Fault Management in Software-Defined Networking: A Survey

Yinbo Yu, *Student Member, IEEE*, Xing Li, Xue Leng, *Student Member, IEEE*, Libin Song, Kai Bu, *Member, IEEE*, Yan Chen, *Fellow, IEEE*, Jianfeng Yang, *Member, IEEE*, Liang Zhang, Kang Cheng, and Xin Xiao

***Abstract*—Software-defined networking (SDN) has emerged as a new network paradigm that promises control/data plane separation and centralized network control. While these features simplify network management and enable innovative networking, they give rise to persistent concerns about *reliability*. The new paradigm suffers from the disadvantage that various network faults may consistently undermine the reliability of such a network, and such faults are often new and difficult to resolve with existing solutions. To ensure SDN reliability, *fault management*, which is concerned with detecting, localizing, correcting and preventing faults, has become a key component in SDN networks. Although many SDN fault management solutions have been proposed, we find that they often resolve SDN faults from an incomplete perspective which may result in side effects. More critically, as the SDN paradigm evolves, additional fault types are being exposed. Therefore, comprehensive reviews and constant improvements are required to remain on the leading edge of SDN fault management. In this paper, we present the first comprehensive and systematic survey of SDN faults and related management solutions identified through advancements in both the research community and industry. We apply a systematic classification of SDN faults, compare and analyze existing SDN fault management solutions in the literature, and conduct a gap analysis between solutions developed in an academic research context and practical deployments. The current challenges and emerging trends are also noted as potential future research directions. This paper aims to provide academic researchers and industrial engineers with a comprehensive survey with the hope of advancing SDN and inspiring new solutions.**

***Index Terms*—Software-defined networking (SDN), SDN reliability, SDN faults, fault classification, system monitoring, fault diagnosis, fault recovery and repair, fault tolerance.**

Manuscript received October 9, 2017; revised March 22, 2018 and July 1, 2018; accepted September 1, 2018. Date of publication September 6, 2018; date of current version February 22, 2019. This work was supported in part by the National Key Research and Development Program of China under Grant 2016YFC0106301, in part by the Huawei HARP under Grant HO2016050002CH, and in part by the Provincial Science and Technology Pillar Program of Hubei under Grant 2017AAA027, Grant 2017AAA042, and Grant 2017AHB048. *(Corresponding author: Jianfeng Yang.)*

Y. Yu and J. Yang are with the School of Electronic Information, Wuhan University, Wuhan 430072, China (e-mail: yyb@whu.edu.cn; yjf@whu.edu.cn).

X. Li, X. Leng, and K. Bu are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China (e-mail: xing_li@zju.edu.cn; lengxue_2015@outlook.com; kaibu@zju.edu.cn).

L. Song and Y. Chen are with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208 USA (e-mail: libinsong2020@u.northwestern.edu; ychen@northwestern.edu).

L. Zhang, K. Cheng, and X. Xiao are with the Nanjing Research and Development Center, Huawei Technologies Company Ltd., Nanjing 210012, China (e-mail: zhangliang1@huawei.com; chengkang@huawei.com; xinxiao@huawei.com).

Digital Object Identifier 10.1109/COMST.2018.2868922

I. Introduction

SOFTWARE-DEFINED networking (SDN) is an emerging network paradigm that promises to simplify network management and enable innovations in networking [1]–[3]. In SDN, the traditional network architecture is split into a programmable data plane and a logically centralized control plane, rather than the two being integrated in the same configurable black box [4]–[6]. The split architecture places most of the network control logic (specified by software programming) into the control plane and simplifies the data plane, which merely acts on forwarding decisions installed by the control plane [4], [5]. SDN reduces the complexity of network management and provides powerful programmability for networking. A network implemented with SDN can quickly evolve to satisfy network users' rapidly changing demands for network resources, e.g., in cloud computing [7], network function virtualization (NFV) [8] and Internet of Things (IoT) [9] scenarios. The network innovations it provides position SDN as the future of networking.

SDN mainly originated from the OpenFlow project created by McKeown *et al.* [6]. Until now, SDN has undergone constant development and has been the subject of significant attention and active exploration in academia and industry. In addition to being the focus of a large number of research publications [1]–[3], SDN has also achieved many successful deployment stories presented by IT corporations, such as the Google B4 project [10], Microsoft Ananta [11], and NTT Cloud gateway [12]. Currently, the development of SDN is being strongly promoted by various organizations (e.g., industries, enterprises, data center vendors, governments and academic institutions) [13], and SDN-based solutions are also advancing the development of many other emerging network technologies (e.g., NFV, IoT, cloud, and 5G technologies) [1], [7]–[9]. SDN is therefore highly promising for modern network management. However, regarding the adoption of SDN techniques, one area that continues to cause concern for network operators and users is the *reliability* of SDN [1], [6], [13].

Reliability refers to the probability of failure-free operation over a specified period of time under stated conditions [14].

Reliability is a critical ingredient of all designs created in all industries, and SDN is no exception. Certainly, the centralized nature of networking in SDN offers clear advantages in terms of reliability. For instance, because of the global network visibility, the control plane can easily compose different network policies and materialize them on the data plane without conflicts. However, several new features in the SDN architecture also raise doubts about its reliability. Such features include the *control/data plane separation architecture*, which can increase network processing latency and lead to cross-layer network faults [15]; the *software-based management* approach, which can introduce various software bugs/defects into networks [16], [17]; and the *limited network processing capability of SDN controllers*, which can affect the reliability of network control for networks of diverse sizes [17]. These features all have the potential to cause faults in SDN that exhibit various symptoms and have complex root causes, thus making them more complex and difficult to avoid and diagnose. For example, state inconsistency between the control and data planes is a common type of fault in SDN and can induce various types of network failures, e.g., forwarding loops and blackholes; moreover, a fault of this type has several possible root causes, including vendor-specific optimization of switches [15], [18], software bugs in the control plane [16], [17], or conflicts among different policies [19], [20]. Thus, the ability to resolve faults to achieve high reliability remains a key concern when implementing SDN.

To control the effects of these faults, several techniques are used [21], [22], such as system state monitoring, fault detection, localization and resolution, and fault tolerance mechanisms. These techniques are collectively referred to as *fault management* techniques [21]. In the domain of SDN, multiple recent investigations have been conducted on fault management solutions, including software fault troubleshooting [16], [17], [23], policy conflict arbitration [24]–[26], forwarding path verification [19], [20], [27], network behavior inspection [15], [28], network measurement [29], [30], as well as fault recovery and tolerance design [31], [32]. These studies have greatly contributed to improving the reliability of SDN. However, we find that most such studies resolve SDN faults from only a partial perspective, not a global one; this may result in incomplete and flawed solutions and may even induce other side effects. More seriously, as the network paradigm evolves, more potential faults are being exposed. Thus, it is necessary to conduct a comprehensive and systematic survey of SDN faults and related management solutions, accompanied by an in-depth discussion and analysis, to provide researchers and engineers with a foundation for motivating continual improvements in SDN fault management.

However, to the best of our knowledge, such comprehensive survey of SDN fault management has yet been performed [1]–[3], [33]–[38]. Some previous surveys have been conducted from the overall perspective of SDN development [1]–[3], some have focused on issues in the SDN security domain [33]–[35], [39], and some have focused on SDN network measurement [36] and SDN fault tolerance [37]. SDN faults and management solutions have been discussed only to a lesser extent, without detailed discussions or analysis. This situation motivates us to systematically summarize and evaluate existing solutions for SDN fault management to achieve such a fundamental and comprehensive survey.

The main objective of this paper is to survey the academic publications and industrial projects related to SDN fault management over the period of 2008–2017 and to present a systematic discussion and analysis of SDN faults and management solutions. The main contributions of this paper are as follows:

- *State of SDN:* We characterize the current overall state of SDN development to help new researchers and network operators understand SDN and the related reliability issues (Section II-C).
- *Taxonomic Framework:* We design a two-dimensional taxonomic framework to provide an overview of SDN faults and related management solutions (Section II-E).
- *Fault Classification:* Through a bottom-up analysis of the SDN architecture (Section II-B), we develop a systematic classification of faults in SDN networks and analyze their symptoms and root causes (Section III).
- *Evaluation and Analysis of Existing SDN Fault Management Solutions:* We present an in-depth analysis of SDN fault management with respect to **system monitoring** (Section IV), **fault diagnosis** (Section V), **fault recovery and repair** (Section VI) and **fault tolerance** (Section VII) by classifying, comparing and analyzing existing solutions.
- *Gap Analysis Between Academia and Industry:* Considering the gap between academic research and industrial engineering, we also highlight fault management projects implemented in mainstream SDN-related ecosystems and identify the major barriers to developing powerful fault management tools for practical SDN network management (Section VIII).
- *Future Research Opportunities:* Finally, we present a discussion of potential research directions related to SDN fault management, including current challenges and emerging trends (Section X).

II. Background

In this section, we present the background on SDN, focusing on its differences from traditional networks, relevant terminology, and the current state of development. The fundamentals of SDN fault management and a two-dimensional taxonomic framework for classifying SDN faults and related management solutions are also presented.

A. Traditional and SDN Networks

As shown in Fig. 1 (a), a traditional network consists of various configurable network devices and is commonly operated in a distributed manner. Each of these devices is an autonomous system that can build its own forwarding information bases (FIBs) and network topology by exchanging network information with its neighbors and can then decide how to forward packets based on its FIBs. In these network devices, the control logic is tightly coupled with forwarding functions, and this

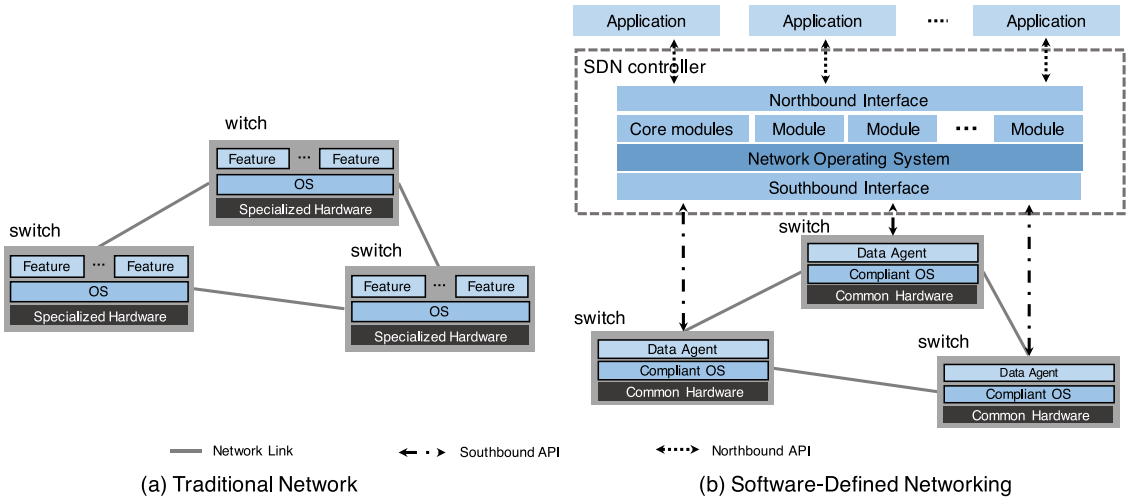


Fig. 1. Architectures of a traditional network and an SDN network.

tight coupling, however, leads to a variety of problems with regard to, e.g., the compatibility and extension of network protocols, switch software updates, network device maintenance, and network innovations.

In SDN (see Fig. 1 (b)), the network architecture is modified by clearly separating the network into three network planes (i.e., the *application plane*, *control plane*, and *data plane*) connected by two internal interfaces (i.e., the *northbound interface* and *southbound interface*). The application plane allows network operators to specify their desired network control logic through the northbound interface. The control plane is the core of an SDN network. It abstracts the network state and resource information to simplify the network reasoning for decision making in the application plane; meanwhile, it also translates the control logic into low-level flow instructions and installs them into the data plane through the southbound interface (as defined by SDN protocols, e.g., OpenFlow [40] and ForCES [41]) to control traffic forwarding. The data plane then simply forwards packets according to the installed flow instructions. These features of SDN offer a promising means of addressing the problems in the traditional network architecture and have driven SDN to gain significant traction in both academia and industry.

B. SDN Terminology

To further introduce SDN, we now discuss the major elements of the SDN architecture. Currently, several standard organizations (e.g., the Open Networking Foundation (ONF),[1] the Internet Engineering Task Force (IETF) [41] and the Internet Research Task Force (IRTF) [5]) and industrial and community consortia (e.g., OpenDaylight)[2] and ONOS[3] all conduct standardization activities for SDN. In this subsection, we present the essential SDN terminology used throughout this paper, as collected from documents regarding these standardization activities and academic literature surveys [1]–[3], [33], [34]. We provide a list of abbreviations in Table I. The detailed descriptions of the relevant terminology are as follows:

- *Network Device:* A network device can be either physical or virtual and performs a set of network operations relevant to packet forwarding. It can be implemented in common hardware (e.g., NetFPGA and Pica8 3920) with a compliant operating system or in software (e.g., Open vSwitch) on common servers.
- *Data Agent:* A network device contains one or more *data agents*, as defined by SDN southbound protocols (e.g., OpenFlow, ForCES or Programming Protocol-Independent Packet Processors (P4) [42]), for forwarding packets and interacting with SDN controllers; examples of such data agents include OpenFlow agents and ForCES Forwarding Elements.
- *Data Plane (DP):* The DP is the bottom layer of an SDN network and is the collection of network devices.
- *Southbound Interface (SBI):* Located between the control plane and the DP, the SBI is responsible for all interactions between these two planes, e.g., configuration issuing, event notification, and device performance querying. These interactions are often defined by SDN southbound protocols.
- *Controller:* A controller is a software entity for network control. In addition to providing core management functions, e.g., topology discovery, device management, and state synchronization, it also enables the installation of external programs (i.e., applications) through northbound APIs or controller APIs (i.e., controller function interfaces) [43], allowing them to deploy their control logic into the underlying networks.
- *Control Plane (CP):* As the middle layer, the CP is the collection of all controllers and acts as a network operating system. The CP is logically centralized, but it can be implemented either in a physically distributed manner or in a cluster to manage all network devices in the DP.
- *Eastbound/Westbound Interface (EBI/WBI):* Since the CP is often physically distributed, it is necessary

[1]ONF - https://www.opennetworking.org/.
[2]OpenDaylight - https://www.opendaylight.org/.
[3]ONOS - https://onosproject.org/.

