SN Computer Science (2020) 1:278
https://doi.org/10.1007/s42979-020-00282-0

ORIGINAL RESEARCH

Side-Effects Causing Hidden Conflicts in Software-Defined Networks

Vitalian Danciu[1] · **Cuong Ngoc Tran**[1]

Received: 14 April 2020 / Accepted: 30 July 2020 / Published online: 20 August 2020
© The Author(s) 2020

Abstract

The Software-Defined Networking (SDN) architecture facilitates the flexible deployment of network functions by detaching them from network devices to a logically centralized point, the so-called SDN controller, and maintaining a common communication interface between them. While promoting innovation for each side, this architecture also induces a higher chance of conflicts between concurrent control applications compared to existing traditional networks. We have discovered a new type of anomalies that we call hidden conflicts. They appear to occur only due to side-effects of control application's behaviour and to be independent of and distinct from the class of conflicts between rules present in the network devices. We analyse the SDN interaction primitives susceptible to such disruptions and present experiments supporting our analysis, the result of which indicates the necessity of the knowledge on the control mechanics in detecting hidden conflicts. We present a hidden conflict prediction approach that employs speculative provocation to determine the deployed applications' behaviour. The observed behaviour can be leveraged to predict undesired network state. Evaluation of our prediction prototype suggests that prediction functions should be integrated into control applications.

Keywords Hidden conflicts · Side-effects · Conflict detection · Conflict prediction · Software-defined networks · Speculative provocation

Introduction

In Software-Defined networking (SDN) architecture, the network elements (SDN devices) forming the data plane lack a control plane of their own. The control functions are centralized in a logical component, the so-called SDN controller, that serves as a platform for control applications. These applications issue rules that govern the behaviour of the SDN devices in the data plane. The devices themselves retain only the essential functions for forwarding messages according to the rules stored in their flow tables and to process instructions from the controller.

This article is part of the topical collection "Software Technology and Its Enabling Computing Platforms" guest edited by Lam-Son Lê and Michel Toulouse.

✉ Cuong Ngoc Tran
cuongtran@mnm-team.org
http://www.mnm-team.org/~cuongtran

Vitalian Danciu
http://www.mnm-team.org/~danciu

[1] Ludwig-Maximilians-Universität München, Oettingenstr. 67, 80538 Munchen, Germany

SDN offers a higher degree of flexibility in the specification of network behaviour than is achievable in traditional networks composed of autonomous network elements. The need for control protocols facilitating the negotiation between autonomous elements in traditional networks is eliminated in SDN and replaced by a central specification of network behaviour.

This architectural feature increases the flexibility in specifying network behaviour. In particular, new or experimental network behaviour can be introduced at one single point in the network (the controller) instead of requiring changes to every network element.

This same flexibility renders SDN prone to conflicts between the intents of concurrently active control applications. Different control applications may intend to specify different behaviour that possibly leads to conflicts at the policy level. In other cases, its implementation in terms of rules may include rules conflicting with each other at a technical level.

We consider a conflict to be present when the network's behaviour differs from the expected behaviour, as a result of the combined deployment of control applications. The new type of conflict demonstrated in this paper originates from

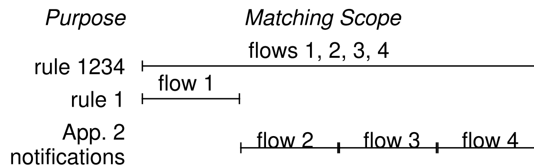

Fig. 1 Scope of the rules issued by the control applications

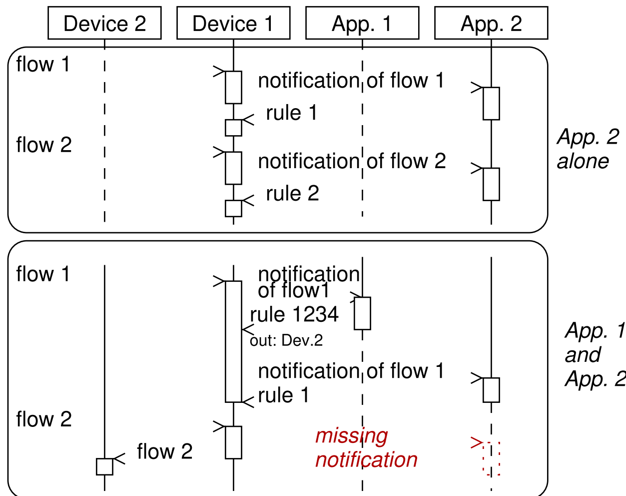

Fig. 2 Interactions of a control application in isolation and when conflicting with another. For clarity, the controller intermediary has been omitted

side-effects and is hidden from analysis of the rules in the data plane alone.

Hidden Conflicts

An instance of a generalization conflict in our experimental setup for conflicts in SDN showed unexpected, anomalous network behaviour different from that described in literature. The generalization conflict class [3, 21] is defined by two rules i and j differing in their action while the match expression of the rule with higher priority describes a subset of the other's: $priority_i > priority_j, match_i \subseteq match_j, action_i \neq action_j$. The consequence of generalization conflicts has been assessed to be minor.

The conflict instance caused by our two control applications conformed to the class definition: one application installed a "broader" rule with a more general match field, while the other of higher priority installed a rule matching a subset of the first. Figure 1 shows the scope of the rules introduced by the applications.

Figure 2 illustrates the case where two control applications verified to function correctly in isolation create a generalization conflict when executed concurrently. The upper box of the sequence diagram shows the simple reactive mechanics of App. 2: it reacts to new flows, of which it is notified, by installing new rules in the (Device 1).

The case of concurrent execution of both applications is shown in the lower box of Fig. 2, in which App. 2 is effectively disabled. Analysis of this behaviour showed the observed effect to be not a consequence of the generalization conflict between rule1 and rule1234 but contingent on the suppression of notifications (or events) issued to the applications. The presence of the broader rule rule1234 resulted in packets interesting to App. 2 being processed locally by the device, instead of being escalated to the controller. Hence, App. 2 was deprived of the notifications it requires to function as expected. A concrete experiment corresponding to this case is presented in "EpLB and TE1" section.

The expectation from the descriptions in literature and the apparent effect of the rules would suggest, indeed, only a minor issue: normally, the broader, low-priority rule would defer to the more specific one of higher priority. In our case, the sheer presence of the broader rule causes suppression of events and thus the failure of one application, as a side-effect of its (correct) handling of incoming packets. An alternative interpretation of this effect is a conflict between the broader rule and the default behaviour of the device to escalate unknown flows to the controller.

We name this type of conflicts hidden conflicts or side-effect conflicts, as their cause cannot be discerned from analysis of the rules in the data plane's devices alone but requires insight into the mechanics of the control plane. Their discovery raises the question how sensitive the SDN control is to side-effects. To address it, we analyse the operational model for OpenFlow SDN [16] to identify potential side-effects that can cause anomalous behaviour in the network.

Synopsis

We first describe a new type of conflicts in SDN that we call hidden conflict. In contrast to the conflict types portrayed in literature, hidden conflicts are not detectable by rule analysis alone. The cause and mechanism of hidden conflicts appear to be orthogonal to those of conflicts between rules. Thus, hidden conflicts appear to be a different dimension of conflicts. Our initial examination of this conflict type shows it to occur due to suppression of the event mechanism as a side-effect of an otherwise conflict-free rule set. Consequently, events necessary for the function of a control application are no longer provided to it.

