7.3 show that all detected cases are valid, they also highlight that not all cases are detected. This is due to detection latency and the experimental process. It is hard to reason if this is an intelligible or interpretative conflict [PNC+19] that can be resolved without information loss. If we assume a faulty or incomplete configuration by an administrator, a simple resolution method is the insertion of rules on uncovered paths. However, by making that assumption we already try to infer the intent of the configuration.

Loops

Even with repeated tries, the experimental process did not produce any results on runs 3 and 5 shown in Table 7.3 as well as runs 1 and 4 in Table 7.4. This can either mean that detection latency is too high or that the experiment is based on a configuration that leads to its failure. Successful experiments showed that the prototype can detect loops, but valid cases are substantially lower than the detected. An analysis of the detection output revealed that the prototype produces duplicate detections for one loop. The duplicates differ in the ordering of the rules in a loop and might contain vertices in the rule graph that are connected to but not part of a loop. The prototype stores patterns for detected loops in memory, but fails to address the fact that the start and end vertex in a loop is not significant since it is the same closed structure. Also the rule graph can contain loops that are never traversed by traffic. Policies can be inserted to the data plane that form loops, but they are never activated and receive no bandwidth. Furthermore, loop detection with a rule graph has the caveat of uncontrollable complexity with increasing size of a topology and increasing complexity of the loop itself. The experiments on loop conflicts exhibited long runtimes, although the input parameter space 2.5 contained only one active application and limited complexity. As stated in formula 3.3, the time for an experimental run increases with the complexity of the input parameter space. This may indicate that the increased bandwidth on links and the computational complexity of loop detection impair network stability and detection. This conflict is intelligible, as defined by Pisharody et al. [PNC+19], and can be resolved automatically. We can break loops by inserting rules to destination endpoints with a higher priority than the affected policies, but that would entail that the prototype has to implement an efficient routing algorithm that does not produce loops itself.

Injection

Injection conflicts are detected on all evaluated topologies. The results in Tables 7.2, 7.1, 7.4 show a number of invalid cases for detection of this conflict pattern. As with loops, the detection method is prone to producing duplicates. The algorithm described in Chapter 6 fails to address the issue that an application may insert subsequent rules to the rule graph and can cover all paths from a source to a destination endpoint on subsequent switches. This is an interpretative conflict [PNC+19] and cannot be resolved without information loss.

Spuriousness

Since we deploy a filtering implementation based on bandwidth, TCP traffic generated by the automation process fails to reach the defined thresholds. Evaluation runs on random topologies show that all detected cases are valid but that detection suffers from latency. We could resolve Spuriousness conflicts by migrating filtering policies to an appropriate edge node in a network. However, security applications in SDN are subject to a number of requirements, as discussed in Section 7.5, which suggests that a traditional firewall deployment in SDN is insufficient.

7.2 Prototype

A rule graph as the key concept of the detection prototype has advantageous properties in light of detectable conflict patterns but negative implications on runtime and memory complexity. While other work [PCH16] [PNC+19] [LCL+18] [KZCG12] states the necessity for an optimized representation of data plane state and the need for scalability to produce real-time responses, they are limited in the domains and patterns for conflict detection. A common approach is a trie data-structure [PCH16] [PNC+19] [LCL+18] [KZCG12], a tree-like implementation where nodes from root to leaf store individual bits of IP addresses. This allows for conflict detection and data plane maintenance in one pass of the trie. A trie is based on destination addresses only and fails to comprise the scope of conflicts on multiple address fields. Adaptions of the trie include additional match fields of flow entries, but are still limited to destination addresses. The bit-wise comparison for rule insertion and detection of address overlaps constitutes the efficiency of the algorithms and also its limitations. The maintenance of links between rules and comparison of multiple address fields entails exponential complexity. Rule graphs are the only thorough approach that includes all match fields, the entire address space and links between rules. This defeats the need to request any flow statistics from switches on demand and reduces control channel bandwidth, but as said, adds memory and time complexity for the insertion and removal of rules. Scalability becomes an issue in large-scale networks with a high number of unique flows and control applications that enforce individual policies. While policies based on solely destination addresses seem feasible, obliviousness of more general conflict patterns is the result. The poor runtime performance of rule insertion and deletion with a rule graph is a drawback concerning local conflicts. However, for detection of distributed conflicts, two conditions render the entailed latency irrelevant. A distributed conflict can only be detected once a packet traverses at least two nodes in a topology. This means that a policy to forward a packet needs to be inserted to a switches data plane before the next decision on a subsequent switch will be triggered. Flow entries become active before they can be

processed for distributed conflicts. Furthermore, without a means to interpret application intent, resolution of distributed conflicts as of yet is not feasible. Without a strategy for automated resolution of distributed conflicts, it is not relevant if the detection is performed in real-time since manual resolution cannot be performed in real-time either. Therefore, time complexity is not a critical issue for detection of distributed conflicts since we potentially need to wait for insertion of all rules of a flow from source to destination. Loop conflicts support this argument most intuitively. Depending on the shape of a topology, loops can be of arbitrary size. The loop is only verified once a packet traverses the next to last node in the loop and an automated approach to break the loop can violate any application intent involved in the loops emergence. Also, the control channel is not notified of a second pass of packets through a node with a matching rule. An alternative solution is the use of historical data on bandwidth to check if there is an extraordinary increase in a certain region of the network.
Memory complexity in conflict detection based on a rule graph is an unresolved issue. Any implementation of conflict detection suffers from maintaining any form of copied data plane state, and controller nodes for large-scale SDN. This requires significant amounts of resources in terms of processing power and memory even for smaller networks and a high volume of flows.
The detection methods described in Chapter 6 respect the existence of ambiguous application intents and the need to apply heuristics to avoid parsing the entire rule graph where possible.