TABLE I
ABBREVIATION

Acronym	Description
DP	Data plane
SBI	Southbound interface
CP	Control plane
NBI	Northbound interface
EBI/WBI	Eastbound/westbound interface
AP	Application plane
App	SDN application

to implement EBIs and WBIs (e.g., SDNi [44]) to enable interactions among the distributed controllers (e.g., state synchronization, control coordination and topology exchange).

- *Northbound Interface (NBI):* The NBI is the communication interface between the application plane and the CP. It is responsible for providing upper-level SDN applications with an abstract view of the underlying network and interfaces for accessing network resources. Unlike in the case of the SBI, currently, the controllers often use a common interface (e.g., REST API, Onix API or Java API) to implement the NBI.
- *Application Plane (AP):* The AP is the top layer and includes various applications and external affairs (e.g., high-level orchestration systems). Network users can interact with this plane to control and manage the entire network through a variety of applications.
- *Application (App):* In the context of SDN, apps are the software programs that define the network control logic. Apps can be implemented in the AP (accessing the network through the NBI) or CP (accessing the network through controller APIs). Apps that are implemented in controllers are more commonly called *control programs* or *application modules* [23], [45], [46] ; examples include the application *agents* in SDN controllers defined by ONF [4] and the *plugins* in OpenDaylight. Some of them can provide northbound APIs for external apps. In this paper, we do not draw explicit distinctions among the terms "application", "control program" and "application module"; instead, we consider them interchangeably to help us focus on the faults induced by design flaws and coding mistakes.

C. State of SDN Development

In this subsection, we characterize the state of SDN development, including open networking ecosystems, network programmability, southbound protocols, and network complexity, to help readers understand the reliability issues facing SDN.

1) Open Networking Ecosystems: In the traditional networking world, each network company (e.g., Cisco, Brocade and Huawei) typically dominated its own network ecosystem, with these ecosystems being either closed or semi-closed and interoperable with others only through IEEE or IETF standards. These closed ecosystems hinder the rapid development of network technologies. By contrast, one of the main promises of the SDN era is to provide an open networking ecosystem, which can offer various benefits, e.g., deep network service sharing, cross-product integration, multi-vendor interoperability among products and support for open-source software. For example, SDN app stores[4] have emerged to provide a centralized SDN app management platform for facilitating new app submission and app acquistion. The implementation of open networking ecosystems can reduce networking market monopolies and promote the rapid development of network technologies.

2) Network Programmability: One of the advantages of SDN is *network programmability*, whereby an SDN controller can provide NBIs to allow SDN apps to access network resources and control network behaviors based on their own needs through underlying programmable protocols (e.g., OpenFlow). Through these programmable interfaces, each network operator can write his own programs to inject his desired network policies into the network. Thus, SDN apps act as the "network brains" for various network services, e.g., traffic engineering, mobility, measurement, data center networking, and security [1].

Although network programmability can simplify network management, verifying the correctness of SDN apps is difficult since they often co-occur with complex and varying network states [23], [46]. Additionally, northbound APIs are often defined at the abstraction of the network provided by the southbound protocols that enable this programmability, such as OpenFlow, which operates at a very low level. Thus, apps must perform reasoning on network states based on numerous flow rules to solve problems. To reduce the complexity of network programming, many solutions currently proceed from low-level flow languages to high-level abstract languages [47], such as Frenetic [24], Pyretic [48] and PGA [49]. These high-level programming languages can simplify and even remove numerous processes in flow rule orchestration (e.g., through the elimination of overlapping rules and priority ordering) and thus provide a high-level and efficient abstraction layer for apps. This advancement in network programmability has further enhanced the development of SDN.

3) Southbound Protocol: SDN is often linked to the OpenFlow protocol, which emerged from an academic experiment in 2008 [6]. Over the past few years, OpenFlow has become the predominant SDN southbound protocol and has directly affected the development and implementation of the SDN architecture. An OpenFlow switch contains an OpenFlow agent and a datapath built on common hardware [40]. The OpenFlow agent is responsible for all operations involving OpenFlow messages, such as generating and sending *Packet_In* messages for newly arriving packets, receiving and installing OpenFlow rules from the controller, and generating barrier messages for rule[5] installation. The datapath is a pipeline of the flow tables, the group table and the forwarding ports, with which the switch can perform match-action processing for incoming packets. The proposal of OpenFlow has resulted in simpler and more open switches and has enabled network operators to dive deeper into traffic forwarding since

[4]Examples include the CoreStack SDN App Store (https://www.cloudenablers.com/corestack-sdn-app-store.php) and HPE SDN App Store (https://marketplace.saas.hpe.com/sdn).

[5]We use the term "rule" to refer to an OpenFlow flow rule which is also called an OpenFlow flow entry.

they can send flow-level instructions to switches for packet manipulation.

In addition to OpenFlow, ForCES is another protocol that has been attracting significant attention [50]. Its goal is also the separation of the CP and DP; in ForCES, this goal is achieved through the definition of a set of protocols (e.g., routing protocols and signaling protocols) and a model [51] that is necessary to separate these planes. The model is a modeling language (ForCES model) that allows developers to define their own abstraction models to control traffic activities in the DP [50]. Moreover, CP/DP separation is also achieved in many other protocols, such as OVSDB [52], NETCONF [53], and Protocol-Oblivious Forwarding (POF) [54]. These protocols also provide flexible APIs for the control of packet forwarding or configurations (we refer the reader to [1] for detailed discussions). However, although these southbound SDN protocols are powerful and vendor agnostic for network programmability, they provide open programmability only in the CP; the *packet parsing* and *header field matching* in the DP still depend on specific network protocols, e.g., VLAN and NvGRE [42]. This is also why these specifications need to ensure passive evolution to support more network requirements; however, this capability often suffers from a tedious development cycle for updating switch software and handling backward compatibility issues [42], [55], [56].

Fortunately, some efforts have been made to address this issue, in protocols such as POF and P4. POF was proposed to enable a protocol-obvious SBI that can remove any protocol dependency using a low-level forwarding instruction set (FIS). By defining instructions in the FIS, POF simplifies the process of packet header parsing to a controller task. P4 is similar to POF and has the same goal of extending SDN programmability. P4 is a high-level language for configuring switches and enables network operators to specify both packet parsing and processing in the DP. These protocols further improve the programmability of SDN for both forwarding logic and packet parsing and reduce the complexity of switch development.

4) Network Control Capability: The DP provided by current SDN southbound protocols is powerful enough to satisfy various layer 2/3 (L2/L3) network demands. However, in modern networks [39], [57], many more complex middleboxes or network functions (NFs) (e.g., load balancing, deep packet inspection, and firewalls) are combined with L2/L3 forwarding devices (e.g., switches and routers) to offer highly specialized network services and more powerful networking capabilities. The packet processing in many NFs is often stateful (also called context-dependent) in nature and is based on historical packet information [39], e.g., packet connection contexts, local link states, and port traffic loads.

Unfortunately, due to the switch-to-controller signaling load and processing latency induced by the split architecture and centralized control, the SDN paradigm faces difficulties in managing such stateful networks [58]. In addition, flow-rule-based network control can only deal with basic L2/L3 NFs, which may cause SDN to be able to obtain only limited visibility of the entire network [58]. Recently, many solutions [58]–[62] have been proposed to overcome this issue. These solutions extend the DP implementations to offload some stateful traffic processing and control tasks to be handled directly within switches, such as by adding a state table for stateful traffic processing [58]–[60] or modifying the NFs to insert tagging policies for steering traffic via flow rules in switches [61], [62]. Nevertheless, to satisfy additional network demands, the further extension of the network control capability of SDN still requires greater attention.

D. Fundamentals of SDN Fault Management

In this subsection, we describe the fundamentals of SDN fault management. For a network, a *failure* is the inability of the network or some component thereof to perform required functions; an *error* is a mistake made based on human actions or other factors that produces an erroneous result; and a *fault*, or more commonly a "bug", is a manifestation of an error in the form of an incorrect condition or defect that can cause the network to behave in an unintended manner. Thus, the result of an error is a fault, and a fault can lead to a failure. For example, a network operator may modify the OpenFlow rules in some OpenFlow switches without notifying the controller, which can lead to an inconsistent rule state between these switches and the controller. This fault may then lead to a network failure (e.g., a forwarding loop) as the network updates.

The occurrence of faults, errors, and failures is the most common and direct avenue through which the reliability of SDN is undermined. Fault management is the process of detecting, localizing, resolving and preventing faults [21], [22]. Thus, the design of suitable fault management solutions is indispensable for achieving reliable SDN deployment. Based on the taxonomy of fault management techniques for distributed systems [21], we divide the SDN fault management process into four tasks, each of which makes distinct contributions to SDN reliability. Our survey is conducted based on the following four tasks:

- *System Monitoring:* to monitor and trace system behaviors and collect statistical data with different granularities according to specified monitoring metrics;
- *Fault Diagnosis:* to detect possible faults and localize their root causes from collected data;
- *Fault Recovery and Repair:* to reconfigure or reconstruct a system or its components after faults have occurred;
- *Fault Tolerance:* to prevent faults that have occurred in some components from affecting other components or the entire system or to reduce the damage they cause such that proper operation will continue.

E. Taxonomic Framework

For clarity in presenting our fault management survey, we define a two-dimensional taxonomic framework (shown in Fig. 2) as an overview of the survey structure. This framework is specifically designed for categorizing SDN faults and related fault management solutions. We list the fault classification and the four fault management tasks in the horizontal dimension, and we group corresponding solutions to these four tasks into subtasks in the vertical dimension according to their basic

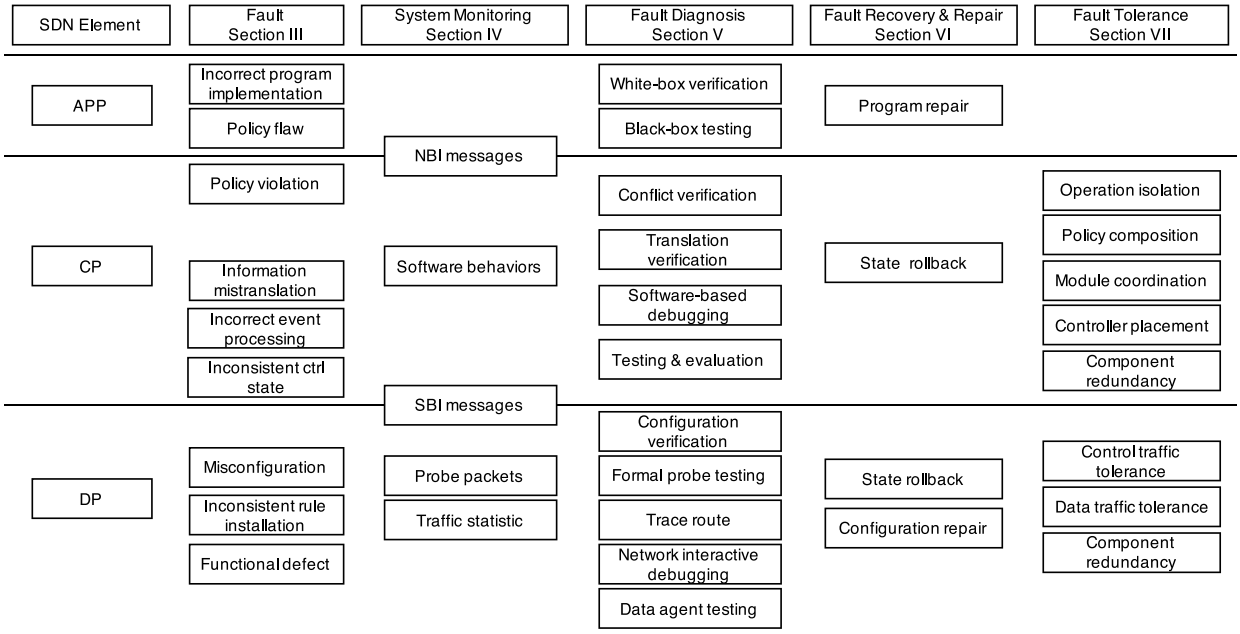


Fig. 2. A two-dimensional taxonomy of SDN fault management solutions in the academic literature. These solutions are classified into four tasks (i.e., system measurement, fault diagnosis, fault recovery and fault tolerance), which are presented along with the fault classification on the horizontal axis, and a bottom-up approach for discussing each task for SDN fault management is represented on the vertical axis. Note that the structure of our paper is based on this taxonomy.

TABLE II
SDN FAULT CLASSIFICATION

SDN Element	Fault	Short Description	Manifest Symptoms	Root Causes
Data Plane	Network misconfiguration	Incorrect configurations in the DP.	Network failures.[1]	Flaws in the control logic and bugs in switch hardware and software.
	Inconsistent rule installation	Rules in the forwarding table are not consistent with the controller's view.	Inconsistent network states.	Inconsistent switches, coding mistakes, hardware errors and external operations.
	Functional defect	Missing features and or poor compatibility or interoperability.	Missing error messages, easily triggered undesired shutdown, message reordering and invalid requests etc.	Inconsistent and/or incorrect implementations of OpenFlow specifications.
Control Plane	Policy violation	Installed policies violate the network invariants or operators' specifications.	Network failures.	Policy flaws, policy conflicts, policy mistranslations and coding mistakes in the AP/CP.
	Policy conflict	Policies from different apps conflict with each other.	Inconsistent network states and component crashes.	Independent development.
	Information mistranslation	Incorrect translation of policies into rules or abstract network states.	Inconsistent network states.	Logic errors and coding mistakes.
	Inconsistent controller states	Different controllers in a cluster execute different actions when processing the same event.	Inconsistent controller states and actions in response to events.	Design and logic flaws and software bugs.
	Incorrect event processing	External/internal events cannot be processed properly.	Abnormal warnings or errors, service delays or interruptions, abnormal network states and invalid access.	Design and logic flaws and coding mistakes.
Application	Incorrect program implementation	Apps process network events incorrectly.	Silent discarding of messages, abnormal event responses and unexpected network states.	Coding mistakes and logic flaws.
	Policy flaw	Policies in apps violate the network properties or operators' specifications.	Violations of invariants and unexpected network states.	Coding mistakes, insufficient system knowledge and misplaced assumptions.