Aiming to identify all possible side-effect sources, we examine the interaction primitives of SDN in "Analysis" section and identify those combinations of primitives that can be influenced by the operation of a control application. Where possible, we determine the probable observable consequences of such influence. We have conducted

experiments, discussed in "Empirical examination" section, that indicate the consequences of side-effects to be uncorrelated to conflicts between rules. In particular, we demonstrate identical side-effects for two different types of conflicts between rules, as well as the absence of side-effects in a situation in which the influence on the control primitives is removed.

Being deprived of rule analysis as an effective method for the detection of hidden conflicts, we introduce a speculative method to predict hidden conflicts through provocation of side-effects. By issuing surrogate, fake events to a control application, a predictor is capable of observing the application's response behaviour and determining if the state of the data plane, including the rule set, would hinder that behaviour. We describe this approach and the function of our predictor prototype in "Hidden conflict predictor" section.

Our prediction approach is general in that it is not application specific. Conversely, it is initially agnostic of the behaviour of the control applications that are being examined. Thus, the prediction itself may cause undesired effects in the network, in response to the surrogate/fake events issues by the predictor. Similarly, run-time prediction may cause race conditions between genuine and fake events. We discuss these issues in "Discussion" section. We review related research on conflicts and their detection in "Related work" section, including race conditions and conflicts between rules.

We conclude in "Conclusion" section by highlighting properties of the hidden conflict class and propose that predictor code should be included into applications at design time. The specialization of our prediction approach to render it useful during control application development is an interesting topic for further study.

This article extends our previous work [28], that introduced the notion of hidden conflicts, by emphasizing the following points:

- We elaborate the implementation details of the hidden conflict predictor (Sect. "Hidden conflict predictor") ranging from the prediction mechanism, the choice of candidate traffic for generating fake events and the interception of methods as reactions of control applications once receiving the fake events.
- We discuss the properties of the hidden conflict predictor, the challenges encountered during its deployment and find that they suggest the integration of predictor code in control applications to be beneficial.
- We indicate interesting research directions based on the analysis of the limitations of our own work, particularly when dealing with dynamic topology change and for choosing a fixed matching policy during the examination of conflicts.

Analysis

We discuss the methodology to analyse the hidden conflicts described in "Introduction" section, which requires the introduction of the interaction primitives between SDN participants at different layers. We infer the disturbance factors that could lead to hidden conflicts and their possible impact.

Methodology

The occurrence of the conflict instance demonstrated in "Hidden conflicts" section is contingent on an influence of one of the applications on the control mechanics of the other. Our examination therefore targets the potential influences exerted by applications and the SDN control mechanics that are susceptible to each influence.

As a starting point, we use an analytic examination method in that we decompose the operational model of OpenFlow into primitive interactions between the devices, the controller and the control applications. Such interactions are triggered by events in the network, including packets arriving at a device. Each combination of interactions is a candidate for influence. We assess each candidate with respect to its susceptibility to influence, i.e. we enumerate the conditions in which the interaction can be disturbed. For each of these susceptible candidates we attempt to assess the impact on an application whose correct function relies on it. Thus, we acquire a conflict model, that includes (i) the susceptible primitive interaction combinations, (ii) the conditions in which they may be influenced and (iii) the potential impact on the function of an application relying on a given interaction combination at a time when one of the conditions is met. We validate the model in "Empirical examination" section by documenting experiments that support our analysis.

Interaction Primitives

From the study of the OpenFlow specification [1] we extract the basic actions of the devices and a controller. Since OpenFlow does not specify a north-bound interface, we define the interaction between controller and applications to consist of a controller interface and an event system, as it is commonly implemented in SDN controller software.

We list the trigger events along with the possible actions being triggered in Tables 1, 2 and 3.

[1] https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.3.5.pdf

Table 1 Device primitives

Events	Actions
Packet	Out
Flow removal by internal timer [2]	Drop
Packet with action	Modify
Device/link/port startup	Escalate data packet
Device/link/port shutdown	Escalate status
Device/link/port failure	Escalate notification
Device's state query	Modify and out
	Modify and escalate

[2] Flow timeout

Table 2 Controller primitives

Events	Actions
Device/link/port enabled	Publish event
Device/link/port disabled	Instal rule
Device/link/port failure	Modify rule
Packet escalated	Delete rule
Notification escalated	Nop
Function call from application	
Device's state response	

Table 3 Application primitives

Events	Actions
Startup	Instal rule
Shutdown	Modify rule
Packet-in	Delete rule
Topology change	Delegate packet to device [3]
Notification	Nop
Device's state response	

[3] By sending packet-out

Interaction Combinations

We assume that devices and applications do not interact directly and thus all interactions are relayed and translated by the controller. We note that some combinations are impossible in practice. We constrain the actions listed to those pertaining to an interaction and refrain from assumptions on the internal actions of controller and applications. Some of the combinations shown in Table 4 can be eliminated from further analysis. These include:

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

It is possible to generate mock events, i.e. events that have no base in an actual state change in the network, to exploit the north-bound interface, either for productive use, e.g. to diagnose a network problem, or with malicious intent. The combinations of the interactions between applications and the controller, shown in Table 5, result from the mock events being introduced at the controller level or at the application level. Note that a mock event could be any of the events from Table 4 intended for the controller or the application.

Disturbance Factors

Network behaviour can be influenced (negatively) by the disruption of the interaction between the actors. In the following, we list disturbance factors that have been observed in an experiment or that are conceivable and give an example of how they can disturb the mechanics of an application.

Event suppression by local handling A switch handles an incoming packet locally, instead of escalating it to the controller. Consequently, the application is deprived of the event notification. An illustration for this disturbance factor is shown in Fig. 2: the presence of rule 1234 results in the missing notification for App. 2 when flow 2 arrives at Device 1. Experiments 3.2.1 and 3.2.2 also reveal the same effect.

Event suppression by changes to paths Prevention of escalation by changes to paths, e.g. when a packet interesting to an application is routed around the switch holding a rule that would escalate the packet to the controller.

Action suppression by packet modification A device executing rules that modify packet fields before the packet is escalated by itself or by subsequent devices could modify the packet so that it is no longer accepted within an application's scope. For example, application A1 instals a rule on switch S1 to modify all packets to D1 by changing the destination to D2 before sending these packets out of S1–S2. Application A2 is interested in the traffic destined to D1 and subscribes to event E originated from switch S2. As a result, the event escalated at switch S2 is ignored by A2.

Undue trigger Conversely to Action suppression by packet modification, an application can be "tricked" into, e.g. installing or removing rules by packets modified before escalation. This can happen in the course of an attack by mock packets sent by attackers.

Tampering with event subscription This disruption is contingent on applications being able to modify each other's subscriptions. In that case, an application might cause "undue trigger" disruption to another application or simply suppress events by unsubscribing events for it. This case can also happen as a result of an attack.