7.3 Experiment Automation

The Mininet project [MPC] states several features that constitute the utility of network testbeds that serve as a guideline for evaluating experiment automation in this work.

- Functional realism concerning fidelity to deployed hard- and software
- Realism and reproducability of any generated network traffic between experiments and researchers based on a common specification of experiment input and results
- Topology customization with any topology shape and size
- Easy duplication of experiment setup and deployment

Mininet [MPC] is a network emulator based on Linux namespaces and follows two additional characteristics, namely the criteria of low cost and timing realism. Emulab [Uta] implements network testbeds based on hardware virtualization with Xen [TLF] domains, both with dedicated physical resources for individual nodes and more scaleable topologies on a single machine. A combination of desirable features of both projects is given by the presented approach in this work, in addition to a configurable approach for conflict detection and monitoring as well as a framework for pluggable applications.
Mininet [MPC] is based on Linux namespaces within a wrapper VM, while controllers can be situated outside of the VM. The concept of using a wrapper VM to ease deployment on any operating system compares to this work yet virtualization techniques are limited to network emulation and resource allocation. The use of Xen [TLF] domains allows for increased realism with separation of memory space and process execution as well as dedicated shadow structures for kernel execution of guest domains. Emulab [Uta] also makes

use of Xen [TLF] domains and external controller execution, while offering more flexibility in terms of Linux kernel versions. However, Emulab [Uta] lacks specification and automation for traffic generation and monitoring, a key characteristic for repeatable experiments. Emulab [Uta] requires the deployment of custom monitoring controllers to generate and monitor traffic or manual command line interaction. Mininet [MPC] provides configurable webservices such as HTTP servers and connectivity via iperf [Sofb]. It features topology wide reachability tests and custom control over MAC and IP addresses. The approach in this work completely automizes reachability tests and traffic generation of specified transport types between endpoints, further improving reproducibility of results. Access to individual endpoints and switches is given to add manual testability. Mininet [MPC] does not automate network tests, but rather provides a set of commands to generate traffic or implement the automation via custom scripts. Network bandwidth is advertised as large, but that is due to direct use of one VM kernel, the absence of shadow structures for each individual host OS and a lack in separation of processes execution. Para-virtualization as used in Emulab [Uta] and this work creates more realistic testbeds in separation and virtualization of machine and network state. Neither of the two implementations provide automated conflict detection and analyzing of monitored state, namely comparison of data plane and deviations of generated bandwidth. Emulab [Uta] and this work permit for individual control over host domains, with the option of custom setup, where new features in Mininet [MPC] are introduced to all nodes in the topology due to the shared application stack.
Both alternative implementations allow for deployment of custom controllers and more switch types besides Openflow switches, with a strong emphasis on reproducible network infrastructure experiments. This work provides a conflict detection mechanism and aims at reproducible experiments with artifacts for determining new conflicts. With this approach, performance isolation is a given. Mininet [MPC] does this artificially via control groups and processor bandwidth limits. Mininet [MPC] is the best fit for experiments that are limited by network properties such as bandwidth, latency, and queuing, as opposed to this work which focuses on debugging of applications and a controllable environment for conflict exposition. Scalability concerning topology size and setup time is optimized with Mininet [MPC]. Functional realism and control is improved by para-virtualization, while the resource constraints are given by disk space and memory. Disk space and memory are determined by the number of nodes in the topology, while memory is adjustable.
Remaining issues in this work are the dependency of conflict classes on application intent and instability on larger topologies. More application types are needed to really verify the comprehensiveness of conflict patterns and detection. Another unsolved issue is the yet unsolved problem of local conflicts. To enable easy integration of applications without the need of adaptions to fit the experiment infrastructure, automated local conflict resolution is required. Another topic is the choice of application configurations and positioning of endpoints in a topology. For objective exploration of the parameter space, settings need to be chosen randomly without intentional preconditions. A balance between exploitation of conflict potential anticipated by researchers and exploration through randomness promotes an aggregation of reproducible data for further analysis. Nevertheless, the experiment model and technical implementation eases deployment and testing of infrastructure under realistic conditions and with affordable resources. The utility of the experiment model can be further improved towards an Infrastructure as Code service that facilitates shared, generic experiments. As a result, configurations and applications can deployed in a cloud environment. The adjustable threshold for monitoring and reporting bandwidth limits addresses the fact that

individual topologies, configurations and the underlying hardware produce different results. This affects any Invariant Contention conflicts, which are only exposed through repeated experiments in the same environment and gradual adjustment by a researcher. Bandwidths and timings for conflict detection are therefore not reproducible across machines.
Application intent, other than endpoints and switches, is not directly incorporated in the experiment model. Current work incorporates application intent in mediation and resolving mechanisms [SMR+14]. The proposition of viable network states by applications [AMB+14] and distinction between resource control interests and general invariants [PNC+19] [AMB+14] [SMR+14] is another form of application intent. The common denominator in the approaches is that application intent is interpreted based on these inputs, yet we need a means to capture application intent without interpretation. There is a key difference in utilizing application intent for resolution tactics and for discovering conflict classes. Bandwidth is a first and obvious interest that could be declared by applications. A sensible set of invariant flags can be incorporated in the experiment model, and any applications that interact in a rule path can be checked for intersecting interests, although that again only exposes a potential for conflict, not the conflict class itself.
Comparison of expected and observed state is an objective method that can expose some but not all conflict classes. Flow entry timeouts represent one caveat in this method. If flows timeout too fast the implementation fails to respect them in comparison of data plane state. Short timeouts and high algorithmic complexity leads to incomplete comparisons that can distort the results.