[1] Detailed classification is presented in Table III.

methodology. Note that in the vertical dimension, we do not follow the order of the layers in the SDN architecture, namely, the DP, CP and AP; we prefer to classify faults and their fault management techniques at the DP, CP and app levels. This is because the app level represents both apps in the AP and control programs in the CP, which can help us to distinguish faults due to coding mistakes in apps from those due to system flaws in the CP and thus to describe SDN reliability issues more clearly.

III. FAULT ANALYSIS AND CLASSIFICATION

Given the background on SDN and fault management, we now discuss the main faults in the SDN stack, which manifest with different symptoms and have different root causes. In Table II, we summarize all analyzed faults in SDN with a bottom-up classification, since abnormal network behaviors commonly occur in the DP, and we present concise descriptions of these faults along with their symptoms and root causes.

TABLE III
NETWORK FAILURES

Category	Short Description	Symptom	Definition
Forwarding	Abnormal behaviors in packet forwarding.	Reachability failure	Given a starting location and a flow, packets of this flow do not ultimately reach their destination [27], [28], [63].
		Forwarding loop	Given a flow, its forwarding path forms a loop such that packets of this flow will ultimately be forwarded to their originating hosts [27], [28], [64].
		Blackhole	Given a forwarding path, the packets traversing this path are absorbed and dropped by some abnormal nodes (e.g., switches and routers) along this path [20], [27].
		Waypoint routing	Given two nodes in a network, the traffic between them always passes through a "waypoint" node [19], [64].
		Incorrect slice isolation	Given two traffic groups (e.g., VLAN-based subnetworks), the actual network behaviors between them violate operators' desired isolation properties [19], [20].
		Long path length	The length of the actual forwarding path of a given flow is greater than expected [19].
Transformation	Incorrect packet transformation.	Incorrect packet changes	Some of the bits of a packet's header or payload are rewritten incorrectly [65], [66].
		Packet tunneling errors	A packet is encapsulated or de-encapsulated into a new packet with an incorrect header for tunneling (e.g., IP-in-IP or MPLS) [66], [67].
		Incorrect packet splitting/coalescing	Something is done incorrectly in traffic processing across multiple packets, e.g., TCP segment coalescing/splitting, IP fragmentation or payload combination [67].
Dynamic	Dynamically or randomly abnormal behaviors.	Link congestion	The utilization of links exceeds a certain threshold, resulting in queuing delays, jitter, connection jams and packet loss [28].
		Unexpected traffic loss	Packets are dropped at an unexpected destination or over a random period due to switch failures, link congestion and attacks [15], [28].
		Intermittent connectivity	Some network nodes drop users' connections at random intervals [28].
		Link/packet delay	The time that packets take to traverse from their sources to their destinations is longer than usual [28].
		Load imbalance	Given a cluster, the traffic load on each host is unbalanced [19], [68].

A. Data Plane Faults

As discussed in Section II-A, SDN southbound protocols can provide a simple network device with a data agent and a datapath for processing and forwarding packets, such as the OpenFlow agent and match-action pipeline in an OpenFlow switch. Faults in these two components or in the device hardware can potentially induce abnormal network behaviors, i.e., *network failures*. These network failures are deviations from typical network properties and are common and frequent in computing networks. According to their characteristics with respect to packet processing, they can be divided into three categories: *forwarding issues*, *transformation issues* and *dynamic issues*. Forwarding issues are abnormal behaviors in packet forwarding, such as reachability failure, forwarding loops, waypoint routing, host isolation, blackholes, and long path lengths. Transformation issues involve incorrect packet transformation, such as incorrect packet header modification, packet encapsulation or de-encapsulation errors, and incorrect packet replication. Dynamic issues are abnormal behaviors that occur dynamically or randomly, such as unexpected packet loss, link congestion, link latency, load imbalance, and intermittent connectivity. We summarize all these network failures in Table III. In this subsection, we discuss three main types of faults in the DP, all of which can induce such network failures.

1) Network Misconfiguration: The main cause of network property deviations is incorrect network configurations, i.e., *network misconfiguration*. Network misconfiguration can induce various unexpected network states such as reachability failures and forwarding loops [19], [20]. These configuration issues are the most common problems in computing networks, but they are thorny to resolve because there may be a large number of configuration files to check. SDN decouples the CP and DP and uses a centralized controller to configure the DP. Although this split architecture can simplify network configuration, it also makes configuration issues more complex since their root causes may be cross-layer in nature. For example, a missing rule may be caused by policy conflicts in the CP [25], [69], software defects in switches [15], [70], attacks [71], [72], or even careless external rule modification [73]. Thus, we need to consider these challenges and resolve such faults through more cross-layer analysis. Note that the details of issues concerning the CP that can induce network misconfiguration are discussed in Section III-B, whereas other faults in the DP are discussed below.

2) Inconsistent Rule Installation: At first glance, the causes of network failures appear to be *network misconfiguration*. However, even when the configurations generated by the CP are correct, the actual networks implemented by flow rules in the DP may still violate network operators' requirements [18], [28], [70]. We call this issue of inconsistent rules in the DP *inconsistent rule installation*, which means that the actual rules installed in a switch are not consistent with the designated configurations generated by the controllers. Inconsistent installed rules may not necessarily lead to network failures; rather, they can directly cause undesirable forwarding behaviors and undermine the controller's network visibility in the DP, which can then lead to incorrect policy actions. More seriously, configuration correctness checks relying on the CP view [19], [20] may lose their effectiveness.

For clarity, we classify inconsistent rule installation faults into two types: *rule loss* and *rule reordering* [18], [28], [70]. Rule loss refers to a fault whereby some flow entries in flow

tables are lost without the controllers being notified [18], [28]. Rule reordering occurs when the actual order of the flow entries in the flow table does not follow the designated order and the priorities of some rules overlap with others [18], [70], [73]. The root causes of this type of fault vary from faulty internal software implementations in the switch to external misoperation. We summarize as follows:

- *Inconsistent data agents:* The implementations of SDN southbound protocols in different switches may be inconsistent with each other and thus may deal with configuration messages generated by controllers in different ways [18]. For example, the given order of the rules generated by a controller may be arbitrarily reordered in a switch to maximize its performance [18], [74].
- *Functional defects:* As a software entity, a data agent will inevitable be wrong at times. For example, software defects in an OpenFlow agent can directly impact rule installation, causing rules to be dropped or even modified [28]. Note that in addition to the ability of functional defects to impact rule installation, more issues concerning data agents will be identified in Section III-A3.
- *Hardware faults:* This is a typical and intractable problem in network hardware infrastructures. Hardware faults can be divided into soft errors (e.g., a bit flip in memory) and hard errors (e.g., a bit that is always bad). Whereas soft errors are temporary and can be eliminated or repaired by rewriting, hard errors are permanent and cannot be repaired. Soft errors can lead to rule installation failures and rule tampering at run time [15]. Hard errors in memory can also lead to rule loss and even some feature crashes. This means that some hardware components or all of the hardware may need to be replaced.
- *External operations:* The forwarding tables in a switch can be modified not only by SDN controllers but also by external operations (e.g., attacks or manual configuration activities) that may not be noticed by the controller [72], [73]. By hijacking the controller or accessing the control channel, an attacker can insert or modify flow rules to redirect traffic elsewhere [33], [34], [71], [72]. In addition, manual rule modification by means of the configuration tools in switches (e.g., `ovs-ofctl` in Open vSwitch) can also potentially induce rule inconsistencies since it is necessary to carefully inspect the existing rules to avoid unexpected overriding any of them [73].

To solve rule inconsistency issues, one feasible approach is to design positive acknowledgments for per-rule installation. Unfortunately, the OpenFlow specification does not provide such a mechanism since doing so would increase its complexity [40], [75]. However, OpenFlow does provide barrier messages for the controller that serve similar functions, e.g., ensuring message ordering and notification of completed operations [40]. When a switch receives a barrier message, it must finish executing all previously received commands. This is a high-level negative acknowledgment mechanism and thus is inefficient and error-prone for verifying the success of rule installation. Since OpenFlow does not provide lost-message detection and recovery mechanisms [40], any lost barrier reply message may lead to failures in control-data state consistency [75]. More seriously, as discussed in [18], [76], and [77], these barriers may not always be implemented in switches since the delay imposed by confirming rule installation may cause prolonged packet loss. It is safe to conclude that the occurrence of inconsistent rule installation in the DP is quite possible, and therefore, we must pay greater attention to this issue.

3) Functional Defects: When implementing network devices, device manufacturers often follow standard network protocols or specific network requirements to develop their software, i.e., *data agents*. Software-based data agents may contain defects that can cause various network failures such as violations of protocol compliance and missing features. We refer to a fault of this type in the DP as a *functional defect* in this paper. We use an OpenFlow agent as an example to discuss such faults, considering two root causes, and we then extend the discussion to the more complex DP.

The first cause of functional defects is *incorrectness and incompleteness* of the OpenFlow specification implementation. Defects in OpenFlow agents are inevitable and can affect normal network operations following the OpenFlow specification. This issue will be aggravated as OpenFlow evolves since many new features will need to be implemented in agents to support more network management functions. As tested in [55], several defects have been found in OpenFlow agents; such defects include incorrect message dropping, missing features, missing error messages, OpenFlow agent termination with an error, different orders of message validation, and statistic requests silently being ignored. These defects can directly cause inconsistent rule installation by blocking the installation of a rule or arbitrarily changing some fields in a rule or the order of rules. In addition, they may break the correctness of the operational state between the CP and DP by causing barrier messages to be dropped or incorrect barrier messages to be sent to the controller due to incorrect barrier implementation [55], which can, in turn, induce incorrect policy actions and exacerbate inconsistent rule installation.

The other cause is the poor *compatibility and interoperability* among different OpenFlow switches. Currently, the OpenFlow specification has evolved to version 1.5.1, which can support more functions than previous versions could [40]. Specification version growth is generally beneficial; however, the long debugging cycle for switch software updates makes it difficult for switches to keep up with the pace of specification growth. Switch manufacturers need to invest more manpower and material resources in upgrading to new OpenFlow versions, and they may experience an imbalance between costs and benefits. Currently, most switches only support OpenFlow 1.0 or 1.3 and few can support more versions [78]. The resulting problem of poor compatibility and interoperability not only limits the features of networks built on the basis of these switches but also makes these switches unable to work together, in contrast to the original goal of OpenFlow [6]. The need for different switches to be able to work together is a serious concern. For example, in a data center or cloud environment, hardware switches are responsible for host interconnections, and software switches are responsible for the intraconnections among the virtual machines in a host or

between hosts in overlay networks. In [18], three commercial OpenFlow switches (5406zl, P-3290 and 8132F) were tested and found to exhibit different characteristics with respect to rule modification, rule reordering for updates, and the time of synchronization with the CP. The poor compatibility and interoperability of OpenFlow switches make them difficult to manage and can cause further issues such as rule loss and rule reordering. This issue also is one of the reasons why OpenFlow-based SDN has not achieved fast adoption.

These above-mentioned issues concern forwarding rules. However, for complex NFs or software-defined DPs (e.g., P4, POF or Click [79]), the issues may be different in nature, and analyzing forwarding rules to check the network behavior may be inefficient. On the one hand, many NFs are stateful and depend on the flow history for their processing of network traffic. Although many approaches have attempted to add SDN dimensions for the steering of flows in NFs [62], [80]–[82], current SDN technology does not offer a sufficient control capacity for NFs. To find the root causes of network failures, we may need to analyze the actual implementations in the source codes of these NFs. On the other hand, many new software-defined specifications can provide greater programmability for new DPs. With these specifications, a network operator can design a custom DP in a flexible manner and implement specific functions for satisfying any network demand. These specifications are powerful but cause faults that occur in the DP to be more complex. Logic errors or design flaws in DP programs may result in network failures, and thus, such programs also require careful inspection.

B. Control Plane Faults

The SDN controller is a software entity that is not bug-free. Software errors in the CP can lead to a controller behaving abnormally or even crashing, and they can also induce network problems in the DP [16], [83]. In addition, flaws in critical design and logic in SDN controllers can degrade SDN reliability [84]. When efforts are made to satisfy the requirements related to the management of highly dynamic and flexible networks, the incidence of flaws in controller software may increase because the needs of large-scale complex software programs are driving those networks. Insufficient domain knowledge, incorrect evaluations of service requirements and network capabilities, poor assumptions about the network environment and unexpected concurrency issues [84] all make faults in the CP inevitable. Since the CP is the core of an SDN network, the effects of CP faults are felt throughout the network.

CP faults can not only affect the reliability of the CP itself but also harm the whole network. In this survey, we categorize faults in the CP into four types: policy violation (Section III-B1), state mistranslation (Section III-B2), inconsistent controller states (Section III-B4), and incorrect event processing (Section III-B3).

1) Policy Violation: As discussed in Section III-A, although some issues in the DP can induce incorrect network configurations, their main causes, such as incorrect network policies and policy mistranslations, originate from the CP. This is an extremely common issue in SDN networks, and a number of studies on fault diagnosis (e.g., [16], [19], and [20]) and recovery (e.g., [85]–[87]) have resulted in the design of various policy verification mechanisms to address this issue. In these works, a list of fundamental network-wide invariants is commonly considered in combination with network operators' desired properties as the basis on which to verify the correctness of the generated configurations. An invariant represents a basic correct network state (e.g., reachability or being loop-free), and a violation of an invariant can directly induce abnormal network states. Thus, we call this type of fault in the CP, in which the configurations generated by the controller violate the operators' designated network policies, a *policy violation*.