Table 4 Combinations of interaction primitives

#	Device		Controller		Application		Disturbance factor	Note
	Event	Action	Event	Action	Event	Action		
1	Startup	Escalate status	Device/port/link enabled	Publish event	Topology change	NoP		
2	Startup	Escalate status	Device/port/link enabled	Publish event	Topology change	Instal rules	a,b,c	
3	Startup	Escalate status	device/port/link enabled	Publish event	Topology change	Delete rules	a,b,c	
4	Startup	Escalate status	Device/port/link enabled	Publish event	Topology change	Packet-out	a,b,c	
5	Shutdown	Escalate status	Device/port/link disabled	Publish event	Topology change	NoP		
6	Shutdown	Escalate status	Device/port/link disabled	Publish event	Topology change	Instal rules	a,b,c	
7	Shutdown	Escalate status	Device/port/link disabled	Publish event	Topology change	Delete rules	a,b,c	
8	Shutdown	Escalate status	Device/port/link disabled	Publish event	Topology change	Packet-out	a,b,c	
9	Failure	Escalate status	Device/port/link failure	Publish event	Topology change	NoP		
10	Failure	Escalate status	Device/port/link failure	Publish event	Topology change	Instal rules	a,b,c	
11	Failure	Escalate status	Device/port/link failure	Publish event	Topology change	Delete rules	a,b,c	
12	Failure	Escalate status	Device/port/link failure	Publish event	Topology change	Packet-out	a,b,c	
13	State query	State response	D. state response	Publish event	D. state response	NoP		
14	State query	state response	D. state response	publish event	D. state response	instal rules	a,b,c	
15	State query	state response	D. state response	publish event	D. state response	delete rules	a,b,c	
16	State query	state response	D. state response	publish event	D. state response	packet-out	a,b,c	
17	Packet	Out	–	–	–	–		Device only
18	Packet	Drop	–	–	–	–		Device only
19	Packet	Modify	–	–	–	–		Device only
20	Packet	Escalate	Packet esc.	Publish event	Packet-in	NoP		
21	Packet	Escalate	Packet esc.	Publish event	Packet-in	Instal rules	a,b,c	
22	Packet	Escalate	Packet esc.	Publish event	Packet-in	Delete rules	a,b,c	
23	Packet	Escalate	Packet esc.	Publish event	Packet-in	Packet-out	a,b,c	
24	Flow timeout	Escalate notification	Notification esc.	Publish event	Flow removed	NoP		
25	Flow timeout	Escalate notification	Notification esc.	Publish event	Flow removed	Instal rules	a,b,c	
26	Flow timeout	Escalate notification	Notification esc.	Publish event	Flow removed	Delete rules	a,b,c	
27	Flow timeout	Escalate notification	Notification esc.	Publish event	Flow removed	Packet-out	a,b,c	

Table 5 Mock events based on the interaction primitives

Controller		Application		Disturbance factor	Note
Event	Action	Event	Action		
Mock event	Publish event	Event	NoP		Mock event sent by apps relayed via controller
Mock event	Publish event	Event	Instal rules/ delete rules/ packet-out	a,b,c,d,e	Mock event sent by apps relayed via controller
		Mock event	NoP		App sends mock event directly to app
		Mock event	Instal rules/ delete rules/ packet-out	a,b,c,d,e	App sends mock event directly to app

Susceptible Interactions and Impact

The combinations shown in Tables 4 and 5 may be susceptible to one or more of the disturbance factors. We analyse each of them to determine which, if any, disturbances they are sensitive to and the result are shown in the Disturbance factor column of these tables. In our analysis, we have determined a combination to be sensitive if it is conceivable that one of the disruption factors may be able to disturb its process. We present only the results and the effects we consider

possible consequences of a disruption. Unsurprisingly, these effects strongly relate to the purpose of the interaction set. They include missing rules, redundant rules or wrong rules in one device or more, which may cause anomalous network behaviour. We note that the suppression of handling and the suppression of events appear prevalent in our assessment of the susceptibility of interaction combinations.

In partial validation of our analytical assessment on the disruption of interaction primitive combinations, we present selected experiments in "Empirical examination" section. One of them is a detailed description of the motivating example sketched in "Introduction" section.

Empirical Examination

We conduct two experiments to demonstrate the consequences of side-effects on applications operating in reaction to events issued by the SDN controller. In another experiment, we show that side-effects do not occur at all for event-free applications.

Experiments are deployed on the topology shown in Fig. 3. The testbed are built based on virtual machines as described in [7]. We use the Ryu SDN framework[2] for SDN controller with OpenFlow1.3 as the controller southbound API. Open vSwitch [20] with OpenFlow support is employed for SDN switches. Traffic among end-points is generated by common tools: iperf [3], nc and ping.

Applications for Experiments

We employ several control applications, that are run concurrently in different combinations, depending on experiment. They are described in the following.

Shortest Path First (SPF)

The SPF application uses the topology information provided by the controller to realize the shortest path first routing function for all common kinds of traffic: ARP, ICMP, TCP, UDP.

SPF can be configured to deploy rules in two manners for IP traffic including ICMP, TCP, UDP:

- *SPF1* the rule's match field includes: source IP address, destination IP address, IP protocol number [4]
- *SPF2* the rule's match field includes only destination IP address.

[2] https://ryu.readthedocs.io/en/latest/.

[3] https://iperf.fr/

[4] https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml

End-point load balancer (EpLB)

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

In the experiment presented in this paper, EpLB is deployed on switch S7 to balance the UDP/TCP sessions between PC3 and PC4. The first incoming session destined to PC3 will be sent to PC3, the second session to PC3 will be changed to PC4 by rewriting the destination information of the relevant traffic, the third will come to PC3 and so on. The balancing operation is transparent to end users in that the traffic in response from PC4 to the original source will be changed to appear as if it was sent from PC3.

Traffic Engineering (TE)

In the role of a network administrator, we employ the Ryu REST API [5] to pro-actively perform traffic engineering in two different manners in different experiments. Note, that in these experiments the TE "application" has been simulated by manual entry of the flow rules, however an actual application for the Ryu controller performing these actions automatically in response to policy configuration is easily conceivable. Due to the intended function of the application (static configuration of flows), its only benefit compared to the manual input would lie in the automation of the task.

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

Experiments

Table 6 shows the settings for the experiments according to the concerned factors discovered in our earlier work [26, 27]. Each application is deployed with only one configuration in each experiment, they may start at the same time or one

[5] https://ryu.readthedocs.io/en/latest/app/ofctl_rest.html

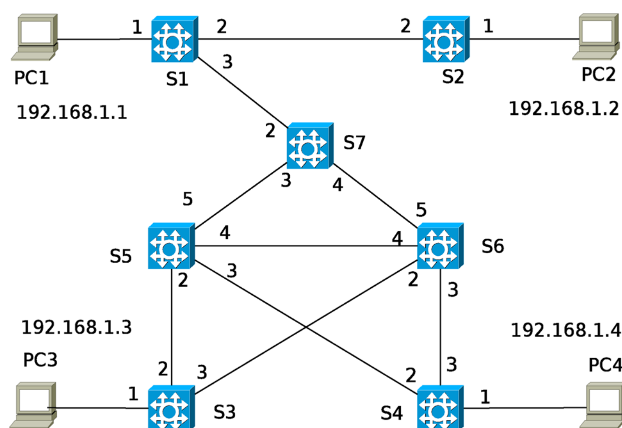

Fig. 3 Topology for the experiments. The numbers surrounding a switch indicate the port number assigned by the SDN controller

after another and their rules may have different priority or the same. The routing application SPF1/2 is necessary for the functioning of the network, so it will affect all switches while the EpLB and TE1/2 deploy their rules on switch S7 only. In our experiments, traffic sources are from PC1 and PC2 while traffic sinks are on PC3 and PC4. We employ the constant bit rate (CBR) traffic profile for all related endpoints in the experiment. The network topology is shown in Fig. 3. We test with UDP traffic in the first and second experiments and a mix traffic of TCP/UDP in the third one.