7.4 Parameter Space

A choice of dimensions in the parameter space presented by Tran and Danciu [TD20] merit reassessment in light of results of this work. These dimensions include app start order, target switches and app configuration, transport type, topology design and app priority.
One insight is that priority is not a deciding factor for distributed conflicts. App priority within flow tables decide which policy in a pool of matching flow entries is applied to packets, which makes them inherent to local conflicts. The winning policy of a local conflict is the prospective candidate for a distributed conflict, yet the outcome might as well inhibit potential inter-policy conflicts. In absence of local conflicts, priority is a negligible experiment dimension. Furthermore, there is no mechanism to restrain implementations from altering their priority, resulting in increased complexity of the parameter space and convolution in conflict resolution. The only feasible approach is role-based source authentication as presented by Porras et al. [PSY+12], effectively voiding any input priorities by applications and basing precedence on application intent.
Network topologies showed to be relevant concerning characteristic shapes, such as bottleneck nodes, alternative routes provided by rings and meshes, as well as path lengths and respective routing decisions. While we can argue that there is an infinite number of topologies, a simple bus or ring topology at some point will no exert more insights no matter how many nodes are added. The deciding factor for emergence of conflict patterns is the utility an application exerts for a given topology. Endpoint load balancing requires multiple endpoints to work as expected and path load balancing multiple paths. Any deployment in absence of such properties is defective. While randomization can and will produce relevant shapes, an increasing number of applications and diversity between application implementations is

correlated to a higher number of defective shapes. Input by application developers leads to smaller entropy in the randomization process. Consequently, application intent appears to be a necessary input for topology design, representing a bounding factor for improved randomization and a limit to the extent of the topology dimension. Several experiments for random topologies 3.8 and 3.10 that feature the same number of switches and endpoints indicates this issue. Application configurations are devised for one topology and then used for the second in an simple attempt to randomize the parameter space. The result is usually a failed experiment run due to dysfunctional application configurations.
Transport types are another subject for evaluation, i.e. in this case IP protocols. For local and distributed conflicts both, the deciding property for conflict emergence is a part-of relationship, whereas disjunct types are irrelevant for any conflict patterns. Policies enforced on TCP or UDP traffic specifically will never compete with each other. Application intents aimed at disjunct protocols will also never be interdependent. Therefore, conflicts on transport types emerge when a policy that is oblivious of transport type competes with a policy that matches a specific type. A suggestion for a more comprehensive input parameter for conflicts, is a distinction of within- and cross-layer conflicts on OSI layer protocols. Within-layer conflicts are limited to the mentioned part-of relationship and cross-layer conflicts can occur between any protocol types of two distinct OSI layers that are used in sensible combinations.
Start order of applications and its relevance for experiments is subject to two assumptions. When implementations are added to the control layer, which is closely coupled to the controller implementation, a restart of the controller is required. Controller restarts strongly suggest a complete flushing of data plane state to avoid hidden conflicts [TD20] and uphold correctness of data plane state. After a restart, all applications of the control layer start at same time, receive the same set of subscribed messages and operate on the same, clean view of the data plane. The only applications that can be added or removed safely to or from a running system are loosely coupled implementations, e.g. integrated through REST interfaces or another north-bound API to the application layer. As elaborated in Section 3.2, the increased latency of loosely coupled technical interfaces usually exempts packet-in events from inclusion in such interfaces. Therefore, the only candidate applications for the start order dimension are exempt from the messaging system and are advised to rely on querying and maintaining data plane state.
Dependency between target switches of an application and application configuration is the final point of discussion. A trivial example of such a dependency is an application deployed on target switches and operating on switches as part of its application. Another example are applications that receive configuration for individual target switches.

7.5 Security

A rule graph can serve as the core implementation of a security enforcement layer as defined by Dixit et al. [DKZ+18]. The implemented conflict detector meets the requirement of centralized policy enforcement and requirement of storing all forwarding graphs based on data plane state, yet conflict detection between policies is performed offline. Concurrency is handled by the inevitable delay for detecting distributed conflicts, but this leads to possibly invalid traffic already penetrating the network. Unmet requirements include tracking of connection state and vulnerabilites concerning scalability, which is a general deficiency of

the OpenFlow [ONFb] protocol. The concept of analyzing only the first packet of a flow and installing flow entries for handling any subsequent packets deprives the control layer of any further packet inspection. If rules are too general, this can lead to vulnerabilities to attacks where benign packets are sent first. Once a too coarse policy is inserted, e.g. by failing to limit valid IP protocols, malign packets can follow. To keep track of connection state, controllers need to enforce the use of timeouts for rules or monitor bandwidths on ports and flows as exemplified in the Firewall application. In addition, automated conflict resolution is missing due to the interpretative nature of distributed conflicts. Several of the applications involved in this work maintain a view on data plane state within the control layer. This includes the Host Shadower, Endpoint Load Balancer and Firewall and, at a global level, the detector. In light of a productive environment in a data centre, with very large numbers of concurrent flows and high fluctuations, this can affect scalability.
Not all of the conflict classes described in Section 5 implicate security issues. Injection relates most strongly to faulty configuration and conflicting interest to applications that aim at distributing bandwidth between network links or hosts. Loops also affect efficiency rather than security and are an exception. Other Transformation Contention conflicts, Invariant Contention as well as Spuriousness and Path Ambiguity, can entail security issues next to network deficiencies. Transformation implementations can be exploited in attacks by adapting flows circumvent existing firewall rules, as mentioned by several works [HHAZ14] [PNC+19] [LCL+18]. This concerns General Multi-Transform conflicts. However a reverse approach can also lead to information leaks. Considering the security scheme of moving target defense as mentioned by Hu et al. [HHAZ14], incomplete transformation and Bypass conflicts can be exploited by attackers to explore and disable these efforts. The concept of moving target defense aims at disguising resources for services by altering packet destinations or reassigning addresses and relates to the implementation of the Endpoint Load Balancer and Host Shadower applications. Attackers exploiting said conflict classes are able to discover the resources hidden by the implementations and can employ reconnaissance and penetration mechanics. Furthermore, penetration testing could reveal defects in the configuration by eliciting invariant contention conflicts. The issues concern both the location of conflicts as well as missing transformations. Resolving placement issues can be performed by automatically inserting any missing policies to cover all paths between traffic source and destinations. Spuriousness and Occlusion arise due to improper deployment of applications within a network. Undesired loads on endpoints and vulnerable network regions to port scanning and other network reconnaissance techniques are the consequence. The location of enforced policies or interests dictates the emergence of these conflict classes, which indicates a resolving strategy. Spuriousness could be resolved by migrating a rule to an edge switch, but global application intent can lead to new conflicts. An intrusion detection mechanism could be occluded by the migration. Occlusion conflicts need to be analyzed and the rational placement of involved applications reassessed. The implemented firewall shows how a control application can request data plane state to check current rules and deploy it's own policies to override other policies with a higher priority. Therefore, role-based authentication is suggested by Porras et al. [PSY+12]. The concept aims at preventing operations of unknown actors and can also be a tools for resolving conflicts by adding a layer of authorization priority. However, distributed conflicts are unaffected by rule priorities.
The idea of an extra layer of dependencies by an administrator can help avoid false positives in conflict detection and put the focus on true positives and possible security issues. We propose the introduction of security alignments to declare any valid interactions between