We surveyed several academic publications [16], [23], [83], [88]–[92], based on which we can summarize the root causes of policy violations as follows:

- *Policy flaws:* Logic errors or human mistakes in policy writing can directly violate desired properties and result in network failures [23], [88], as will be described in the next Section III-C.
- *Policy conflicts:* A policy conflict is a situation in which two network policies have overlapping domains for network manipulation [89], [90]. This is a common issue in the CP since SDN allows apps from different vendors to coexist in the same CP for network management; it is difficult to ensure that there are no conflicts among their policies, and this can lead to race conditions, e.g., competition for shared network resources (e.g., link bandwidth, topology or switch CPUs) [25], [69] and rule priority overlap [25], [69], [93]. In OpenFlow networks, this issue manifests as flow *rule conflicts*, in which two flow rules can match the same flow but may dictate different actions. The OpenFlow specification defines a *priority* to distinguish these rules such that packets can be matched only by the rule of higher priority. However, when network operators want to deploy new policies into networks without impacting existing network services, if a newly generated rule is assigned a higher priority than an existing rule due to insufficient or incorrect domain knowledge on existing network policies or incorrect assumptions on the operating environment, this can cause a rule to overlap with existing network services. In turn, this can result in a network state that is inconsistent with the operators' intentions [89], [90].
- *Policy mistranslations:* After being generated by apps, policies are translated by a controller into low-level instructions for installation in the DP. Errors in this translation process can lead to the generation of unexpected rules, which may also be inconsistent with the operators' desired properties [91], [92]. This problem is further analyzed in Section III-B2.
- *Incorrect event processing:* Bugs in CP components (e.g., OpenFlow packet handler [16], [17], [83] and link discovery [16]) can cause incoming events/messages from apps to be processed incorrectly; this may result in incorrect decisions regarding network updates. We discuss this issue in detail in Section III-B3.

2) Information Mistranslation: Controllers are responsible for providing abstractions of the network state to upper-layer apps and translating policies from apps into low-level instructions for the DP. We use the term "information mistranslation" to refer to a fault in which states or policies are mistranslated. These faults are caused by errors (e.g., logic flaws, coding mistakes and software misconfiguration) in state abstraction and policy translation. Such a fault may have a very small impact on the common control logic implemented based on the low-level abstraction (e.g., OpenFlow) of the DP. However, for high-level programming languages [24], [48], [94], [95] that aim to simplify network programming in SDN, their susceptibility to such faults is one of the main hindrances to their success. These programming languages constitute the main paradigm for SDN app development and enable network operators to focus on which networks they want to implement without needing to consider how to implement them [24], [47]. For apps written in high-level programming languages, a compiler, or more precisely a language interpreter is applied to translate the "what" (i.e., the high-level properties defined in the apps) into the "how" (i.e., low-level flow rules or other configuration commands) based on the network abstraction. This, however, is also not a bug-free process and can lead to mistranslations of information (i.e., policies and network states) for the DP and AP as well as inconsistent states and network failures. Verifying the correctness of the process of state translation in the CP is necessary to ensure cross-layer state consistency in SDN.

3) Incorrect Event Processing: In the SDN paradigm, each component (e.g., switches, controllers and apps) leverages request and response events to maintain contact with the other planes. The CP can receive external events from switches and apps as well as internal events (e.g., master election, distributed database reading/writing or component collaboration) among controllers, and it will follow its inherent logic to process each received event and generate responses to be sent to the originating component or forwarded to others. Design and logic flaws or software bugs in controller subcomponents can directly result in the generation of abnormal actions or responses, delays in the responses to other events or even dropped events [16], [17]. We refer to such faults as *incorrect event processing*.

To describe this type of fault more intuitively, we present an example of a bug in POX found in [16]. In POX's discovery module, there is a logic error for *Packet_In* handling that can lead to a race condition in which a *LinkEvent* event is first sent to apps rather than to *SwitchUp* when a premature *Packet_In* is forwarded to POX. Bugs in the CP are very common and varied; for example, each component of the OpenDaylight controller contains many bugs,[6] and numerous new bugs continue to emerge as the controller evolves. The occurrence of such a fault in a network can lead to many issues such as abnormal network behaviors, missing controller features, incorrect actions in apps, invalid controller access, and even crashes of a component or an entire controller. To guarantee SDN reliability, we need to design more effective fault diagnosis techniques for finding faults in the CP and fault tolerance techniques for preventing component failures or system crashes.

4) Inconsistent Controller States: To achieve high availability and scalability, the SDN CP is currently designed to be logically centralized but physically distributed. As a distributed system, the CP needs complex software to ensure data sharing, state synchronization, module collaboration, and access management among different CP components. Although a distributed CP can be used in a large-scale network, errors in the distributed control system can induce various failures (e.g., distributed database locking, master re-election or inconsistent event actions), which can impact the network processing performance of the CP [16], [17].

The root causes of *inconsistent controller states* are various and include logic and design flaws, coding mistakes, concurrency errors, incorrect algorithms, unexpected operations, hardware failures, and connection interruptions. Note that these issues arise in most distributed systems, including SDN controllers, and are difficult to diagnose or prevent. To resolve such issues, greater effort needs to be applied toward optimizing and improving the distributed architecture.

C. Application Faults

SDN apps constitute the main "brains" of networks. Each app implements its own control logic to manipulate networks by means of either general-purpose languages (e.g., Java, C++ or Python) or domain-specific programming languages (e.g., Frenetic [24], Pyretic [48], FlowLog [94] or FatTire [95]). However, these apps all potentially contain software faults that can affect the entire network, e.g., incomplete specifications, incorrect algorithms, design mistakes, programming bugs, coding mistakes, incorrect installations, and user mistakes. We divide these faults in apps into two groups: *incorrect program implementations* (Section III-C1), which can result in network events being processed incorrectly, and *policy flaws* (Section III-C2) (also called control logic flaws), which can cause network states to violate operators' desired specifications.

1) Incorrect Program Implementations: We use the term "incorrect program implementation" to refer to a fault that is caused by software bugs or coding mistakes and can induce unexpected behaviors during run time or when processing network operations. Such faults are quite common in software engineering [96] and may be more complex in SDN since SDN programs often need to reason based on a massive amount of network information to make forwarding decisions. Any coding mistakes, concurrency errors, incorrect operations, or incorrect installations can potentially induce abnormal app behaviors or even network problems, such as the generation of requests with incorrect parameters or the mishandling of network events from the DP or NBI requests [16], [97], [98]. A simple example of an incorrect program implementation is detailed as follows. To process a new OF Packet_In message, three separate tasks need to be implemented: 1) caching the message, 2) generating and sending Packet_Out messages

[6]For detailed descriptions, the reader is referred to the OpenDaylight Bugzilla (https://bugs.opendaylight.org/).

to other nodes to find the destination of the message, and 3) generating Flow_Mod messages to install flow rules in related nodes for guiding traffic consisting of packets of this type. If the programmer forgets about task 3), packets of this type will never be forwarded. Because these programs are executed under conditions of complex network states, diagnosing faults becomes a thorny issue [23], [46].

2) Policy Flaws: Although programming apps with domain-specific programming languages can reduce the occurrence of incorrect program implementations, insufficient network knowledge or incorrect assumptions about the network environment can also result in faulty policies. A fault of this type is called a *policy flaw* in this survey. Policy flaws in apps are the main cause of violations of operators' desired specifications or fundamental network properties, e.g., forwarding loop-freedom, isolation among groups, and basic reachability. Many research efforts [19], [20], [23], [46] have focused on identifying these issues from flow rules generated by the CP or from logic in the source codes of apps; some tools [87], [99], [100] can automatically produce fault patches to repair such flaws.

IV. System Monitoring

Data are crucial in enabling network operators to promptly find and comprehend problems that compromise SDN reliability. In this section, we focus on *system monitoring* for SDN networks, namely, the tracing of system behaviors and the collection of data from various SDN components for fault management. For traditional networks, many *network measurement* tools, such as tools for instrumentation (e.g., `syslog`), traffic counting and sampling (e.g., NetFlow, sFlow and SNMP), traffic mirroring (e.g., wireshark), probe testing (e.g., `ping` and `Iperf`), and packet tracing (e.g., `IP traceroute`), have been developed for monitoring network states. Although SDN can provide more opportunities for advancing these tools, it also poses challenges for system monitoring because of the multiple software components (e.g., data agents, controllers, and apps) and interaction channels involved in network operations. Several types of data are involved in SDN system monitoring; in this section, we divide these data into four types, namely, *probe packets*, *traffic statistics*, *channel messages* and *system events*, and discuss approaches for collecting these data. For clarity, Table IV summarizes these four types of data and their corresponding monitoring approaches, their functions and their advantages and disadvantages for SDN fault management.

A. Probe Packets

As discussed in Section III, the actual behaviors of a network can be inconsistent with the network configurations generated by the CP due to rule loss or priority reordering. Thus, the network snapshots built from control messages are not fully credible [15]. To address this issue, multiple probe-based techniques have been developed, in which specific probe packets are injected into the DP and the probe results are collected for further forwarding inspection. The approaches used for the generation and collection of the probe packets in such monitoring mechanisms can be divided into two types: *test host*-based and *caching rule*-based approaches. A more detailed discussion of the usage of these probe packets can be found in Section V-A.

In the *test host*-based approach, test hosts with testing agents are deployed around switches, and these test hosts are responsible for generating probe packets based on specific testing strategies, collecting the probe results, and then sending these results to the controllers or to a dedicated analysis server [28], [101], [102]. This approach allows packet probing to be performed with minimal interference with normal network operations; however, it requires additional devices and is infeasible for large-scale networks.

Instead of using additional devices, the *caching rule*-based approach involves writing caching rules into switches downstream of the target switch, and it leverages SDN controllers to generate probe packets and inject them into the DP through a control channel; these probe packets will subsequently be trapped by the deployed caching rules [15], [70], [77]. The *caching rule*-based approach can also be called *packet trajectory tracing* [103]–[105], and methods based on this approach can be divided into three types based on the use of caching rules. In methods of the first type, caching rules are used to send matched packets directly to controllers, which then reinject these packets to the DP [15]. In methods of the second type, the packet header information at each hop is copied to the controller [70], [106], [107]. Methods of the third type use caching rules to encode path information (e.g., switch identifiers and path flags) in the header of each probe packet; when the packet is sent to the controller, this information is used to decode the forward path of the packet [103]–[105]. This approach can provide more precise data about network behaviors, but it can also induce traffic overhead in the control channel and interfere with normal network operations. Due to the limited memory capacities of the switches, one also needs to be concerned with the trade-off between the resource overhead for switches and the accuracy of the measurements.

B. Traffic Statistics

For monitoring network behaviors, traffic statistics are also an important type of data that can provide necessary information on the network state (e.g., network topology and link bandwidth utilization) for network manipulation. Traffic statistics are often collected and stored in the local storage of switches in the form of various metrics and are then proactively reported to or passively extracted by controllers. This type of monitoring is called *network measurement*. With the benefits of SDN, network measurement techniques for collecting traffic statistics can be optimized in terms of both accuracy and overhead. In this section, we introduce the techniques for network measurement in SDN.

1) Traffic Counting: Counting the packets traversing switches from the memories of those switches, via mechanisms such as NetFlow, sFlow and SNMP, is a common approach for collecting traffic statistics. SDN protocols also provide many types of counters, such as per-port counters, per-rule

TABLE IV
SDN SYSTEM MONITORING FOR FAULT MANAGEMENT

Data Source	Short Description	Collection Approach	Functions	Characteristics
Probe packets	Probe packets are injected to test the actual network behaviors.	Recording by test hosts or caching rules in switches	Enable inspection of the actual network forwarding behavior in the DP.	High accuracy and potential link overhead.
Traffic statistics	Network traffic statistics represent network states.	Traffic counting	Provide coarse network monitoring.	Easy deployment and low accuracy.
		Packet mirroring	Provide more detailed packet information for external network analysis.	High accuracy and high overhead.
		End-host monitoring	Provide network measurements from endpoints.	High scalability, fine grainularity and massive volume.
Channel messages	Interaction messages are exchanged among the three planes.	Recording by controllers or an external monitoring proxy.	Build snapshots of the whole SDN system for network maintenance and verification.	Low computing cost, low signal load, potentially low credibility and potential for interference with network operations.
Software behavior	Software behavior is reflected by program executions and state changes in control software.	Logging, instrumentation.	Used to implement fault debugging and troubleshooting for SDN software.	High maturity, low accuracy and massive volume.

counters and per-group counters [40], [41], for collecting traffic statistics. The statistical data collected by these counters can be used in faulty device localization, traffic matrix estimation, network anomaly detection and configuration verification [108], [109].

However, these counters do not currently support sampling and thus can result in high overhead with respect to switch memory [110]. Moreover, not all counting data are useful for evaluating certain requirements, e.g., detection based on time-window data. In addition, to ensure satisfactory network forwarding performance, these counters can catch traffic events only at a subsecond rate and update their statistics every second [111], [112], which is too slow for stringent real-time monitoring requirements. To address these issues, many researchers have combined SDN protocols with traditional network measurement mechanisms to reduce memory overhead and improve measurement granularity [113], [114]. To reduce the overhead imposed by traffic counting, Hu and Luo [113] implemented a network tomography algorithm in which only a few OpenFlow rules are set up for direct measurements and other flow data are inferred based on SNMP link counters. OpenSample [114] leverages sFlow packet sampling to provide near-real-time measurements.

There are also other tools, such as UMON [29] and StateMon [115], in which the focus is placed on decoupling measurement policies from forwarding policies to provide flexible and fine-grained measurements. These tools rely on an additional monitoring table with more fine-grained match-action entries (e.g., TCP FIN, TCP SYN and ACK, which are not supported by OpenFlow counters) in the OpenFlow flow table pipeline for monitoring specific flows. With this table, measurement control can be decoupled from forwarding control, and specific APIs can be provided to allow network operators to specify custom measurement policies. Based on this idea, several highly efficient measurements, namely *sketch-based measurements* [30], [116], [117], have been proposed. A *sketch* is a programmable data structure for collecting and storing flow information in the DP and can effectively reduce the overhead of data collection. It can also improve the flexibility of measurement deployment and support more measurement features, e.g., bit checking, different levels of measurement granularity, and specific probabilities [30], [116], [117]. In addition, with the emergence of P4, this approach can be easily implemented in network devices.