EpLB and SPF1

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

Observation and analysis Further UDP sessions from PC1 to PC3 cannot be balanced as expected. All the next UDP sessions from PC1 always come to PC3 while they were meant to be alternately handled by PC3 and PC4.

The problem can be identified by comparing the flow entries 2 and 6 highlighted in Table 7. EpLB features a UDP session by additional information of layer 4 source and destination ports as reflected in flow entry 2. It is supposed to instal new flow entries to handle further UDP traffic from PC1 to PC3 having different combination of layer 4 source–destination ports when being triggered by the corresponding packet-in events for this kind of traffic from the controller. However, since flow entry 6 matches the mentioned incoming traffic already, no packet-in event will be generated. As a consequence, the EpLB intention cannot be achieved.

Table 6 Experimental settings

Dimensions	Exp. 1		Exp. 2			Exp. 3	
	EpLB	SPF	EpLB	TE	SPF	TE	SPF
App config.	1	SPF1	1	TE1	SPF1	TE2	SPF2
App start order	1	1	1	2	1	2	1
App priority	2	1	2	1	1	2	1
Target switches	7	all	7	7	all	7	all
Ept traf.prof.	CBR	CBR	CBR	CBR	CBR	CBR	CBR
Ept combi.	{PC1,PC2} −− > {PC3,PC4}		{PC1,PC2} –> {PC3,PC4}			{PC1,PC2} –> {PC3,PC4}	
Topology	1		1			1	
Transport type	UDP	UDP	UDP	UDP	UDP	TCP, UDP	TCP, UDP

Table 7 Experiment 1: switch S7's flow table after the first UDP session

#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action
1	Ctrl	65535	-	01:80:c2:00:00:00	LLDP						Ctrl
2	EpLB	2	-	-	-	192.168.1.1	192.168.1.3	UDP	48834	5001	output:3
3	EpLB	2	-	-	-	192.168.1.3	192.168.1.1	UDP	5001	48834	output:2
4	SPF1	1	00:16:3e:00:00:41	00:16:3e:00:00:43	ARP						output:3
5	SPF1	1	00:16:3e:00:00:43	00:16:3e:00:00:41	ARP						output:2
6	SPF1	1	-	-	-	192.168.1.1	192.168.1.3	UDP	-	-	output:3
7	SPF1	0									Ctrl

The highlighted rule pair 2 and 6 expose the redundancy conflict pattern, other rules (1,3,4,5,7) cause no problem at all

Table 8 Experiment 2: switch S7's flow table after the first UDP session and deploying TE1's rules

#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action
1	Ctrl	65535	-	01:80:c2:00:00:00	LLDP						Ctrl
2	EpLB	2	-	-	-	192.168.1.1	192.168.1.3	UDP	38643	5001	output:3
3	EpLB	2	-	-	-	192.168.1.3	192.168.1.1	UDP	5001	38643	output:2
4	SPF1	1	00:16:3e:00:00:41	00:16:3e:00:00:43	ARP						output:3
5	SPF1	1	00:16:3e:00:00:43	00:16:3e:00:00:41	ARP						output:2
6	TE1	1	-	-	-	-	192.168.1.3	UDP	-	5001	output:4
7	SPF1	0									Ctrl

The highlighted rule pair 2 and 6 expose the generalization conflict pattern while others cause no problem

Table 9 Experiment 3: switch S7's flow table after establishing TCP sessions from PC1 to PC3 and PC4 and deploying TE2's rules

#	App	Prio	L2:Src	Dst	Type	L3:Src	Dst	Prot	L4:Src	Dst	Action
1	Ctrl	65535	-	01:80:c2:00:00:00	LLDP						Ctrl
2	TE2	2	-	-	-	-	192.168.1.3	TCP	-	-	output:3
3	TE2	2	-	-	-	-	192.168.1.4	TCP	-	-	output:4
4	SPF2	1	00:16:3e:00:00:41	00:16:3e:00:00:43	ARP						output:3
5	SPF2	1	00:16:3e:00:00:43	00:16:3e:00:00:41	ARP						output:2
6	SPF2	1	00:16:3e:00:00:44	00:16:3e:00:00:41	ARP						output:2
7	SPF2	1	00:16:3e:00:00:41	00:16:3e:00:00:44	ARP						output:3
8	SPF2	1	-	-	-	-	192.168.1.3	-	-	-	output:3
9	SPF2	1	-	-	-	-	192.168.1.1	-	-	-	output:2
10	SPF2	1	-	-	-	-	192.168.1.4	-	-	-	output:3
11	SPF2	0									Ctrl

The rule pair 2 and 8 highlighted in blue expose the redundancy conflict pattern, the rule pair 3 and 10 highlighted in green expose the generalization conflict pattern. Other rules cause no problem

EpLB and TE1

SPF1 is modified to work in concert with the EpLB and TE1 in this experiment, so its rules will be overwritten or not deployed at all where EpLB's or TE1's rules are active. EpLB balances sessions between PC3 and PC4 where PC4 acts as a replica of PC3. TE1 instals static rules to direct all UDP traffic having the specified destination port (5001 in this case) to PC3. The flow table of switch S7 has only rules 1, 7 (controller management rules) in the beginning of the experiment. Rule generation happens on first traffic both for EpLB and SPF1. In the role of an administrator, we instal TE1 rules later via REST API. This experiment shows the importance of the application deployment order (Table 8).

Observation and analysis Similar to the first experiment, EpLB is completely disabled for subsequent UDP sessions having the destination port of 5001 after the TE1 rule becomes effective.

Two flow entries 2 and 6 are identified to be responsible for the problem and are highlighted in Table 8. Again, since flow entry 6 is more general in that it matches only the destination IP address and the destination UDP port, further UDP sessions having these fields will be handled by this flow entry and no packet-in event will be generated to keep EpLB functioning correctly.

TE2 and SPF2

This experiment shows that side-effects do not happen at all when the application with more specific rules does not operate on the basis of the packet-in event (Table 9).

The flow table of switch S7 has only rules 1, 11 (controller management rules) in the beginning. Rule generation happens on first traffic by SPF2. TE2 rules are installed subsequently.

Observation and analysis The flow rules 2 and 8 follow the *redundancy* conflict pattern [3, 21], which features a similar relationship between two rules as in the generalization pattern but their actions are the same and their priority relationship does not matter. The flow rules 3 and 10 exhibit the generalization conflict pattern. The network behaves as expected for the main effect and there is no side-effect at all: all TCP traffic to PC3 and PC4 are forwarded according to rules 2 and 3, other traffic, e.g. UDP, ICMP is controlled by SPF2 rules.