applications in order to avoid detection of intentional configuration as conflicts. Every application is part of a set of security alignments and does not conflict with rules deployed by the same alignment. Alignments are enforced in the controller implementation. Data plane state is not allowed to store alignment information in order to avoid malign applications from learning alignments by accessing data plane state through REST interfaces. Let us consider an alignment set of [1,2] for the Endpoint Load Balancer, [2] for the Host Shadower and [1] for the Firewall. Any conflicts arising from a rule path generated by the first two applications or the last two would therefore be ignored in conflict detection as they intersect in their alignment. Rules paths that involve conflicts between the Host Shadower and Firewall in this scenario represent valid conflicts and possible security issues. Administrators can even start from blank alignments in a specific configuration and gradually add to the alignments to invalidate conflicts and discern problems that need to be addressed and interpreted manually. This implementation can eliminate the need for knowing application intent by avoiding the necessity for resolving strategy for the validated interactions. As mentioned, the discovered distributed conflicts are interpretative, and resolution requires application input or heuristics. Nevertheless, any resolution strategy can entail indefinite evocation of new conflicts and the impairment of valid, authorized traffic. Security alignments can reduce the set of candidates and scale the need for automated resolution.

8 Conclusion and Future Work

This work shows that new distributed conflict classes are exposed and revalidated by an experimental approach. Policy conflicts are examined on a diverse set of network topologies and control applications. The experimental process still exerts technical and conceptual weaknesses that need to be addressed by future work. We present several key points for future improvements and provide a summary of this work.

8.1 Future Work

Several issues remain with the current detection prototype. Rule graphs as depicted in Figure 6.1 entail a high runtime complexity and scalability issues. A naive approach to improvement could be the discarding of unique flows once the complete forwarding path has been checked since they are not relevant for other traffic. At the moment, the detector needs to ignore errors for connectionless traffic because the added latency of the detection layer is too high. The evaluation of the experimental process has show that the deployment of control applications is not stable in large networks.
Exploration of new conflict classes and the objectivity of current classes could be further accelerated by the randomization of endpoint placement and randomization of application configuration. The presented approach includes random topologies concerning switch placement but research on the dependencies between application configurations and their requirements on topologies are necessary.
The evaluation of the conflict detection prototype with automated experiments demonstrates that a controllable and quantifiable approach is necessary to further assess the utility of detection algorithms and a rule graph.
Introduction of resolving strategies can be based on security alignments 7.5 that mark acceptable dependencies between apps. Further methods include replacement strategies to resolve Spuriousness and Occlusion conflicts and the automated insertion of missing transformation policies.
The most ambiguous topic for future work concerns application intent. Is it sensible to define and continually grow a set of flags, keyword or properties that can be communicated by apps? This will limit an applications independence in SDN. The possibilities and benefits for conflict resolution or avoidance need to be researched and the implemented.

8.2 Summary

Software Defined Networks are a promising paradigm in computer networking and facilitate network policy enforcement by multiple entities through a centralized interface. The concept entails new possibilities for policy enforcement but also new problems. This work introduces the concept of distributed conflicts based on the definitions by Hamed and Al-Shaer [HA06] in the SDN context. Policies by two applications on multiple nodes in a network can con-