2) Packet Mirroring: Since the implementation of the above approaches often requires switches to be modified, these approaches are not supported by all types of devices in today's networking world. Port mirroring is an alternative monitoring approach that is supported by most modern switches. When port mirroring is set up, a switch will send a copy of every network packet seen on one traffic port to another measurement port for external traffic analysis. Port mirroring can be more easily implemented in SDN networks; specific port or flow mirroring rules can be installed in switches such that the matching packets will be copied to a network analysis agent [106], [107]. Here, we call this approach *packet mirroring*.

Packet mirroring has been applied in many SDN fault management solutions to enable accurate network analysis [70], [106], [107]. By installing mirroring rules in switches, `ndb` [106] can tell switches to create a "postcard" (which consists of a packet header, output ports, the version number of the matched rule and a switch ID) for each packet traversing a switch and send this postcard to a packet history analysis agent for fault identification. A similar idea is also implemented in Planck [118] and EverFlow [119]. Planck focuses on mirroring the traffic at ports in a single commodity switch to a directly attached server through the installation of mirroring rules; thus, millisecond-level network measurement can be implemented. EverFlow uses mirroring rules with more matching fields (e.g., TCP SYN, FIN and RST) to collect network packets in a large data center network. Packet mirroring can provide detailed network information for the detection of various abnormal network states, e.g., arbitrary packet loss, link congestion, and forwarding loops. However, high overheads are associated with both mirrored packets and "postcards".

3) End-Host Monitoring: The aforementioned approaches are all implemented in switches; thus, they require switch support, and they can lead to increased memory consumption and computing loads for the switches. Recently, several proposals have pushed network measurement tasks to end hosts, with a uniform interface at each end host and a concentrated controller for processing any queries; this

approach can reduce traffic overhead in terms of SDN network resources [112], [120], [121].

HONE [120] combines end hosts with switches to implement certain measurement tasks, thereby providing a uniform interface for querying the states of network devices via the concentrated controller. With end-host monitoring, a wide variety of management tasks, such as performance diagnosis, the collection of TCP statistics and link utilization calculations, can be implemented. However, computing these data locally may result in a lack of fine-grained control. To address this issue, Felix [121] generates matching filters from high-level user queries and routing configurations to control local data processing. Trumpet [112] focuses on implementing end-host measurements at line speed for a wide variety of monitoring use cases, e.g., detecting correlated bursts and losses, identifying the root causes of transient congestion, and detecting short-term anomalies.

C. Channel Messages

SDN maintains several types of channel interfaces among its components, including the SBI, NBI, EBI and WBI. By monitoring messages over these channels (namely, *channel messages,* including *SBI messages*, *NBI messages* and *EBI/WBI messages*), the interrelations among the planes can be reconstructed and used to provide global visibility of the whole SDN system for fault management [16], [17], [20], [88], [97].

1) SBI Messages: In the SDN paradigm, SDN controllers leverage southbound protocols (e.g., OpenFlow and ForCES) to manage network devices and implement custom network services in the DP. Monitoring SBI messages can provide visibility of all network configurations and state changes in the DP [19], [20]. For example, snapshots of the network (including the network topology and the forwarding function of each device) can be constructed from these messages and then used to verify the correctness of the network states via advanced formal verification or other checking techniques, as discussed in Section V-A1. In addition, these data can be used to validate the correctness of controller actions for southbound API requests [16], [17].

To collect the messages sent over the control-switch channel, a common approach is to use the controllers to record them. However, this is useful only for open-source SDN controllers and may, in turn, increase the computing costs and signaling loads for the controllers. Another approach is to deploy an external collector proxy on the control-switch channel to intercept and replicate these messages and subsequently send them to an out-of-band server for further analysis [19], [20]. Although this approach can avoid the imposition of additional overhead on the controllers, it is difficult to ensure that no additional latency will be incurred for communications between the controllers and switches. In addition, the occurrence of *incorrect rule installations* in the DP (Section III-A2) can affect the credibility of the network snapshots constructed from SBI messages.

2) NBI and EBI/WBI Messages: Similarly, NBI and EBI/WBI messages also are interaction events, but NBI messages are used for communications between the CP and AP, whereas EBI/WBI messages carry communications among distributed controllers. These messages can also be collected using the same approaches applied for SBI messages and can be used to analyze the correctness of the corresponding interactions (e.g., policy logic updates, message requests/responses, data synchronization, and monitoring/notification) between controllers and apps or among controllers [16], [17], [88], [97], [98]. However, unlike for SBI messages, there is currently no standard for messages exchanged through these two types of interfaces, and various common APIs or custom protocols are implemented in different controller platforms [1]. This situation may result in low compatibility and interoperability among different controllers and apps. Thus, the main challenge in implementing an efficient monitoring mechanism for NBI and EBI/WBI messages is how to optimize that mechanism for different APIs to achieve the maximum performance.

D. Software Behavior

As a software entity, an SDN controller often consists of numerous program codes, which constitute various modules and are distributed in different machines to guarantee high availability (HA) and scalability. The monitored actions of these program codes can be used to explain the software behaviors of controllers, such as how external events (e.g., Packet_In and NBI requests) are handled, how network services are materialized into underlying network devices, and how these controllers maintain their scalability and efficient network control capabilities [16], [17]. These data are crucial for finding and understanding problems in controllers that induce network anomalies or function failures; such problems appear to be particularly important for the development of SDN controllers in this immature stage of SDN.

In software engineering, various profiling techniques [96], [157], [158] have been developed for monitoring software behaviors, e.g., program logging, profiling and code instrumentation. Program logging is the most popular tool for recording system events and can be used to achieve basic debugging functions. The logging data can be used to construct control flow graphs or program workflows to determine the root causes of faults that have occurred [158]–[160]. Since many controller platforms (e.g., ONOS and OpenDaylight) are implemented based on distributed architectures, many advanced software tracing tools can be applied in SDN networks, such as Google Dapper [158], which is designed for large distributed systems and implemented based on code instrumentation. Interested readers are referred to [96] for more details on monitoring and debugging mechanisms in software engineering.

V. Fault Diagnosis

Troubleshooting networks is always a difficult and arduous task, especially when combined with the multi-tier architecture and complex network states in SDN. Given the detailed SDN fault characteristics described in Section III and the SDN system monitoring techniques discussed in Section IV, we provide an overview of the currently available fault diagnosis solutions

TABLE V
FAULT DIAGNOSIS TAXONOMY FOR SDN

SDN Element	Solution	Faults	Research Work	Manifest Features
DP	Static configuration verification (Section V-A1)	Policy violations	[27], [122]–[124]	Analyze network properties based on configurations, policies or flow rules for correctness verification.
	Real-time configuration verification (Section V-A1)	Policy violations	[19], [20], [64]–[66], [68], [125]–[127]	Verify network behaviors in real time.
	Formal probe testing (Section V-A2)	Inconsistent rules	[15], [28], [70], [77], [101], [102], [128]	Inject probe packets into the DP and inspect the correctness of the network behaviors reconstructed from the probe results against with defined invariants.
	Route tracing (Section V-A3)	Inconsistent rules	[72], [103]–[105], [129]–[131]	Leverage specific rules to encode traffic path information into probe packets.
	Network interactive debugging (Section V-A4)	Inconsistent rules, policy violations and protocol compliance errors	[106], [107], [132]–[134]	Debug and analyze network behaviors based on network observations.
	Testing for data agents (Section V-A5)	Protocol compliance errors	[55], [56], [135], [136]	Test the compatibility and interoperability of OpenFlow switches.
CP	Conflict verification (Section V-B1)	Policy conflicts	[74], [89], [90], [137]–[142]	Resolve race condition problems for all hardware or software resources or rules.
	Controller verification (Section V-B2)	Information mistranslations and policy conflicts	[91]	Provide verifiers for policy conflicts and information mistranslations when compiling high-level policy languages.
	Software-based debugging (Section V-B3)	Policy violations, incorrect event processing and inconsistent controller states	[16], [83], [85], [143], [144]	Record network events and replay them to provide debugging functions.
	Controller evaluation (Section V-B4)	Inconsistent controller states and incorrect event processing	[17], [145]–[149]	Evaluate SDN controllers with test events to identify abnormal actions or other performance issues.
App	App verification (Section V-C1)	Policy flaws and incorrect program implementations	[23], [45], [46], [97], [150]–[154]	Verify the correctness of policies specified in apps based on the desired properties.
	App testing (Section V-C2)	Policy flaws and incorrect program implementations	[88], [98]	Design black-box testing mechanisms for SDN apps to check the correctness of their behaviors.
All	Integration (Section V-D)	All potential faults	[92], [155], [156]	Strive to provide an overall fault diagnosis framework for SDN.

that have been proposed for SDN networks. As illustrated in Table V, we classify these fault diagnosis solutions based on the methodology applied for each element (i.e., at the DP, CP and App levels) in the SDN stack following a bottom-up approach. Finally, we close this section by discussing work on the integration of fault diagnosis for the whole SDN paradigm.

A. Fault Diagnosis for the Data Plane

By properly configuring network devices, network operators can implement various network services based on specific requirements. As discussed in Section III-A, faults in the DP can potentially induce unexpected network states that are quite different from those desired by operators. Many network troubleshooting tools (e.g., `ping`, `traceroute` and `tcpdump`) have already been proven to be inefficient for SDN as a newly emerging networking paradigm. In this section, we discuss fault diagnosis for the DP based on five main approaches: *configuration verification* (Section V-A1), *formal probe testing* (Section V-A2), *route tracing* (Section V-A3), *interactive network debugging* (Section V-A4) and *testing for data agents* (Section V-A5).

1) Configuration Verification: In any networking paradigm, configuring networks toward desired specifications is an arduous and error-prone process, and SDN is no exception. Recently, a number of network verification techniques [19], [20], [27], [122] have been proposed for detecting issues in SDN network configurations. In these techniques, the network states (obtained from network configurations or control messages) are modeled in the form of specific data structures or formal expressions, and their *correctness* is verified based on the desired properties. We categorize these research solutions into two groups: *static verification* and *real-time verification*.

a) Static verification: Static verification originated from an early paper [63] in which the whole network was modeled as a 3-tuple $G = (V, E, P)$, where V is the set of devices, E is the set of links between vertexes, and P represents the forwarding policies applied on links. By virtue of its centralized control strategy, SDN can simplify the verification process by providing global visibility of the whole network.

Header space analysis (HSA) [27] was proposed as a general and protocol-agnostic framework for statically checking for configuration issues. In HSA, the packet header space is modeled as a concatenation of bits, and a packet is represented as a point in the header space, which can support newly emerging protocols and arbitrary field formats. All NFs are modeled as *box transfer functions*, which transform one subspace into other subspaces, and the entire network is formulated as a network transfer function and a topology transfer function. Based on this model, several search strategies are applied in HSA to identify many configuration issues in the DP.

HSA has been applied in many SDN research articles and commercial products, [16], [19], [133], [134], [141]. However, HSA has some limitations, as follows: 1) *Low accuracy:* It is difficult to distinguish different packet types, e.g., IP and TCP, with the HSA model. 2) *Low scalability:* HSA assumes fixed forwarding rules and fixed packet headers, which may cause it to fail in complex stateful networks. 3) *Low expressiveness:* The specifications are written with ad-hoc codes.

To address these issues, many mature formal methods (summarized in Table VI and further detailed in [38]) are used to verify network correctness based on formal models and provide counterexamples [122], [123]. In these solutions, the terms *implementation* and *specification* refer to the actual

TABLE VI
FORMAL METHOD

Technique	Definition	Implementation	Specification
Model checking	Automatically check whether a specification holds in all states of an implementation.	A finite-state transition graph.	A temporal logic.
SAT solving	Given a propositional formula with Boolean variables (AND, OR and NOT), determine whether there is any satisfying assignment or whether it is possible to prove that none exists.	Suitable models for extracting assignments.	A propositional formula.
Theorem proving	Given an arbitrary theorem, verify the truth of the theorem with formal proofs.	A formal logic.	A formal logic.
Symbolic execution	Abstract programs to symbolic values and explore all possible feasible execution paths.	Symbolic values.	/

network states and the desired network properties and policy specifications, respectively. In addition, the *fault localization problem* is reformulated as a formal verification problem based on formal models (e.g., finite state machines (FSMs) and binary decision diagrams (BDDs)), which can then be solved with mature verification tools (e.g., Z3, Alloy and KLEE). For example, in FlowChecker [122], the verification problem is solved through *symbolic model checking*, in which network configurations are encoded as BDDs and the desired properties are written in computational tree logic (CTL). Anteater [123] treats configuration analysis as a Boolean satisfiability (SAT) problem, in which the packet header is represented by a symbolic variable and the network is modeled as a 3-tuple $G = (V, E, P)$.

In addition to comparisons with network properties, Xu *et al.* [124] considered verification based on SDN controller states. In their approach, configurations are extracted from both controllers and end hosts, their states in three network layers (L2, L3 and L4) are modeled as BDDs, and cross-plane correspondence checking is performed to identify any differences in the mappings between the controllers and end hosts in the same layer. This approach offers a more detailed verification of the states in different network layers, e.g., L2 connectivity, L3 reachability, and L4 security groups and packet filtering.

b) Real-time verification: Static verification is an offline process and cannot be used to monitor the correctness of the current network state in real time. Recently, several real-time verification solutions have been proposed. The core idea of real-time verification is to intercept and replicate control messages by establishing a proxy between controllers and switches (as described in Section IV-C) to obtain network update messages at run time and incrementally update the network model to verify the network configurations.