Hidden Conflict Predictor

The hidden conflict demonstrated in the experiment described in "EpLB and SPF1" section leads to the rather severe consequence of a control application becoming ineffective. To protect the correct behaviour of the network, it is therefore necessary to detect this class of conflicts. Unfortunately, hidden conflicts cannot be detected by mere analysis of the data plane's flow tables (the collection of flow tables of all devices in the data plane). The assertion of their presence requires information on the control plane's behaviour in a certain state, in reaction to an event.

Full knowledge about the control plane's behaviour includes all combinations of control application action options given the state of the data plane's flow tables and

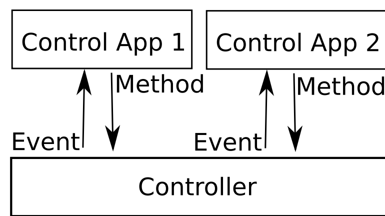

Fig. 4 Application-controller communication [9]

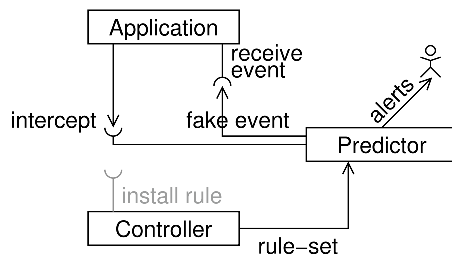

Fig. 5 Interaction of the predictor component with controller and control applications

the incoming traffic at the data plane. In any practical case, this level of knowledge about the network is obviated by several of its properties:

- We may not know exactly the control application's behaviour. This is conceivable in SDN since control applications can come from different parties.
- The behaviour of a control application varies generally according to the network state while the network state also changes from time to time, and mostly in an unpredictable fashion.
- The incoming traffic at the data plane is also unpredictable, e.g. end-points can generate diverse traffic types (TCP, UDP, ICMP...) with different traffic profiles (CBR, VBR, bursty...) and in different groups (unicast, multicast).

To address this issue, we have experimented with speculative provocation of conflicts as a method to predict the creation of conflicts. We rely on a conflict predictor that selects possible conflict situations ("speculative") and simulates situations in which the applications may issue rules conflicting with the existing rule set ("provocation").

We exploit the interaction between an SDN controller and the control applications being realized by the event/method mechanism [9] illustrated in Fig. 4. A control application registers as a listener for certain events, and the controller will invoke the application's callback method whenever such events occur.

Our chosen SDN controller for experiments, the Ryu SDN framework, also complies to this model and facilitates the conflict predictor implemented as a built-in control application in the controller to create and dispatch events to other control applications. In our experiments, the predictor generates the packet-in events associated with the candidate packets to provoke the reactions of the control applications which register to this event type; the choice of the candidate packets is elaborated in "Choice of candidate traffic" section. In practice, the predictor can generate any type of event, e.g. those related to topology change, SDN devices' state change.

Prediction Mechanism

The procedure has three main steps in which the predictor component interacts with the controller and control applications, as illustrated in Fig. 5:

1. The predictor analyses the rule tables and determines what type of additional rules would lead to conflict. The potential additional rules correspond to those completing one of the conflict patterns.
2. The predictor provokes control applications into generating such rules, by having the controller issue them fake events and subsequently intercepting the calls for rule installation in response to the fake events. As with the additional rules, the content of the fake events is derived from the conflict patterns describing a conflict class. Thus, the predictor attempts to provoke a specific class of conflict.
3. Each intercepted rule installation call is analysed by the predictor to determine if an actual installation of that rule would create a conflict.

These steps can be performed in parallel for several classes of conflicts in several situations. Therefore, this method allows for a trade-off between expended computing power and detection latency. In addition, it relies on the network state at the time of prediction, thus limiting the number of cases that would need to be probed.

In the followings, we demonstrate how a conflict predictor can be realized by elaborating the above steps for a subset of hidden conflicts related to the generalization and redundancy conflicts identifiable in data plane's flow tables.

Choice of Candidate Traffic

One of the necessary conditions for two rules i and j to expose a generalization or redundancy conflict is that the matching scope of one rule is "broader" than the other. The more specific rule must have higher priority for *generalization*

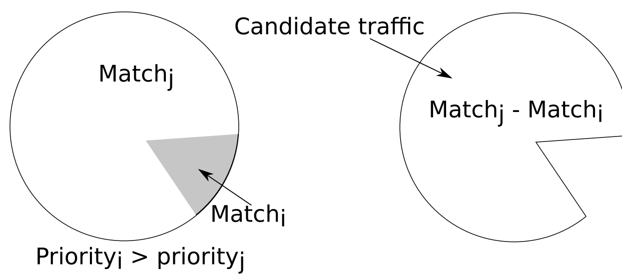

Fig. 6 Candidate traffic for probing hidden conflicts

conflict (cf. Sects. 1.1 and 3.2.3). Without loss of generality, we assume that: $\text{priority}_i > \text{priority}_j, \text{match}_i \subseteq \text{match}_j$.

The experiments in "Empirical examination" section show that hidden conflicts occur as one of the control application is deprived of the events it requires to function. These events are expected to be generated for the traffic being matched against the more general rule (rule j). This observation indicates that the candidate traffic for generating the fake event to probe hidden conflicts must belong to the set difference of the set created from the matching scope of rule i and the set from that of rule j, as illustrated in Fig. 6.

An exemplary candidate packet to probe hidden conflicts deduced from rules 2 and 6 in Table 7 would have header fields:

$$L3 : src = 192.168.1.1, dst = 192.168.1.3,$$
$$\text{Prot} = \text{UDP}, L4 : src = 50000, dst = 5001$$

Note that a packet for the fake event generation must be complete, i.e, the above candidate packet requires layer 2 headers. Assuming it is an Ethernet frame, this includes source and destination MAC addresses and the EtherType value. The MAC addresses can be set to arbitrary values or given correct values obtained from an ARP cache control program such as the one described in [13].

Interception of Methods

The predictor has to supervise the rule deployment of the control applications as a result of its fake event generation in order to intercept this action and determine the presence of hidden conflicts. For this reason, we have implemented the predictor as an application integrated in the controller and providing the add-flow interface for other control applications to instal their rules in data plane devices. We notice another possibility in deploying the predictor as an independent program like any other control application, which appears more elegant but causes more overhead in communication between the rule deployment module of the controller and the predictor and higher latency in rule installation process. Besides, the predictor may well be part of an orchestrator who logically situates centrally below all control applications and moderates their actions, which advocates our choice.

The pseudo-code sketching the prediction procedure is shown in the Algorithm 1. The predictor uses the controller interfaces to regularly pull the data plane's flow tables every interval period, analyses them to detect conflicts based on the provided conflict patterns (e.g. redundancy, generalization). If conflicts exist, a conflict_flag is set and the predictor will choose and create candidate packets to generate fake events associated with them. For each generated event, there may be multiple reactions from different control applications. During the conflict_flag_timeout period when the conflict_flag is set, all calls to the add-flow function by control applications will be checked if their rules to be installed correspond to the generated fake events and whether installing these rules in the data plane would cause conflicts there; if yes, an alarm of the likelihood of the hidden conflicts relevant to the chosen candidate packets is raised. A rule to be installed in the data plane is asserted to be corresponding to a generated fake event if its matching scope covers the chosen packet for that event. Thus, the predictor can decide whether a method issuing a rule will create a conflict. This is the case if the new rule would contradict one of the existing rules in the device where it would be installed, i.e. they have overlapping matching scope but different actions.