tradict and can result in connection failure or compromised security. The focus on local conflicts by related work and missing insights on distributed conflicts are the motivation for this thesis. The differences of SDN compared to traditional networks are a separation of management plane where administration of network devices takes place, control plane where network policies are generated and the data plane where policies reside. This separation allows for centralized decisions through a controller as opposed to local decisions that need to be propagated between nodes in a network.
Research questions include the exploration of distributed conflicts and their detection. Furthermore, a technical infrastructure to enable and promote research on conflicts is desirable. Applications with competing interests produce conflicts and are a necessary input for exploration of new conflict classes. We examine existing work and tools in SDN and what the existing definitions of policy conflicts are. Security in SDN depends on a global view of data plane state on all network devices, real-time processing of policy insertions as well as conflict detection and automated resolution [DKZ+18]. We evaluate the presented approach with regard to these requirements.
Tran and Danciu [TD19][TD20] provide the basis for this work and define an input parameter space for experiments on policy conflicts as well as a methodology for conflict exploration. The parameter space comprises topology, application and network traffic characteristics. Tran [Tra22] implements a policy rule graph that can be utilized for conflict detection.
The contributions of this work include new distributed conflict classes illustrated in Chapter 5 and the validation of known conflict patterns in SDN. They are the product of 403 experiment rounds. Each experiment round comprises at least three experiments for two applications and at most 541 for five applications with one configuration. The classes are formalized and examples are given for each. Detection algorithms for the discovered conflict classes are implemented and evaluated on a fresh set of topologies. Detection algorithms are demonstrated in Chapter 6 and insights of the evaluation in Chapter 7. A software architecture for reproducible experiments on SDN conflicts is devised and presented in Chapters 3.3 and 4. The devised software architecture is an automated and technical implementation for the experimental methodology described by Tran and Danciu [TD20]. Topologies that are used for conflict exploration and evaluation are presented. This includes designed topologies as well as random networks based on three algorithms from research on network generation. Three new control applications from this work and four existing ones by Tran [Tra22] facilitate the experiments. They include implementations for load balancing, traffic engineering and filtering. The experiment automation process is compared to similar approaches in Chapter 7 and evaluated in light of mentioned security requirements in SDN. The insights show that firewall applications are subject to new challenges due to the novel properties of SDN. The presented implementations are evaluated on their validity by a second set of experiments and qualitative insights on the soundness of detection algorithms are explained. The automated experimental infrastructure as well as a set of seven existing control applications support future work on network policy conflicts, lowering the threshold for performing experiments and gaining further insights. While the implementations are based on Openflow as a technical framework, the formal definitions for conflict classes and presented detection algorithms apply to SDN in general. We show that experimental exploration of conflicts yields results that increase understanding of policy conflicts and enables interpretation and resolution.

List of Figures

Acronyms

API Application Programming Interface(s)

EPLB Endpoint Load Balancer

FIB Forwarding Information Base(s)

FW Firewall

HS Host Shadower

HTTP Hypertext Transfer Protocol

ICMP Internet Control Message Protocol

IP Internet Protocol

JSON JavaScript Object Notation

LRZ Leibniz-Rechenzentrum

MAC Media Access Control

MB Megabyte

MWN Münchner Wissenschaftsnetz

NFV Network Function Virtualization

OS Operating system

OSI Open Systems Interconnection

PE Path Enforcer

PLB Path Load Balancer

PPLB4d Destination based Passive Path Load Balancer

REST Representational State Transfer

RFC Request for Comments

RIB Routing Information Base(s)

SDN Software Defined Network(s)

SSH Secure Shell

TCP Transmission Control Protocol

UDP User Datagram Protocol

VM Virtual machine(s)

Bibliography

[AB01] Albert, R. ; Barabasi, A.-L.: Statistical Mechanics Of Complex Networks. In: *Reviews of Modern Physics* 74 (2001), 06. http://dx.doi.org/10.1103/RevModPhys.74.47. – DOI 10.1103/RevModPhys.74.47

[AMB+14] AuYoung, A. ; Ma, Y. ; Banerjee, S. ; Lee, J. ; Sharma, P. ; Turner, Y. ; Liang, C. ; Mogul, J.: Democratic Resolution of Resource Conflicts Between SDN Control Programs. In: *CoNEXT 2014 - Proceedings of the 2014 Conference on Emerging Networking Experiments and Technologies* (2014), 12, S. 391–402. http://dx.doi.org/10.1145/2674005.2674992. – DOI 10.1145/2674005.2674992

[ARAM18] Asadujjaman, A. S. M. ; Rojas, E. ; Alam, M. S. ; Majumdar, S.: Fast Control Channel Recovery for Resilient In-band OpenFlow Networks. In: *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018, S. 19–27

[ASBS00] Amaral, L. ; Scala, A. ; Barthelemy, M. ; Stanley, H.: Classes of Small-World Networks. In: *Proceedings of the National Academy of Sciences of the United States of America* 97 (2000), 11, S. 11149–52. http://dx.doi.org/10.1073/pnas.200327197. – DOI 10.1073/pnas.200327197

[BA99] Barabasi, A.-L. ; Albert, R.: Albert, R.: Emergence of Scaling in Random Networks. Science 286, 509-512. In: *Science (New York, N.Y.)* 286 (1999), 11, S. 509–12. http://dx.doi.org/10.1126/science.286.5439.509. – DOI 10.1126/science.286.5439.509

[CSa] Cisco Systems, Incorporation: *2019 Global Networking Trends Summary of results for external use.* https://www.cisco.com/c/dam/en/us/solutions/collateral/enterprise-networks/digital-network-architecture/global-nw-trends-survey.pdf. – Accessed on 2020-08-13

[CSb] Cisco Systems, Incorporation: *2020 Global Networking Trends Report.* https://www.cisco.com/c/dam/m/en_us/solutions/enterprise-networks/networking-report/files/GLBL-ENG_NB-06_0_NA_RPT_PDF_MOFU-no-NetworkingTrendsReport-NB_rpten018612_5.pdf. – Accessed on 2020-08-13

[DKZ+18] Dixit, V. H. ; Kyung, S. ; Zhao, Z. ; Doupe, A. ; Shoshitaishvili, Y. ; Ahn, G.-J.: Challenges and preparedness of SDN-based firewalls. In: *SDN-NFVSec 2018 - Proceedings of the 2018 ACM International Workshop on Security in Software Defined Networks and Network Function Virtualization, Co-located with CODASPY 2018*, Association for Computing Machinery, Inc, März

2018 (SDN-NFVSec 2018 - Proceedings of the 2018 ACM International Workshop on Security in Software Defined Networks and Network Function Virtualization, Co-located with CODASPY 2018), S. 33–38

[DSNW15] Dwaraki, A. ; Seetharaman, S. ; Natarajan, S. ; Wolf, T.: State abstraction and management in software-defined networks. In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2015, S. 189–190

[ER60] Erdős, P. ; Rényi, A.: On the Evolution of Random Graphs. In: *PUBLICATION OF THE MATHEMATICAL INSTITUTE OF THE HUNGARIAN ACADEMY OF SCIENCES*, 1960, S. 17–61