When implementing real-time verification, the time efficiency of the verification process is key. Several solutions based on network slicing have been proposed [19], [20], in which the network model is sliced into *packet equivalence classes* (ECs) to allow the verification to be processed in parallel. Here, an EC is a set of packets that experience the same forwarding actions. Veriflow [20] utilizes a *trie* structure to search for the rules in the ECs that are affected by a new rule, and it models these affected ECs and their forwarding states as forwarding graphs. NetPlumber [19] builds on HSA to incrementally model packet transfer and uses a dependency graph (plumbing graph) to represent the relationships among different rules. It also clusters the graph into several subgraph-based ECs and generates and sends HSA "packets" based on specific queries into these subgraphs (which are related to the new rules) to check the validity of policies and invariants in parallel, thus achieving near-real-time performance similar to that of Veriflow. Through partial analysis performed by means of incremental algorithms, these two tools can perform verification within hundreds of microseconds. While Veriflow provides functional APIs for invoking different traversal strategies to verify various violations of network invariants depending on users' queries, NetPlumber provides a formal language in which to express policy checks and supports more verification functions (e.g., arbitrary header modifications) than Veriflow does due to its protocol-agnostic HSA model.

Seeking a more time- and space-efficient approach for network verification, Yang and Lam [64] proposed *atomic predicates* for specifying packet filters and transformers, which are the coarsest ECs. The AP verifier [64] precomputes the set of atomic predicates for all port predicates in the network and computes separate sets of atomic predicates for forwarding and ACL predicates (represented by BDDs) in real time. Then, it generates a reachability tree labeled with the identifiers of the atomic predicates to search for network property violations. Since the time requirements can be greatly reduced by computing operations on these identifiers rather than on the packet header fields, the AP verifier can better achieve real-time network verification. Yang *et al.* then considered a problem observed in several previous real-time verification approaches, namely, their limited scalability to packet transformations [19], [20]. To address this problem, they proposed APT [66], in which the packet header is represented by a stack of protocol headers. Like the AP verifier, APT also relies on a transformation to generate atomic predicates represented by BDDs to verify the desired network properties.

Although these packet-equivalence-based approaches are efficient for real-time verification, they also raise a problem of how to efficiently handle operations that involve large numbers of ECs [125]. To address this issue, Delta-net [125] exploits the network characteristic that similarities among the forwarding behaviors of packets can be identified from parts of the network rather than its entirety. Instead of slicing the network model, Delta-net leverages a single edge-labeled graph to represent the entire network and incrementally transforms that

graph as network updates. It then uses the concept of *atoms*, as proposed in [64] and [66], and automated precision refinement on the graph to achieve real-time verification.

In [126], a stepwise-refinement-based verification tool, Cocoon, was proposed to further improve the speed of configuration verification. In Cocoon, the verification process is separated into static verification and run-time checks. A programming language and a verifier for this language are provided to allow users to specify a high-level view of the correct network behavior. Cocoon uses a set of run-time-defined functions (RDFs) to capture run-time configurations and checks them against static assumptions defined at design time to find violations. This separation offloads most of the real-time check cost to the static verification process, thus enabling faster verification of configuration correctness than is possible with NetPlumber and Veriflow.

The real-time verification tools introduced above all collect DP snapshots from the control channel. However, collecting snapshots has a potential shortcoming in that, in a large-scale dynamic network, it is difficult to guarantee consistency between a collected snapshot and the network state due to the frequent changes, unsynchronized updates, and update delays in the network. Libra [68] attempts to handle this issue by first replaying the network events within a specified period to reconstruct the current state of the network. Then, for verification, it uses MapReduce to slice the network (modeled as a directed graph) into subgraphs according to different prefixes and analyzes these subgraphs in parallel. Thus, Libra can handle both snapshot inconsistencies and many forwarding faults.

Another issue that arises with these real-time verification approaches, including static verification, is that the network models are programmed in a general-purpose language (e.g., C). Such languages may suffer from scalability issues, especially for stateful or new protocol networks because of the more complex network states involved. Panda *et al.* [127] designed a restricted language for modeling middleboxes and then used an SMT solver, Z3, to verify pipeline and isolation invariants in networks. Their approach leverages Veriflow to check the input topologies and forwarding tables and produces a forwarding graph, to which the user can add assertions describing the physical behavior of the network, which are then provided to Z3 for verification. NOD [65] is an optimized version of Datalog for expressing specifications and network models; it was developed by modifying the Z3 Datalog implementation to provide solutions for all reachability queries with the support of software-defined elements, e.g., P4 and OpenFlow.

c) Discussion: The network verification techniques introduced above are summarized and compared in terms of their network models and functions in Table VII. These research solutions can be divided into two classes: those that rely on new data structures for modeling networks (HSA, NetPlumber, VeriFlow, AP verifier, APT, Delta-net and Libra) and those that represent networks using formal models (FlowChecker, Anteater, Panda and NOD). While the former may be more consistent with the network characteristics than formal models that enable real-time verification are, the latter can provide more flexibility in verification and can support complex networks by leveraging expressive modeling languages and mature verification tools for formalizing network implementations and property specifications. However, the problem of state-space explosion in formal verification needs to receive greater attention with regard to optimization when formal models are applied for network fault diagnosis. Note that although these network verification approaches can identify potentially abnormal network states in the DP, they cannot address *inconsistent rule installation* issues in the DP since the verification is performed solely on the basis of the controllers' views. They also cannot effectively verify stateful network behaviors since the snapshots obtained from the control channel are insufficient to explain these behaviors.

2) Formal Probe Testing: Formal probe testing is a process that leverages probe testing in combination with formal verification techniques to provide a dynamic verification solution for the rule inconsistency issue in SDN networks. Such testing is commonly based on the assumption that the policies in the CP are correct. Probe packets are then generated to observe the actual DP behaviors to validate the consistency between these two planes. Two problems arise in the implementation of such formal testing: *how to generate probe packets for different verification purposes or to cover the entire network* and *how to capture the probe results with low overhead*. By providing a logical view of the entire network, SDN can facilitate packet generation. However, faults in the DP (e.g., software bugs, rule overlap [77], and priority conflicts [70]) also pose challenges for network inspection. Thus, a key question in formal testing is how to implement an accurate and efficient forwarding inspection based on probe results [70].

ATPG [28] leverages the HSA model [27] to model all network behaviors and precompute all possible test packets to cover all rules in each switch. Given a set of test hosts periodically sending and receiving test packets, ATPG formulates the rule liveness problem as a SAT problem that can reveal forwarding and congestion issues in the DP. On the one hand, the use of additional equipment can introduce a scalability problem. On the other hand, the batch packet generation in ATPG may result in much poorer timeliness for large-scale and dynamic networks. To address these issues, Monocle [15] (an extension of ProboScope [77]) formulates the flow table logic in the DP as a SAT problem to generate probe packets and sends them to switches via controllers. Then, it leverages the catching rules installed in the corresponding switches to trap these probe packets. Monocle serves as a proxy between the CP and DP, allowing it to monitor each network event (e.g., rule update) and verify each related rule with probe packets in real time. In addition, in the steady state, Monocle provides periodic testing of the whole network.

However, the verification provided by the aforementioned tools may be inaccurate due to a lack of detailed packet processing information. To overcome this shortcoming, RuleScope [70], [128] leverages a "postcard" [106] (generated through *packet mirroring*, as discussed in Section IV-B2) to obtain more information for accuracy inspection. By means of a carefully designed SAT solver for packet generation and dependency-graph-based fault inference algorithms, it can

TABLE VII
CONFIGURATION VERIFICATION

Tool	Short Description	Issues Addressed	Network Behavior Model			
			Packets	**Network Device(s)**	**Network**	**Expression of Properties**
HSA [27]	Static analysis with a protocol-agnostic structure.	Forwarding failures	Points in $\{0,1\}^L$ space	Space transfer functions in L-dimensional space	Network and topology transfer functions	Ad hoc codes
FlowChecker [122]	Verification based on symbolic model checking.	Misconfigurations and inconsistencies	Packet 4-tuples with locations	BDDs with several constraints	/	CTL
Anteater [123]	Verification based on SAT solving.	Forwarding & transformation failures	Symbolic packets	Boolean constraints	3-tuple graph with devices, links and policies	Ad hoc codes
Xu *et al.* [124]	State consistency verification for SDN-enabled networks in OpenStack.	Inconsistent states between the CP and DP	/	/	Binary matrix for L3 reachability & BDD-based bitmap for L4 state	Ad hoc codes
NetPlumber [19]	Real-time verification based on HSA.	Forwarding & transformation failures	HSA packets	HSA functions	Dependent graph with subgraphs	FML
VeriFlow [20]	Real-time verification based on model slicing.	Forwarding failures	Variables	/	A *trie* structure	Functional APIs
AP verifier [64]	Real-time verification based on atoms.	Forwarding failures	General model with multiple fields	A box with packet filters	A reachability tree labeled with integers	CTL
APT [66]	Real-time verification based on atoms and a protocol-independent packet model.	Forwarding & transformation failures	A stack of protocols to model the header	A box with several transformers	A directed acyclic graph of boxes (BDD)	Fixed policies
Delta-net [125]	Real-time verification with a single edge-labeled graph.	Forwarding failures	Atoms of IP prefixes	A set of atoms	A single edge-labeled graph	Datalog
Cocoon [126]	Combined verification consisting of static and real-time checks.	Forwarding failures	/	/	Run-time-defined functions formulated with SMT	Cocoon language
Libra [68]	Verification based on MapReduce.	Forwarding failures	/	Sliced into subnets for each prefix	A directed graph with several subgraphs	Fixed policies
Panda *et al.* [127]	Verification based on SMT solving for dynamic datapaths.	Pipeline & policy invariant violations	Variables	Model in a declarative language	Forwarding graph produced by VeriFlow	Fixed policies
NOD [65]	Verification based on Datalog.	Forwarding & transformation failures	A separate variable	Predicates	Forwarding model based on Datalog	Datalog

inspect the network for both rule loss and priority faults. This approach can yield a high-accuracy analysis of forwarding behaviors. Unfortunately, the high data collection overhead incurred for packet mirroring is not a negligible issue.

Another difficulty facing these verification tools is that catching-rule-based data collection can increase memory resource consumption and induce interference in normal networks. In addition, the high time cost of the probe generation process still has not been fully resolved in either RuleScope or Monocle. VeriDP [73] offers a different solution for verifying the DP to address these issues. A VeriDP pipeline is added alongside the OpenFlow pipeline in switches to record packets. OpenFlow messages are monitored to construct a path table using a BDD, and the consistency between the path table and the traffic statistics collected from the VeriDP pipeline is inspected to find any inconsistent network behaviors. To localize the root causes of faults, VeriDP traverses all possible paths and infers that the faulty switches lie on the invalid path(s). VeriDP can achieve a verification speed of 3 μs per packet. However, the need to modify switches may limit its widespread adoption.

Currently, many modern networks are stateful. In these networks, many complex NFs (e.g., network address translators (NATs), deep packet inspection (DPI), and load balancers) are mixed with stateless devices (e.g., switches and routers), and most of them are outside the visibility of SDN control. Thus, these formal testing techniques suffer from scalability issues due to the need to process more complicated data. Recently, several proposals, such as FlowTest [101] and BUZZ [102], have been presented in an attempt to solve this problem. In these solutions, probe packets are generated based on a stateful network model. FlowTest models the DP functions as state machines in order to formulate probe packet generation as a formal problem and performs validation in accordance with the probe results. BUZZ [102] models the DP on the basis of the operators' policies and NF models (using the FSM approach) and then generates test traffic based on an optimized symbolic execution to trigger policy-relevant behaviors of the DP model. By means of the DP model and optimized symbolic execution, BUZZ can test a stateful network for policy violations with high efficiency and scalability.

3) Route Tracing: As a complement to inferring forwarding behaviors based on observed packets, *route tracing* focuses on tracing packet trajectories for path inspection and rule verification. In route tracing methods, specific tags are embedded in the headers of target packets traversing each switch to record path information, which can be used to localize faults related to a specific link and provide efficient end-to-end monitoring and verification.

TABLE VIII
PROBE TESTING AND ROUTE TRACING FOR MONITORING NETWORK BEHAVIOR

Reference	Description	Network failures						
		Reachability	Forwarding loop	Rule consistency	Waypoint routing	Unexpected packet loss	Incorrect header changes	Link congestion
ATPG [28]	An HSA-based probe testing tool.	√	√	√	√	√	×	√
Monocle [15]	A network state monitor based on SAT solving.	√	√	√	√	×	×	√
RuleScope [70], [128]	A rule inspection tool based on SAT solving and graph-based analysis.	√	√	√	√	×	×	√
VeriDP [73]	A rule consistency verifier based on traffic statistics.	√	√	√	√	√	×	√
FlowTest [101]	A stateful DP testing framework in which network devices are modeled as state machines.	√	√	×	√	×	×	√
BUZZ [102]	A model-based testing framework for stateful networks based on symbolic execution.	√	√	×	√	×	√	√
SDN traceroute [129]	A traceroute mechanism for SDN networks implemented by trapping packets hop by hop.	√	√	×	√	×	×	√
Everflow [119]	A match-and-mirror-based packet-level network telemetry system with various debugging applications.	√	√	×	√	√	×	√
PathletTracer [103]	A Layer 2 path tracing tool wherein the path tracing is implemented by encoding paths in packets.	√	√	√	√	√	√	√
PathQuery [105]	A network path query system with a declarative query language.	√	√	√	√	√	√	√
REV [72]	A rule enforcement verifier to defend against rule modification attacks.	√	√	√	√	×	×	×
Cherrypick [104]	A packet trajectory tracing tool tailored for networks with symmetric topologies.	√	√	√	√	√	×	√
PathDump [131]	A packet trajectory tracing tool for networks with arbitrary topologies.	√	√	√	√	√	×	√

Similar to `IP traceroute`, SDN traceroute [129] can provide intuitive visibility of packet traversal for SDN network maintenance. It assigns each switch a unique ID and several high-priority rules for forwarding probe packets to a controller, which then re-injects the probe packets into the DP. In this way, hop-to-hop packet information can be gathered. With this tool, operators can find where packet traversal fails and reveal issues such as rules conflicts, controller bugs, and traffic latency in the network. Everflow [119] applied a match-and-mirror strategy to trace individual packets across the network. By configuring all switches with specific rules to trap probe packets tagged with a "debug" bit at a specific sampling rate and mirror each matched packet to send to analyzers via the GRE protocol, packet-level telemetry can be implemented for large data center networks. In addition, Everflow provides several debug applications, including a latency profiler, a packet drop debugger, a loop debugger, and an equal-cost multi-path profiler, in the SDN controller to guide probe packet generation and identify the root causes of specific network failures.