Algorithm 1 Pseudo-code for the predictor

```
1: Input: Conflict patterns (e.g. redundancy, generalization)
2: Output: Alarm of possible hidden conflicts is raised if any
3: Global variables: interval, conflict_flag, conflict_flag_timeout
4: Note: Function ADD-FLOW is called by control applications to install rules in data plane
5: function MAIN
6:     conflict_flag = 0
7:     spawn_thread(PULL-AND-CHECK-DATAPLANE-FLOWTABLES)
8: end function
9: function PULL-AND-CHECK-DATAPLANE-FLOWTABLES()
10:    while true do
11:        pull data plane's flow tables
12:        analyze data plane's flow tables to detect conflicts based on provided patterns
13:        check if there exist rules complying with provided patterns
14:        if true then
15:            conflict_flag = 1
16:            choose and build candidate packets
17:            generate fake events corresponding to built packets    ▷ e.g. packet-in event
18:        end if
19:        spawn_thread(CLEAR-CONFLICT-FLAG)
20:        sleep for interval period
21:    end while
22: end function
23: function CLEAR-CONFLICT-FLAG()
24:    sleep for conflict_flag_timeout period    ▷ conflict_flag_timeout < interval
25:    conflict_flag = 0    ▷ check hidden conflict only during conflict_flag_timeout period
26: end function
27: function ADD-FLOW(datapath, priority, match, action, metainfo)    ▷ metainfo:
    cookie, flow timeout ...
28:    create a rule from the tuple priority, match, action, metainfo
29:    if conflict_flag == 1 then
30:        check if rule corresponds to the generated events
31:        if true then
32:            check if installing rule in device datapath would cause conflict in data plane
33:            if true then
34:                raise alarm of the possibility of hidden conflicts relevant to chosen packets
35:            end if
36:        else
37:            install rule in device identified by datapath normally
38:        end if
39:    else
40:        install rule in device identified by datapath normally
41:    end if
42: end function
```

Discussion

The experiments presented in "Empirical examination" section were selected as to show the independence of hidden conflicts from those between rules, that have been described in literature. Fig. 7 illustrates the combination of single cause–effect pairs of hidden conflicts from the experiments correlated with named conflicts between rules. The arrangement shows the same hidden conflict for two conflicts between rules, in cases 1 and 2 in Fig. 7, described in "EpLB and SPF1" and "EpLB and TE1" sections, respectively. It also shows, that the effect reverts to that of the conflict between rules if the cause for the

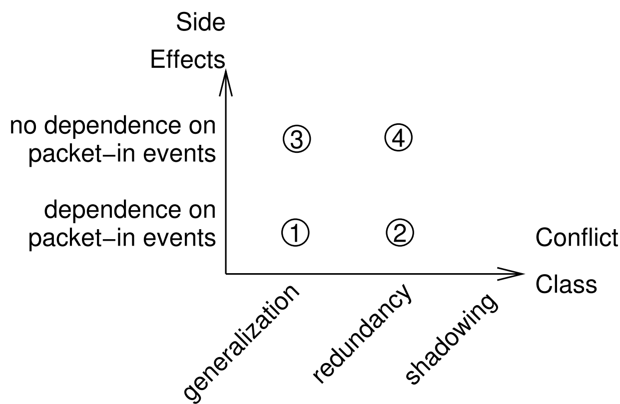

Fig. 7 Orthogonality between hidden conflicts and those described in literature

hidden conflict is removed, as illustrated by cases 3 and 4, described in "TE2 and SPF2" section. This indicates, that hidden conflicts form a dimension of their own, i.e. they are orthogonal to the classes of conflicts between rules. The independence of the two dimensions raises interesting questions.

Challenges to Hidden Conflict Detection

One important question is whether the presence of a conflict between rules is necessary at all in order for applications to exhibit a hidden conflict. If so, the detection of the conflict pattern specified for the rules can be used as a starting point of the search for an associated hidden conflict, even if the effect of the conflict between rules were negligible. If not, then the abnormal behaviour can be said to represent a hidden conflict purely between the assumptions of applications on their environment, that occurs depending on the type of side-effect present.

Properties of the Hidden Conflict Predictor

The detection of hidden conflicts represents a different challenge from the detection of the conflict classes hitherto described in literature. Given that some of the mechanisms leading to abnormal behaviour in the network are hidden within the application code (hence, hidden conflict), we require a black-box analysis approach of the applications as a prerequisite of conflict detection. Our hidden conflict predictor described in "Hidden conflict predictor" section follows such a black-box approach. As such, it is initially agnostic of the behaviour of the control application for which it tries to predict. This allows the approach to be employed with any control application, while also introducing inaccuracy and risks of interference.

Accuracy

The predictor cannot conclude the existence of a hidden conflict with absolute certainty as detection is contingent on the prospective state of the data plane. Hidden conflicts manifest only if the anticipated events and/or the chosen candidate packets (cf. Sect. "Choice of candidate traffic") occur. The results of the predictor can be improved by providing it with management information regarding the data plane. For example, to leverage hidden conflict prediction during maintenance, a predictor could make use of the maintenance schedule, the concrete maintenance activities and the target state of the data plane. It can then be used to predict the reaction of control applications in response to the planned management activities. The predictor can acquire information about end-points (necessary for the packet-in event) from the data plane's flow tables or by consulting end-point discovery applications such as an address resolution proxy [13] if available.

Discrimination of Reactions from Fake Events and Genuine Events

The predictor issues fake events to a control application and records its reaction. However, the application may concurrently receive genuine events and exhibit a reaction to them, as well. Thus, the predictor must be able to differentiate between reactions from fake and genuine events in order to intercept the former and ignore the latter ones. As a partial solution, we have demonstrated the association of an issued rule with a previously generated fake packet-in event by matching its scope to that of the candidate packet for that event. A comprehensive solution to this problem remains for further study. We note that a wrong association of a rule installation request due to a genuine event to a fake event will lead to a benign effect in our current demonstrating predictor prototype: that rule is checked for possible conflict consequence in the data plane and an alarm is raised if conflict may arise.

In cases where the need for discrimination is to be avoided, the race condition between events can be eliminated by mutual exclusion of genuine and fake events.

Poisoning of Control Application State

Our prediction approach makes the quiet assumption that an application exhibits idempotent behaviour. The assumption is not unreasonable, given the reactive nature of many network functions formulated as applications. However, more

sophisticated network software may be designed to hold and evaluate network state separate from the controller's. Also, it may change its behaviour in response to the frequency of certain events. In such cases, attempting to provoke reactions from the application by issuing it surrogate or "fake" events may cause the corruption of its internal state or a change in the strategy used by the application to perform its functions.

For example, a round-robin end-point load balancer balancing traffic between two servers may assume that it has directed flow to the first server in response to a fake event from the predictor. Consequently it will assign the next flow to the second server in response to the next genuine event, thus creating an imbalance. Similarly, an ARP cache application may cache the wrong association of an IP address to a MAC address present in the fake packet-in event.

Applications can avoid state poisoning by verifying that the intended change has been implemented in the data plane. For instance, the above-mentioned load balancer may check if its rule to forward the traffic is present in the data plane's flow tables and rectify its state accordingly, the ARP cache may reexamine the existence of the end-point via its discovery mechanism before putting its information into use.