[Fie00] Fielding, R. T.: *Architectural Styles and the Design of Network-Based Software Architectures*, University of California, Irvine, Diss., 2000. – AAI9980887

[Gil59] Gilbert, E. N.: Random Graphs. In: *The Annals of Mathematical Statistics* 30 (1959), Nr. 4, 1141 – 1144. http://dx.doi.org/10.1214/aoms/1177706098. – DOI 10.1214/aoms/1177706098

[HA06] Hamed, H. ; Al-Shaer, E.: Taxonomy of conflicts in network security policies. In: *IEEE Communications Magazine* 44 (2006), Nr. 3, S. 134–141

[HHAZ14] Hu, H. ; Han, W. ; Ahn, G.-J. ; Zhao, Z.: FLOWGUARD: Building Robust Firewalls for Software-Defined Networks. In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. New York, NY, USA : Association for Computing Machinery, 2014 (HotSDN '14). – ISBN 9781450329897, 97–102

[IETF] Internet Engineering Task Force, (IETF): *YANG A Data Modeling Language for the Network Configuration Protocol (NETCONF)*. https://datatracker.ietf.org/doc/html/rfc6020. – Accessed on 2021-09-24

[IRTF] Internet Research Task Force, (IRTF): *Software-Defined Networking (SDN): Layers and Architecture Terminology*. https://datatracker.ietf.org/doc/html/rfc7426. – Accessed on 2021-09-19

[KRV+15] Kreutz, D. ; Ramos, F. M. V. ; Veríssimo, P. E. ; Rothenberg, C. E. ; Azodolmolky, S. ; Uhlig, S.: Software-Defined Networking: A Comprehensive Survey. In: *Proceedings of the IEEE* 103 (2015), Nr. 1, S. 14–76. http://dx.doi.org/10.1109/JPROC.2014.2371999. – DOI 10.1109/JPROC.2014.2371999

[KZCG12] Khurshid, A. ; Zhou, W. ; Caesar, M. ; Godfrey, P.: VeriFlow: Verifying Network-Wide Invariants in Real Time. In: *ACM SIGCOMM Computer Communication Review* 42 (2012), 10. http://dx.doi.org/10.1145/2342441.2342452. – DOI 10.1145/2342441.2342452

[LCL+18] Li, Q. ; Chen, Y. ; Lee, P. P. C. ; Xu, M. ; Ren, K.: Security Policy Violations in SDN Data Plane. In: *IEEE/ACM Transactions on Networking* 26 (2018), Nr. 4, S. 1715–1727

[LFX+19] Lu, Y. ; Fu, Q. ; Xi, X. ; Chen, Z. ; Zou, E. ; Fu, B.: A policy conflict detection mechanism for multi-controller software-defined networks. In: *International Journal of Distributed Sensor Networks* 15 (2019), 05, S. 155014771984471. http://dx.doi.org/10.1177/1550147719844710. – DOI 10.1177/1550147719844710

[Mar] Marriott, N.: *TMUX - Terminal Multiplexer*. https://github.com/tmux/tmux. – Accessed on 2021-07-30

[MPC] Mininet Project Contributors, Project: *Mininet Overview*. http://mininet.org/overview/. – Accessed on 2021-08-09

[MT] MNM-Team, Research Group: *Munich Network Management Team*. https://www.nm.ifi.lmu.de/mnmteam/. – Accessed on 2021-09-10

[Nip] Nippon Telegraph and Telephone, Corporation: *What's Ryu*. https://ryu.readthedocs.io/en/latest/getting_started.html#what-s-ryu. – Accessed on 2021-04-27

[NSW01] Newman, M. ; Strogatz, S. ; Watts, D.: Random Graphs with Arbitrary Degree Distributions and their Applications. In: *Physical review. E, Statistical, nmiscar, and soft matter physics* 64 (2001), 09, S. 026118. http://dx.doi.org/10.1103/PhysRevE.64.026118. – DOI 10.1103/PhysRevE.64.026118

[OC] Oracle Corporation, Corporation: *Oracle VM VirtualBox*. https://www.virtualbox.org/. – Accessed on 2021-04-30

[ONFa] Open Networking Foundation, (ONF): *Global Transport SDN Prototype Demonstration*. https://opennetworking.org/wp-content/uploads/2013/02/oif-p0105_031_18.pdf. – Accessed on 2021-09-06

[ONFb] Open Networking Foundation, (ONF): *OpenFlow Specification*. https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.3.5.pdf. – Accessed on 2021-02-28

[PCH16] Pisharody, S. ; Chowdhary, A. ; Huang, D.: Security policy checking in distributed SDN based clouds. In: *2016 IEEE Conference on Communications and Network Security (CNS)*, 2016, S. 19–27

[PNC+19] Pisharody, S. ; Natarajan, J. ; Chowdhary, A. ; Alshalan, A. ; Huang, D.: Brew: A Security Policy Analysis Framework for Distributed SDN-Based Cloud Environments. In: *IEEE Transactions on Dependable and Secure Computing* 16 (2019), Nr. 6, S. 1011–1025

[PSF] Python Software Foundation, Organization: *Welcome to Python.org*. https://www.python.org/. – Accessed on 2021-04-27

[PSY+12] Porras, P. ; Shin, S. ; Yegneswaran, V. ; Fong, M. ; Tyson, M. ; Gu, G.: A Security Enforcement Kernel for OpenFlow Networks. In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. New York, NY, USA : Association for Computing Machinery, 2012 (HotSDN '12). – ISBN 9781450314770, 121–126

[Sim09] Simmel, G.: The problem of sociology. In: *American Journal of Sociology* 15 (1909), Nr. 3, S. 289–320