However, sending packets to a controller hop by hop may result in significant overloading of the controller workload and link bandwidth. Some in-band testing tools (i.e., tools that encode path information into packets), in which packets are incrementally embedded with different tags according to specific rules during their traversal and are sent to a controller as their destination, have been proposed to solve this problem [103]–[105]. PathletTracer [103] and PathQuery [105], [130] model network configurations in the form of state machines and encode path queries as state transitions implemented under specific rules, according to which the tags in the packets are changed during their traversal. By analyzing a departing tagged packet to decode its traversal path, these tools can check for consistency issues with high-level policies via state rollback. A similar approach is also applied in REV [72] to address rule modification issues related to unexpected external operations or attacks, but in this case, the tags are updated by means of a secret key shared with the controller in each switch. This type of approach requires only a few packet header fields to carry tags, but it can incur excessive computational costs for generating trace rules and resource costs for installing these rules.

In contrast, Cherrypick [104] directly embeds identifiers into the packet header at each switch and performs link sampling to reduce unnecessary header space costs. The latter capability is implemented based on clever use of a well-structured network topology. This approach can minimize the number of rules required for path queries but suffers from a lack of generality due to its strong assumptions, e.g., a symmetric topology. Thus, the authors of Cherrypick extended this work to support arbitrary network topologies, resulting in a tool called PathDump [131], based on the belief that networks will evolve to support larger packet header spaces to permit the embedding of more identifiers. By providing network operators with APIs for expressing their path queries, these tools can offer many debugging functions for various network failures based on their analysis of packet trajectories.

For clarity, we compare these route tracing and formal probe testing solutions in Table VIII since they have similar diagnosis characteristics.

4) Interactive Network Debugging: Interactive debugging, as performed with tools such as `gdb` and mtrace, is a process that allows software programmers to monitor the execution of a program, stop it, restart it, and set breakpoints. Unfortunately, this approach cannot be directly applied to debug traditional networks since their elements often behave as a black box with distributed protocols. Some network troubleshooting tools, e.g., `ping`, `tcpdump` and NetFlow, provide only limited debugging capabilities. With SDN, the decoupling of the network architecture opens the door toward the development of powerful network debuggers.

By means of *caching-rule*-based traffic tracing, interactive debugging can be implemented for SDN network maintenance. Similar to `gdb`, `ndb` [106] has been proposed as a network debugger with two basic debugging functions (*breakpoint* and *backtrace*) for SDN network maintenance. It traces network flows by mirroring packet information (i.e., generating postcards) at each hop. Then, a breakpoint (i.e., a flag or filter defined in the packet header) is used to catch the target packet, and its forwarding history can also be reconstructed from the collected postcards. Based on the detailed packet processing information, these debugging functions can help network operators to uncover protocol compliance errors, inconsistent rules and controller logic bugs. This work was subsequently extended to develop NetSight [107], which provides APIs for setting breakpoints and more powerful debugging functions, including an invariant violation monitor (*netwatch*), a packet history logger (*netshark*), and a network profiler for link utilization (*nprof*).

`ndb` [106] and NetSight [107] both provide a positive *backtrace* function, which infers the root causes of a fault starting from the observation point (an observed alarm event or several abnormal messages). However, their troubleshooting attempts may fail without a proper starting point for backtrace. To address this shortcoming, Y! [133] utilizes the concepts of *positive provenance* and *negative provenance* to construct a more complete backtrace for detecting configuration issues. In Y!, the positive provenance represents the normal backtrace and the negative provenance is responsible for explaining why the desired network state does not occur. Y! records network behaviors, configurations and packet headers with timestamps and uses this information to construct a provenance graph (i.e., a causal connection tree). By replaying the graph, it processes a backtrace on each possible branch to find the reasons for the observed faults (e.g., logic inconsistencies, failed assertions or policy invariants). DiffProv [134] extends this work by adding a *differential provenance*, which can explain the *differences* between two provenance trees, rather than relying on a single provenance as in Y!. DiffProv finds similar events with the correct behaviors to construct a reference provenance tree by looking back in time at the same system or looking for a different system with similar operations. It then compares the reference provenance tree against a buggy provenance tree based on a branching backtrace approach. Compared with Y! DiffProv can be used to find more network issues, e.g., incorrect rule installations, unexpected rule expiration, multiple faulty entries, and rule conflicts due to multicontroller inconsistencies.

5) Data Agent Testing: As described in Section III-A3, faults in data agents (e.g., in the *completeness and correctness* of OpenFlow implementations and the *compatibility and interoperability* of switches) must be addressed before they are deployed in actual networks. Designing testing processes for SDN switches is useful for finding inconsistent implementations and potential bugs. The main research solutions proposed for addressing this issue are identified in this section.

OFTest [135] aims to provide a unified testing framework with hundreds of test cases for testing OpenFlow switches. Several test hosts are deployed around the switch to be tested, and these hosts are responsible for generating test packets to be sent to the switch based on the test cases and analyzing the results received from the switch. However, the test cases are manually developed, which makes it difficult to evaluate OFTest's coverage of the OpenFlow specifications. In addition, to test any new feature, it is necessary to update the set of test cases. OFTest currently supports only OpenFlow 1.0 and 1.1, with a limited ability to support version 1.3. OFLOPS [76] is another testing framework; it offers functions for packet generation, capture and timestamping for testing OF implementations. However, its objective is to identify the performance characteristics of OpenFlow switches (e.g., OpenFlow packet processing actions, rule update rates, flow monitoring capabilities, and OpenFlow operation interactions) rather than functional errors.

Several other works leverage formal verification techniques to find OpenFlow implementation issues [55], [56], [136]. OFTEN [136] is an interactive testing tool that leverages systematic state-space exploration techniques (i.e., NICE [23]) to generate properties from high-level control logic (i.e., apps) for validating real switch executions. However, the properties extracted from a system may often rely on the developers' knowledge, and it is difficult to guarantee the correctness of these properties for OF switch testing. SOFT [55] focuses on issues of compatibility and interoperability between different switches. It is a white-box testing technique in that it needs the source codes of the OpenFlow agents in different switches to extract their symbolic execution models. Combined with concrete testing inputs, SOFT can find more potential inconsistencies between any two OF agents by comparing their *symbolic trees*. However, the necessary source codes may not be easy to obtain. Yao *et al.* [56] presented a model-based black-box testing approach for SDN DPs. In their technique, the forwarding behaviors of the DP as defined in the OpenFlow specification are modeled as FSMs to generate test packets to be sent to the switches, and a smaller data graph is extracted from the model for performing correctness verification based on formal specifications.

B. Fault Diagnosis for the Control Plane

The SDN CP serves as a network operating system and holds various types of control logic for network provisioning, management and maintenance. However, diverse configuration

issues and software faults (as described in Section III-B) can degrade the reliability of SDN networks. A number of solutions for diagnosing faults in the CP have been proposed. We list simple descriptions of these tools and the types of faults they can localize in Table IX. In this subsection, we classify these solutions and discuss them as follows.

1) Conflict Verification: As an open network platform, SDN inevitably faces the issue of race conditions in the CP, which forces operators to design *conflict verification* mechanisms. In these mechanisms, operators often design a set of basic constraints (e.g., security policies, firewall policies, or existing rule sets) as the properties that newly generated rules need to satisfy.

Rule conflict verification is often implemented based on static network security policies, which comprise a set of non-bypass properties representing the correct behavior of the network. FortNox [89] extends the NOX OpenFlow controller with a security policy enforcement kernel that can check for flow rule conflicts in real time. Based on the set of constraints defined in the security policy, it uses *alias sets* to represent rule information and performs a pairwise comparison between a new flow rule and each constraint. FLOVER [137] leverages the Yices SMT solver to check for conflicts between flow rules and the network security policy. When an OpenFlow controller needs to update the flow rule set in response to a new rule request from the DP, FLOVER formalizes the created rule set with these non-bypass properties as a SAT problem to verify the correctness of the rule set.

As an alternative to performing verification by means of a component in the SDN controller, some solutions implement it through a third-party proxy to reduce the controller workload. Natarajan *et al.* [90] addressed the problem whereby existing virtualization solutions implement network resource isolation only at the policy level and implemented flow table isolation in the DP to achieve fine-grained conflict detection and resolution. They designed a detection system and deployed it close to FlowVisor [161] (an OpenFlow network virtualization technique deployed in between the controller and the switches as described in Section VII-A1) to intercept flow installation messages. Based on this system, they proposed two conflict verification mechanisms. The first mechanism leverages a hybrid hash-trie structure representing the flow tables to search for conflicts. The other mechanism infers conflicting flow entries based on an ontology-based logic inference system. When a conflicting flow is identified, the detection system will drop this flow and report the result to FlowVisor. A similar conflict interception system is applied in [138], but that system models the network rules using first-order logic to detect rule conflicts.

Although these solutions can efficiently identify rule conflicts, they do not consider the dependencies between rules and the network state, which may lead to false positives. Wang *et al.* [139] modeled OpenFlow rules in the form of a network topology consisting of the flow paths formed by the NetPlumb graph [19] and checked for any intersections between the flow path space and the Deny Authorization Space defined by the firewall policy. The identified intersections represent policy conflicts and are regarded as bypass threats to the SDN firewall. This work was later extended in the development of FLOWGUARD [140] to provide a tool for comprehensive dynamic conflict verification. With the support of more header fields, FLOWGUARD monitors all control messages to check space intersections to achieve the dynamic detection of firewall policy violations.

SDNRacer [74], [141] concerns concurrency violations in controller operations on the flow tables in switches. It leverages the first happens-before (HB) model to formulate the behaviors of networks, including operations on flow tables (i.e., the reading, addition, modification and deletion of flow entries) and the behaviors of network elements (e.g., OpenFlow switches, controllers, and hosts). A commutativity specification is defined in SDNRacer, which can be used in combination with the HB model to check whether two operations commute. However, even short traces can yield an excessive number of concurrency violations, which poses a challenge for the concurrency analysis. BigBug [142] addresses this issue by exploiting the characteristic that many violations originate from the same cause. It clusters the input set of violations into ECs and identifies the most representative violation in each class using a ranking function. Based on the results of BigBug, developers can quickly focus on understanding the root causes of the most representative violations.

Although these conflict verification approaches are efficient at finding rule conflicts that may lead to network failures, they are reactive in nature and thus cannot prevent such conflicts. We will discuss several solutions that can prevent rule/policy conflicts and endow SDN networks with fault tolerance in Section VII-A.

2) Translation Verification: Since apps written with common programming languages are prone to error, many high-level programming languages have been proposed to simplify the programming of apps. These languages are often designed for specific domains, and a compiler or parser (e.g., NetCore [162]) is needed to translate programs written in these languages into low-level flow rule commands. To ensure the correctness of program translation, the first step is to verify the correctness of these compilers, namely, *translation verification*.

Guha *et al.* [91] investigated the translation verification issue for the NetCore compiler [162] on the basis of properties such as controller-switch consistency, the correctness of barrier messages for message reordering, and the correctness of translation patterns. They presented a verified SDN controller based on the Coq proof assistant (an interactive theorem prover) for the NetCore programming language, which is a declarative language. In this verified controller, NetCore programs are first translated into *flow tables*, which abstract the behaviors of switches, and are then further translated into *featherweight OpenFlow*, which models the OpenFlow switches, the controller and the network topology in the form of operational semantics. The verified controller then leverages the Coq proof assistant to prove the correctness of the program translation by means of a formal specification and a detailed SDN operational model. This approach can provide the most basic guarantee that programs written in high-level languages can be translated correctly.

TABLE IX
FAULT DIAGNOSIS FOR CONTROL PLANE

Reference	Short Description	Issues Addressed	Proposed Solution
FortNox [89]	A security enforcement kernel for rule conflict verification.	Rule conflicts.	Abstract flow rules to be compared with the defined security policies.
FLOVER [137]	SAT-based rule conflict verification.	Rule conflicts.	Use the Yices SMT solver to verify newly generated rules.
Natarajan *et al.* [90]	A flow-table-level rule conflict detection and resolution mechanism.	Rule conflicts.	Provide a conflict search mechanism based on a hash-trie structure and an ontology-based logic inference system.
[138]	A rule conflict verification mechanism based on first-order logic.	Rule conflicts.	Model flow rules using first-order logic.
FLOWGUARD [140]	A dynamic conflict verification mechanism.	Rule conflicts.	Use the NetPlumb graph to formulate network rules.
SDNRacer [74], [141]	A dynamic concurrency analyzer.	Concurrency violations in flow tables.	Model network operations using the first happens-before model to find conflicts.
BigBug [142]	A dynamic concurrency analyzer.	Concurrency violations in flow tables.	Slice the network model into ECs and identify the most representative violation per class by ranking.
Controller verifier [91]	A verified SDN controller.	Policy mistranslation.	Use a theorem prover to verify the correctness of program translations.
OFRewind [83]	A record-and-replay-based troubleshooter for OpenFlow networks.	Abnormal events and policy violations.	Record OpenFlow messages and replay the traces in the DP to localize the root causes of faults.
STS [16]	A black-box troubleshooting tool for automatically identifying a minimal sequence of inputs responsible for triggering a given bug in the CP.	Policy violations, faulty components, inconsistent controller states and coding mistakes.	Design a simulator for replaying input events and a delta-debugging-based replay mechanism for pruning input sequences to generate a minimal causal sequence of inputs triggering a given bug.
JURY [17]	A black-box testing tool for verifying the action consistency among the SDN controllers in a cluster.	Faulty components and inconsistent controller states.	Validate differences in behavior among the controllers in a cluster by injecting the same input events and comparing the results with the defined correct properties.

3) Software-Based Debugging: SDN controllers are software entities that are highly susceptible to software bugs. Unfortunately, traditional program debugging techniques (e.g., breakpoints, assertion, and logging) are insufficient for these SDN software systems since they are tightly correlated with complicated network states and are modular and physically distributed to guarantee their scalability and reliability. To address this issue, many advanced *software-based debugging tools* (e.g., record and replay and delta debugging) have been proposed for controller software; these tools are built on traditional debugging solutions but possess more advanced modifications to improve their feasibility.