However, we cannot rely on such verifications being included in the applications' behaviour. Hence, if such issues are diagnosed, it may be prudent to record the results of the provoked reactions and re-use them in subsequent executions of the application in question.

Hidden Conflict Prediction within Control Applications

A predictor can be integrated in a control application to detect hidden conflicts possibly impacting its behaviour. Our predictor (Sect. "Hidden conflict predictor") functioning as a black-box analysis approach probes for control applications' reactions to determine their behaviour. In contrast, a predictor integrated within the control application has the advantage of full knowledge of the application's behaviour and the state it holds. Hence, it can predict hidden conflicts with higher certainty, though in narrower scope pertaining only to that application, without the risk of state poisoning.

Our experiments have shown that hidden conflicts may render the control application inactive in the data plane, leading to severe network faults. At the same time, the drawbacks of external prediction pose risks for malfunction, as well. Hence, we recommend that the design of control applications includes an integrated hidden conflict prediction specific to the application. We envision that the functional primitives necessary for such prediction (conveying the fake events to internal application code, evaluating the resulting reaction) may be collected in a common library shared between application developers.

Hidden Conflict Handling

Our work focuses on detection and eschews handling strategies for the time being. Therefore, we refrain from specifying concrete measures as a reaction to positive results from the predictor. Although automated handling of conflicts is desirable, it is impossible to determine the importance to a rule being issued or the "reason" of the application for issuing it. Thus, the obvious alternatives for conflict resolution appear unsatisfactory. The conflict may be avoided if

- the creation of rules is suppressed, or if the existing rules they conflict with are removed or altered. However, the effects of such changes cannot be evaluated with respect to their impact on the compliance with network management policy.
- the application provoked into issuing conflicting rules is disabled. Similarly to the removal of rules, the effects on network service cannot be determined beforehand.

Alerting the network administrator delegates the problem of understanding the conflict to a human, but it does not constitute actual handling of the conflict. In addition, the handling of the conflict is relegated to a much wider time-frame, compared to an attempt at automatic handling. However, until reliable resolution strategies are available, warning network management seems the most responsible manner of reacting to a potential conflict. It is conceivable that applications will incorporate probing for potential conflict situations themselves.

Limitations

The study of the dynamic and distributed aspects of conflicts within rule sets introduced in this paper aims to go beyond the existing local and static view of conflict detection. However, there remain dynamic aspects of SDN which we have not taken into account. Philosophically, conflicts occur because of different assumptions in concurrent applications. Therefore, any violation of such assumptions bears the risk of conflict. The basis for such assumptions is tied to the behaviour or the state of an application, i.e. the program being executed as an application or the information about the network that the application may hold itself.

Topology Change

It seems plausible that if switches or links are added or removed from the network or if they fail, then the resulting change in topology may either trigger new conflicts or render the existing rules conflicting, as the assumptions possibly made by the issuing application are invalidated. This may be an interesting topic for future study.

Matching Policy

OpenFlow SDN devices employ a first-match policy when choosing which rules to apply to a packet. In principle, SDN devices could employ other strategies as an alternative or as an option, e.g. exact matching or most-specific-first. To some degree, the conflict instances we study are tied to the first-match policy employed by the devices in our experimental setup. Thus, the conflict patterns we find and the detection code created from it is dependent on a first-match processing of the rule tables. We consider this to be a minor limitation, technologically, as first-match appears to be the most common policy in rule-based packet processing.

It is an interesting question whether a different matching policy may influence the occurrence of conflicts: first, whether conflict classes can be independent of matching policy, i.e. they would occur in some form under any choice of policy; and second, if the choice of matching policy would reduce the propensity for conflict in SDN. When comparing the results of the first-match and most-specific-first policies in the cases exemplified in this text, we find the conflict occurring in both cases. One would expect that the local generalization conflict would be eliminated by using a most-specific-first policy, as that policy would ensure the application of the less general rule irrespective of the rules' priority values. However, any packet not matching the more specific rule would, under a most-specific-first policy, be handled by the general rule, leading to the same effect as under the first-match policy.

While this observation is hardly conclusive on its own, a more in-depth study of the influence of matching policy on conflict emergence may yield insights regarding data and control plane design as well as the improvement of prediction techniques.

Related Work

Where contention between multiple management entities exists, conflicts are potential. Research on conflicts in SDN and in traditional network environments shares certain similarity.

Al-Shaer et al. introduced noticeable results in their research on conflicts in security applications, specifically with firewall policies [2, 3], and generalized them to a conflict taxonomy differentiating between various conflict classes for filtering-based network security policies [10]. We refer to two of their conflict class definitions, namely generalization and redundancy, in this paper. While our experiments show conflicts falling within these classes, the side-effects and thus the hidden conflicts that are our focus have not been described in the taxonomy. This is unsurprising, given that the operation mechanism of firewalls in traditional networks is different from that of SDN, which encompasses interaction between applications, the SDN controller and network devices.

Conflicts in SDN have been extensively studied, e.g. [1, 4, 8, 11, 12, 19, 23–25], albeit with focus on contradictions within the rule set in the data plane. Side-effects affecting the interactions seem to be a new, unexplored topic. Pisharody et al. extended the conflict taxonomy mentioned above [10] in SDN by a new conflict class, namely imbrication, which considers conflicts between rules with matching fields representing different OSI layers [21, 22]. They assumed conflict effects corresponding to those stated by Hamed, e.g. for the generalization conflict class, the effect is assessed to be a "warning" since "the specific rule just makes an exception of the general rule" [3]. Conflicts were considered on the basis of rules in the data plane only, which precludes the examination of anomalies originating from side-effects.

Chowdhary et al examined conflicts in the SDN-based Cloud networks [5, 6] and put their focus, similar to other research, on conflicts between rules in the data plane. Zarca et al developed a framework relying on semantic technologies for policy-based security orchestration in SDN/NFV-enabled IoT systems [30], some conflict classes established their similarity to those categorized in the research group of Sloman [14, 15, 18], e.g. conflict of priorities, conflict of duties, multiple managers. Their studies appear also orthogonal to hidden conflicts presented in our work.

Similar to hidden conflicts caused by side-effects, race conditions can lead to unexpected effects in SDN and are also hard to catch. Race conditions have been studied separately in SDN in the control plane [29] and in the data plane [17]. They appear to be a disjoint problem domain to the side-effects examined in this paper, due to the necessary temporal relationships between the participants of a race condition. However, it might be interesting to learn if the combination of concurrency in both control and data planes of SDN might cause side-effect conflicts.

Conclusion

Hidden conflicts are a new conflict type that occurs due to side-effects or unfulfilled expectations of control plane elements. From the starting point of a conflict instance discovered in our experiments, we have presented a systematic analysis of the propensity of SDN interaction primitives to be disrupted so as to expose hidden conflicts. We complemented the analysis with experiments demonstrating the same side-effect cause and effect in the presence of different conflicts from existing taxonomies. This suggests that

the dimension of hidden conflicts is orthogonal to that of hitherto described patterns of conflicts between rules.