[SMR+14] Sun, P. ; Mahajan, R. ; Rexford, J. ; Yuan, L. ; Zhang, M. ; Arefin, A.: A Network-State Management Service. In: *ACM SIGCOMM Computer Communication Review* 44 (2014), 08. http://dx.doi.org/10.1145/2619239.2626298. – DOI 10.1145/2619239.2626298. ISBN 978–1–4503–2836–4

[Sofa] Software in the Public Interest, Incorporation: *Command-line network traffic analyzer*. https://packages.debian.org/stretch/tcpdump. – Accessed on 2021-04-30

[Sofb] Software in the Public Interest, Incorporation: *Internet Protocol bandwidth measuring tool*. https://packages.debian.org/stretch/iperf. – Accessed on 2021-04-30

[Sofc] Software in the Public Interest, Incorporation: *TCP/IP swiss army knife*. https://packages.debian.org/stretch/netcat-traditional. – Accessed on 2021-04-27

[SPY+13] Shin, S. W. ; Porras, P. ; Yegneswara, V. ; Fong, M. ; Gu, G. ; Tyson, M.: Fresco: Modular composable security services for software-defined networks. In: *20th Annual Network & Distributed System Security Symposium* Ndss, 2013

[TD19] Tran, C. N. ; Danciu, V.: Hidden Conflicts in Software-Defined Networks. In: *2019 International Conference on Advanced Computing and Applications (ACOMP)*, 2019, S. 127–134

[TD20] Tran, C. N. ; Danciu, V.: A General Approach to Conflict Detection in Software-Defined Networks. 1 (2020), 01. http://dx.doi.org/10.1007/s42979-019-0009-9. – DOI 10.1007/s42979–019–0009–9

[TLF] The Linux Foundation, (LF): *Xen Project*. https://xenproject.org/. – Accessed on 2021-04-30

[TR] Tran, C. N. ; Reyes, N.: *SDN conflicts: Experimental infrastucture*. https://github.com/mnm-team/sdn-conflicts. – Accessed on 2021-09-19

[Tra22] Tran, C. N.: *Conflict Detection in Software-Defined Networks*, Ludwig-Maximilians-Universität München, Diss., 2022

[Uta] Utah, The U.: *Emulab*. https://www.emulab.net/portal/frontpage.php. – Accessed on 2021-08-09

Repository File Structure of the Experimental Infrastructure

The following directory tree serves as a reference for developers and researcher that follow the example deployment of an experiment run 4.5. Files marked with an N are new implementations, those marked with an A are adapted in this work. Other files are unchanged or auto generated.

```
+-- control
|   +-- automating_experiment
|   |   +-- app_choice
|   |   +-- app_config
|   |   +-- app_cookie.txt A
|   |   +-- choose_app_python35.py
|   |   +-- collectdata_checkAppAssets.bash N
|   |   +-- collectdata_config.bash
|   |   +-- collectdata_controller.bash
|   |   +-- collectdata_dataplane.bash A
|   |   +-- collectdata_dataplane_single.bash
|   |   +-- collectdata_function.bash
|   |   +-- collectdata_getAppCookie.bash N
|   |   +-- collectdata_getAppHostAssets.bash N
|   |   +-- collectdata_getAppRouterAssets.bash N
|   |   +-- collectdata_getEPLBserver.bash
|   |   +-- collectdata_getHSserver.bash N
|   |   +-- collectdata_getPEjumps.bash N
|   |   +-- collectdata_stop_and_clean.bash A
|   |   +-- generate_config_python35.py
|   |   +-- iterate_parameter_space.bash N
|   |   +-- parameter_space.bash
|   |   +-- read_parameter_space.bash A
|   |   +-- read_parameter_space_v2.bash A
|   |   +-- switches
|   |       +-- calculate_max_bw_from_netbps_script.bash
|   |       +-- dumpflows.sh
|   |       +-- dumpgroups.sh
|   |       +-- netbps
|   |       +-- tcpdump_fromoutside_bwgauge.sh
|   +-- config_and_start.bash N
|   +-- config_ovs.bash A
|   +-- config_pc.bash
|   +-- config_testbed.bash A
```

```
|   +-- README
+-- controller
|   +-- massive
|   |   +-- app_cookie A
|   |   +-- arpcache.py
|   |   +-- delete_all_flow_and_group_tables_v2_0.py
|   |   +-- detector.py A
|   |   +-- eplb.py A
|   |   +-- firewall.py N
|   |   +-- flowmod_class.py
|   |   +-- fw.py N
|   |   +-- globalconfig.py N
|   |   +-- hs.py N
|   |   +-- http_app.py N
|   |   +-- log.py N
|   |   +-- packetin_class.py
|   |   +-- pe.py N
|   |   +-- plb.py A
|   |   +-- pplb4d.py A
|   |   +-- pplb4s.py A
|   |   +-- README
|   |   +-- rest_fw.py N
|   |   +-- rest_ofctl.py A
|   |   +-- routing_excluded_info
|   |   +-- routing.py A
|   |   +-- topology.py
|   |   +-- utility_detector.py A
|   |   +-- utility.py A
|   |   +-- utility_rest.py
+-- README
+-- sdn_results
|   +-- sample_topology
|       +-- all_config
|       +-- conflict.txt
|       +-- detector_log
|       +-- parameter_space.json
|       +-- router10_dumpflows.tar.gz
|       +-- router11_dumpflows.tar.gz
|       +-- router12_dumpflows.tar.gz
|       +-- router13_dumpflows.tar.gz
|       +-- router14_dumpflows.tar.gz
|       +-- router15_dumpflows.tar.gz
|       +-- router1_dumpflows.tar.gz
|       +-- router2_dumpflows.tar.gz
|       +-- router3_dumpflows.tar.gz
|       +-- router4_dumpflows.tar.gz
|       +-- router5_dumpflows.tar.gz
```

```
|       +-- router6_dumpflows.tar.gz
|       +-- router7_dumpflows.tar.gz
|       +-- router8_dumpflows.tar.gz
|       +-- router9_dumpflows.tar.gz
+-- topogen
    +-- autogen
    |   +-- app_config_generator.py N
    |   +-- constants_generate_spec.py N
    |   +-- generate_sdn_spec.py N
    |   +-- parameter_space_loader.py N
    |   +-- random_topo.py N
    |   +-- sdn_testbed_spec.py N
    |   +-- sdn_testbed_spec_v2.py N
    |   +-- sdn_testbed_spec_v2.yang N
    |   +-- sdn_testbed_spec.yang N
    |   +-- setup_pyang_module N
    |   +-- setup_pyang_module_v2 N
    |   +-- sample_topology.py
    +-- README.md
    +-- README_SDN.md
    +-- scripts
    |   +-- build_sdn_conflicts_experiment.sh N
    |   +-- common_generate_qcow_image.sh A
    |   +-- common_generate_xen_fs.sh
    |   +-- common_generate_xen_image.sh
    |   +-- common_generate_xen_topology.sh N
    |   +-- common_network_helpers.sh A
    |   +-- common_qcow_helpers.sh A
    |   +-- common_tar_helpers.sh
    |   +-- common_xen_helpers.sh
    |   +-- constants_build_env.sh
    |   +-- constants_qcow.sh
    |   +-- constants_xen.sh
    |   +-- deploy_sdn_conflicts_experiment.sh N
    |   +-- deploy_vbox.sh N
    +-- specs
    |   +-- sample_topology
    |   |   +-- qcow.spec
    |   |   +-- xen_con0.spec
    |   |   +-- xen_pc1.spec
    |   |   +-- xen_pc2.spec
    |   |   +-- xen_pc3.spec
    |   |   +-- xen_pc4.spec
    |   |   +-- xen_router1.spec
    |   |   +-- xen_router2.spec
    |   |   +-- xen_router3.spec
    |   |   +-- xen_router4.spec
```

```
+-- tars
|   +-- sdn
|       +-- sample_topology_sdn_control.tar
|       +-- sdn_ssh_config.tar
|       +-- sdn_start_xen.tar
+-- templates
|   +-- template-controller.img A
|   +-- template-kvm-debian9-kernel4.9.0-13-amd64.qcow A
|   +-- template-pc.img
|   +-- template-switch.img
+-- topology
    +-- sample_topology.json
    +-- ...
```

Preliminary Proposition of Conflict patterns

This is a first attempt for a mathematical and formal definition of conflict patterns for the conflict classes described in Chapter 5. There is no guarantee for correctness.

1 Occlusion

Let there be paths p, p', application a, a', action set T, target switch s and rules $x_1...x_n, x'_n$, where $p = \{x_1, ..., x_n\}, p' = \{x_1, ..., x'_n\}, x_n$ is produced by app a and x_n, x'_n reside in switch s, such that if there exists at least one $x_i \in p$ with $i \in 2, n-1$, T are actions in rule x_i and $|T| > 1$, rule $x_n = x'_n$, and if no such rule x_i exists, rule $x_n \neq x'_n$.

2 Bypass

Let there be paths p, p', application a, action set T, rules $x_1...x_n, x'_1...x'_n$ and switches s_i, s_j , where $p_1 = \{x_1, ..., x_n\}$, $p' = \{x'_1, ..., x'_n\}$, $x_1 = x'_1$, $x_n = x'_n$, for all $n > i > 1$ and for all $n' > j > 1$, $s_i \neq s_j$, x_i resides in switch s_i and x'_i resides in switch s_j. If there exists no s_j that is a target switch of application a and there exists at least one x_i with x_i is produced by app a, with T are actions in rule x_i and $|T| > 1$, the transformation action in T can by circumvented on path p'.

3 Injection

Let there be paths p, p', applications a, a', action set T, packets P, P', rules $x_1...x_n, x'_1...x'_n$ and switches s_i, s_j, where $p_1 = \{x_1, ..., x_n\}$, $p' = \{x'_1, ..., x'_n\}$, $x_1 = x'_1$, $x_n = x'_n$, $a \in x_1$ and for all $n > i > 1$ and for all $n' > j > 1$, $s_i \neq s_j$, $x_i \in s_i$ and $x'_i \in s_j$. An injection conflict occurs, if there exists at least one rule x'_i, with x'_i is produced by app a', T are actions in rule x'_i and $|T| > 1$ where both packets P, P' are intended for the same destination endpoint.

4 Loop

Let there be a path p, rules $x_1...x_n$, applications a, a', where $p = \{x_1, ..., x_n\}$, and there exist exactly two $x_i, x_j \in p$, where $x_i = x_j$, $j > i$ and there exist at least two $x_k, x_l \in p$, where x_k is produced by app a, x_l is produced by app a' and $a \neq a'$.

5 Spuriousness

Let there be a path p, rules $x_1...x_n$, applications a, a', action set T, where $p = \{x_1, ..., x_n\}$, $x_1 \neq x_n$, $a \neq a'$, x_1 is produced by app a, T are actions in x_n and x_n is produced by app a' and $T = \emptyset$. Due to the empty action set T, packets that are forwarded along the rule

graph by applications preceding application a', are dropped. Since rule x_n is connected, more precisely the tail end of, other rules in a path, packets that are rejected in a node within the topology, still managed to enter the SDN topology and could be dropped on an edge node.

6 General Multi-Transform

Let there be a path p and rules $x_1...x_n$, applications a, a' and action sets T_1, T_n, with $p = \{x_1, ..., x_n\}$, $x_1 \neq x_n$, $a \neq a'$, $|T_1|, |T_n| > 1$, T_1 are actions in rule x_1 and x_1 is produced by app a, T_n are actions in rule x_n and x_n is produced by app a'.