We found that hidden conflicts are contingent on the forthcoming data plane traffic and hence, cannot be detected with certainty. In response, we have developed a hidden conflict predictor that speculatively provokes action from control applications to acquire the information necessary for the detection of hidden conflicts. Our current predictor design is application independent but, as a corollary, introduces the risk of changing application state and behaviour in an undesired manner. Hence, in future research we propose to isolate prediction primitives in order to make them available to application developers for use at design time, allowing hidden conflict prediction to be an integral part of applications.

Acknowledgements The authors wish to thank the members of the Munich Network Management Team (www.mnm-team.org), directed by Prof. Dr. Dieter Kranzlmüller, for valuable comments on previous versions of this paper.

Funding Open Access funding provided by Projekt DEAL.

Compliance with Ethical Standards

Conflict of Interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Al-Shaer E, Al-Haj S. FlowChecker: configuration analysis and verification of federated openflow infrastructures. In: Proceedings of the 3rd ACM workshop on assurable and usable security configuration, SafeConfig '10, New York, NY, USA. Association for Computing Machinery; 2010. p. 37–44.
2. Ehab A-S, Hazem H, Raouf B, Masum H. Conflict classification and analysis of distributed firewall policies. IEEE J Select Areas Commun. 2005;23(10):2069–84.
3. Al-Shaer ES, Hamed HH. Firewall policy advisor for anomaly discovery and rule editing. In: Proceedings of the IFIP/IEEE eighth international symposium on integrated network management, Colorado Springs, CO, USA, March 2003. p. 17–30.
4. AuYoung A, Ma Y, Banerjee S, Lee J, Sharma P, Turner Y, Liang C, Mogul JC. Democratic resolution of resource conflicts between SDN control programs. In: Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies, CoNEXT '14, pages 391–402, New York, NY, USA. Association for Computing Machinery 2014.
5. Chowdhary A, Alshamrani A, Huang D. SUPC: SDN enabled universal policy checking in cloud network. In: Proceedings of the 2019 International Conference on Computing, Networking and Communications (ICNC), Feb 2019;pages 572–576.
6. Chowdhary A, Huang D, Ahn G-J, Kang M, Kim A, Velazquez A. SDNSOC: Object oriented SDN framework. In: Proceedings of the ACM International Workshop on Security in Software Defined Networks and Network Function Virtualization, SDN-NFVSec '19, 2019; pages 7–12, New York, NY, USA. Association for Computing Machinery.
7. Danciu V, Guggemos T, Kranzlmüller D. Schichtung virtueller Maschinen zu Labor- und Lehrinfrastruktur. In: 9. DFN-Forum Kommunikationstechnologien, pages 35–44, Bonn, 2016. Gesellschaft für Informatik e.V.
8. Ferguson AD, Guha A, Liang C, Fonseca R, Krishnamurthi S. Hierarchical policies for Software Defined Networks. In: Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN '12, 2012;pages 37–42, New York, NY, USA. Association for Computing Machinery.
9. Paul G, Chuck B. Software defined networks: a comprehensive approach, 1st ed. Morgan Kaufmann; 2014.
10. Hazem H, Ehab A-S. Taxonomy of conflicts in network security policies. IEEE Commun Mag. 2006;44(3):134–41.
11. Kazemian P, Chang M, Zeng H, Varghese G, McKeown N, Whyte S. Real time network policy checking using header space analysis. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, nsdi'13, 2013;pages 99–112, USA. USENIX Association.
12. Khurshid A, Zhou W, Caesar M, Godfrey PB. VeriFlow: Verifying network-wide invariants in real time. In: Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN '12, 2012;pages 49–54, New York, NY, USA. Association for Computing Machinery.
13. Li J, Gu Z, Ren Y, Wu H, Shi SS. A software-defined address resolution proxy. In: Proceedings of the 2017 IEEE Symposium on Computers and Communications (ISCC), July 2017; pages 404–410
14. Lupu E, Sloman M. Conflict analysis for management policies. In: Proceedings of the International Symposium on Integrated Network Management, 1997; pages 430–443. Springer.
15. Lupu EC, Sloman M. Conflicts in policy-based distributed systems management. IEEE Trans Softw Eng. 1999;25(6):852–69.
16. McKeown N, Anderson T, Balakrishnan H, Parulkar G, Peterson L, Rexford J, Shenker S, Turner J. OpenFlow: enabling innovation in campus networks. SIGCOMM Comput Commun Rev. 2008;38(2):69–74.
17. Miserez J, Bielik P, El-Hassany A, Vanbever L, Vechev M. SDNRacer: Detecting concurrency violations in Software-Defined Networks. In: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, New York, NY, USA, 2015. Association for Computing Machinery.
18. Moffett Jonathan D, Sloman MS. Policy conflict analysis in distributed system management. J Organ Comput Electron Com. 1994;4(1):1–22.
19. Mogul JC, AuYoung A, Banerjee S, Popa L, Lee J, Mudigonda J, Sharma P, Turner YC: Towards the modular composition of SDN control programs. In: Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks, HotNets-XII, New York, NY, USA, 2013. Association for Computing Machinery.

20. Pfaff B, Pettit J, Koponen T, Jackson EJ, Zhou A, Rajahalme J, Gross J, Wang A, Stringer J, Shelar P, Amidon K, Casado M. The design and implementation of Open vSwitch. In: Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, NSDI'15, pages 117–130, USA, 2015. USENIX Association.
21. Pisharody S. Policy Conflict Management in Distributed SDN Environments. PhD thesis, Arizona State University, 2017.
22. Sandeep Pisharody, Janakarajan Natarajan, Ankur Chowdhary, Abdullah Alshalan, Dijiang Huang. Brew: a security policy analysis framework for distributed sdn-based cloud environments. IEEE Trans Depend Secure Comput. 2019;16(6):1011–25.
23. 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, HotSDN '12, pages 121–126, New York, NY, USA, 2012. Association for Computing Machinery.
24. Shin S, Porras PA, Yegneswaran V, Fong MW, Gu G, Tyson M. FRESCO: Modular composable security services for Software-Defined Networks. In: Proceedings of the 20th Annual Network & Distributed System Security (NDSS) Symposium, San Diego, CA United States, 2013.
25. Peng S, Ratul M, Jennifer R, Lihua Y, Ming Zhang, Ahsan Arefin. A network-state management service. ACM SIGCOMM Comput Commun Rev. 2015;44(4):563–74.
26. Tran CN, Danciu V. On conflict handling in software-defined networks. In: Proceedings of the 2018 International Conference on Advanced Computing and Applications (ACOMP), pages 50–57. CPS, 2018.
27. Tran CN, Danciu V. A general approach to conflict detection in software-defined networks. SN Comput Sci. 2019;1(1):9.
28. Tran CN, Danciu V. Hidden conflicts in software-defined networks. In Proceedings of the 2019 International Conference on Advanced Computing and Applications (ACOMP), pages 127–134. IEEE, 2019.
29. Xu L, Huang J, Hong S, Zhang J, Gu G. Attacking the brain: Races in the SDN control plane. In: Proceedings of the 26th USENIX Conference on Security Symposium, SEC'17, pages 451–468, USA, 2017. USENIX Association.
30. Molina ZA, Miloud B, Bernal BJ, Tarik T, Skarmeta AF. Semantic-aware security orchestration in SDN/NFV-enabled IoT systems. Sensors. 2020;20(13):3622.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.