OFRewind [83] is a tool that can record and replay network events to localize the causes of network failures in OpenFlow networks. It acts as a proxy between the CP and DP, recording OpenFlow messages and re-injecting the traces into the DP to identify which events trigger a failure. OFRewind also provides interfaces for specifying the topology, timeline and specific traffic to allow operators to implement their desired debugging queries. OFRewind can help network operators to find issues in controller software, e.g., configuration issues and invalid actions. AFRO [85] also implements a similar record-and-reply mechanism to determine whether there are missing flow rules in the DP. AFRO records all *Packet_In* messages in real time and spawns a new controller instance in an emulated environment to replay the network state by feeding in *Packet_In* messages. This replay procedure enables the computation of a minimal set of rule changes between the emulated and current forwarding states. Finally, these different rule sets are utilized to reconfigure the failed network; this process will be further discussed in Section VI.

Scott *et al.* [16], [143] proposed a delta-debugging-based troubleshooting tool for invariant violation problems caused by improper network configurations and software bugs in controller software, e.g., multicontroller coordination errors, null pointers, race conditions, memory leakage and corruption. This tool leverages an HSA checker [19] to find any invariant violations in the network at run time. When faced with symptoms of a network problem, it uses delta debugging to iteratively select and replay subsequences from the collected network event sequence to reproduce the observed failure by means of a network simulator, namely, STS, which can generate random input sequences based on the causal relations among events from collected event sequences and correctly replay network behaviors with SDN controllers using these sequences. The purpose of this tool is to find a minimal causal sequence (MCS) of the recorded events that can be used to reproduce the observed violation with the minimal cost, thereby answering the questions of "what, where, and when" with regard to a fault in controller software.

4) Testing and Evaluation: The SDN CP plays a crucial role in an SDN system. The *correctness* and *performance* of the CP are the key factors in ensuring that it can be used to manage various types of networks of different scales. Testing controllers to find performance bottlenecks or faulty functionalities is essential to ensure that the controllers can satisfy the requirements of actual networks. In this subsection, we discuss recent research on the *testing and evaluation* of the correctness and performance of SDN CPs.

Cbench[7] is a benchmarking tool for testing OpenFlow controllers. It emulates numerous switches that connect to a controller, generate and send Packet_In messages, and watch for Flow_Mods to be pushed down. Cbench is a very useful tool for evaluating the performance of OpenFlow controllers in terms of their OpenFlow message throughput and latency. NOX-MT [145] leverages Cbench to quantify the performance of NOX. OFCbenchmark [146] is an OpenFlow controller benchmark that can create a set of virtual switches for generating OpenFlow messages and can analyze the controller responses to these generated messages. Based on this tool, the authors also designed a platform-independent testing framework, FCProbe [148], for analyzing the correctness of

[7]Cbench - https://github.com/mininet/oflops/tree/master/cbench.

controller behaviors. SDLoad [147] is a workload testing tool for SDN controllers that can add workloads to evaluate the correctness and performance of SDN CP components.

Instead of testing a single SDN controller, JURY [17] addresses consistency issues in a controller cluster and serves as a black-box testing tool for verifying the action consistency among controllers. It intercepts and replicates external events (e.g., Packet_In messages) and internal events (e.g., distributed database operations, state synchronization, and master elections) between primary and secondary controllers, and it collects the generated responses from these controllers. Then, JURY maps all controller responses to the events and transmits this information to an out-of-band validator to validate the correctness of the controller actions. JURY can detect various controller faults, including cluster faults (e.g., database locking and incorrect master elections) and incorrect event processing faults (e.g., Flow_Mod drops and incorrect Flow_Mod messages).

C. Fault Diagnosis for Applications

Bugs in software programs are inevitable and can become more serious when these programs are executed in the presence of complicated network states. The simple syntax debuggers used in software engineering can play only a small role in debugging programs. Ensuring the correctness of the logic for the underlying networks requires programs to be analyzed in depth. In this subsection, we discuss how to diagnose faults in apps. The approaches for app fault diagnosis are summarized with respect to their goals and proposed solutions in Table X.

1) White-Box Verification: Formal verification techniques can also be applied to debug SDN apps by analyzing both the network state and the program logic as extracted from the source codes of the apps or as defined by the programmers. In [150], two model checkers (Java Pathfinder and SPIN) were applied to reveal bugs in OpenFlow apps using code-based network models, and SPIN was shown to be faster than Java Pathfinder. However, this work was simple and did not consider the inherent state-space explosion issue that arises in model checking since it is often necessary to perform an exhaustive search of the state space of the graph to determine whether the specifications hold in all states or whether counterexamples can be provided.

Canini *et al.* [151] addressed this problem using *symbolic execution*. In this work, packet code paths were clustered by analyzing the source codes of OpenFlow apps via symbolic execution, and inputs were automatically generated to identify errors in OpenFlow apps based on the desired correctness properties. Building on this work, the authors subsequently proposed NICE [23], which combines model checking and symbolic execution to reduce the state space; here, model checking is applied to check for correctness violations in system state propagation, and symbolic execution is used to reduce the size of the searched state space. In NICE, each event handler is treated as a transition, with several inputs for network events and global variables for program states. NICE uses a simple variable to represent the header field of a packet to determine the packet's path through the handler. NICE also models OpenFlow switches and simple host services to generate and forward packets in the program model. Upon encountering an abnormal network state, NICE will apply model checking to explore the state space of the entire system to find the root cause.

Majumdar *et al.* [163] modeled control programs and the network topology (i.e., switches, links and hosts) with a custom programming language, and a model checker built on this language was used to verify whether the programs satisfied a given safety property. To reduce the state space explored by the model checker, they used partial order reduction and abstraction techniques to optimize the behaviors of switches, clients, packets, and controllers. Compared with NICE, this approach offers model checking with improved scalability and coverage.

Extracting models from low-level languages is a time-consuming and error-prone process, and its scalability is often limited. Many high-level language-based solutions have been proposed to handle this issue by providing domain-specific programming languages for modeling [45], [46], [152], [153]. In [152], model checking was applied to verify two SDN programs (a MAC learning switch and a stateful firewall) in large-scale networks. Verificare [45] predefines system components to provide Verificare Modeling Language (VML) APIs, which enable app developers to model their programs with VML and then to use various verification tools (such as SPIN, PRISM, and Alloy) to verify their correctness. VeriCon [46] was developed to provide a sound tool based on infinite-state models for verifying apps. By means of a domain-specific programming language, it converts programs into first-order formulas that specify constraints on the topology and desired properties, including topology invariants, safety invariants and transition invariants. Using a theorem prover and a SAT solver (Z3), VeriCon can prove whether an invariant is inductive through the execution of arbitrary events on any admissible state; otherwise, a readable counterexample will be presented for the observed error. By this means, the correctness of apps can be verified on any admissible topology and for any possible sequence of network events.

Beckett *et al.* [154] found that network properties (or safety invariants) may vary dynamically, which can cause the static verification approaches discussed above to fail. They designed an assertion language (AL) for apps that can annotate control programs with C-style assertions about the DP to support dynamically changing verification conditions. By means of statements in response to these assertions, AL incrementally updates the properties as the verification conditions change and uses VeriFlow [20] to check for bugs in apps.

To provide more debugging functions (e.g., stepping, breakpoints, and watch variables) rather than simply verifying apps, *OFf* [97] was designed as a debugging and testing environment for SDN apps that is built on top of the *fs-sdn* simulator [164]. In contrast to `ndb` [106] and NetSight [107], *OFf* traces both the network and program execution states to identify network failures. It also includes a language-level debugger for basic language debugging functions, a component for trace replay and a verification tool for variation validation.

To fix bugs, update programs or restructure features, network programmers need to evolve their programs; however,

TABLE X
PROPERTY ANALYSIS FOR APPS

Reference	Short Description	Goal	Proposed Solution
Perešíni *et al.* [150]	App verification using two model checkers.	Check the correctness of properties for the network states.	Apply JPF and SPIN to model OpenFlow apps.
NICE [23]	Test OpenFlow apps using a model checker and symbolic execution.	Find policy violations.	Identify network transitions by modeling OpenFlow apps and network elements and explore all possible transitions via symbolic execution and model checking to uncover bugs based on the correct properties and the network topology.
Kuai [163]	Find property violations using a model checker and partial order reduction.	Check for policy flaws and policy violations and reduce the state space for verification.	Model control programs and the network topology with a custom language and leverage partial order reduction and abstraction techniques to implement finite-state model checking.
Sethi *et al.* [152]	Abstraction-based model checking for OpenFlow apps.	Verification of property violations in a large network topology.	Analyze apps via model checking and reduce the sizes of the program model and network state abstraction to support the processing of an arbitrarily large number of packets.
Verificare [45]	App verification based on a domain-specific modeling language (VML).	Check for violations of QoS and safety properties with powerful expressions.	Model apps with VML, which provides a predefined set of components, and verify the app properties with a variety of formal verification tools.
VeriCon [46]	App verification using a theorem prover and SAT solver.	Verify policy flaws and program implementations in arbitary network topologies.	Use first-order logic to specify network topologies and desired invariants and implement app verification using Z3.
Chimp [153]	Static differential analysis of the evolution of controller software.	Verify the correctness of revised programs.	Compare two versions of a control program to extract their semantic or behavioral differences using Alloy.
Assertion language [154]	Incremental verification based on an assertion language for SDN apps.	Find property violations by dynamically changing execution conditions.	Annotate apps with C-style assertions to describe time-varying properties and verify dynamic properties with an incremental data structure.
OFf [97]	A simulation-based debugging and test environment for app development.	Debug policy flaws and incorrect implementations in apps.	Build on top of the *fs-sdn* simulator to trace program executions and network states and debug apps with various debugging features.
Yao *et al.* [88]	A model-based black-box testing for SDN apps.	Uncover implementation bugs and policy flaws in apps.	Use a set of component models to model apps and generate test sequences to be sent to the DP based on these models to uncover bugs in apps.
SIMON [98]	An interactive debugger and monitor approach for SDN apps.	Find policy flaws and implementation bugs in apps.	Capture network events (e.g., NBI&SBI messages, traffic and DP events) and represent abnormal network behaviors along with the related event executions in a time line at the debugging prompt.

techniques for tasks such as verification and testing may be invalid for newly added features if the existing properties or tests have not been updated as the program has evolved. Nelson *et al.* [153] focused on this concern and proposed a verification tool (Chimp) based on differential analysis for the evolution of control software. Chimp is built on Alloy (a lightweight formal modeling and verification tool), and its objective is to find semantic or behavioral differences between two versions of a control program and help programmers to transfer trust between two versions. Chimp can also find bugs in a revised program based on its differential properties using formal methods. This tool plays a role complementary to that of the aforementioned SDN app verification tools, and the issue it addresses deserves greater attention.

2) Black-Box Testing: The white-box approaches discussed above can enable efficient debugging of programs under development. However, they sometimes suffer from poor efficiency (due to close-source apps) or high time consumption (due to state-space explosion); in addition, many apps contain multiple components with various statements that are not easy to exhaustively explore. Yao *et al.* [88] proposed a model-based black-box testing method for SDN apps that does not require their source codes. In this model, the software behaviors of an app are represented by a group of parallel component models (e.g., packet handlers and entry components). Then, by generating test sequences based on the partial states of related components, DP sequences are recorded that can then be used for SDN network simulation traffic to expose both design flaws and implementation bugs.

SIMON [98] is an interactive debugger and monitor for OpenFlow apps that allows network operators to probe network behaviors with custom scripts to find both implementation errors and policy violations in apps. In SIMON, several monitors are established alongside an SDN system to catch network events such as northbound API messages, OF messages, network traffic, and DP events. With SIMON, operators can query abnormal network behaviors without having intimate knowledge of the controller software, and SIMON can extract related events and display their execution in a time line at its debugging prompt.

D. Summary

In this section, we have presented detailed descriptions of fault diagnosis solutions in the SDN domain for the sake of classifying and comparing these different solutions. Most solutions consider the problem of fault diagnosis on only one plane in the SDN stack. While quite powerful, they may also lack comprehensiveness. Several solutions have been proposed as systemic fault diagnosis frameworks for SDN networks. Heller *et al.* [92] proposed a systemic troubleshooting framework for SDN that combines several troubleshooting tools working together. The SDN network is divided into two types of layers: code layers (apps, NetHypervisor, NetOS, and firmware) and state layers (policies, logical view, physical view, device states and hardware). For troubleshooting, binary searches are first performed to reduce the scope of the problem to one code layer through cross-layer correspondence checking of the state layers; then, the fault is diagnosed in the identified code layer with existing tools. To provide more flexible diagnosis functions, Pelle *et al.* [155] defined a lightweight framework that combines existing network and software troubleshooting tools for specific troubleshooting configurations. These tools are combined in the form of troubleshooting graphs, which represent practical troubleshooting patterns. EPOXIDE [156] was developed as an extension of this work

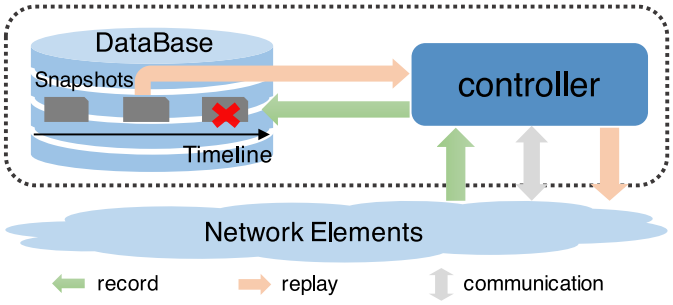


Fig. 3. State rollback in SDN. In the record stage, the controller states and switch states are collected and stored as snapshots in the database. When one component fails, the controller and switch states will be recovered from a previous correct snapshot.

to provide support for the ad hoc creation of tailor-made testing methods from predefined building blocks. These tools represent a preliminary exploration of the development of systematic SDN fault diagnosis systems, and they deserve greater attention.

VI. Fault Recovery and Repair

As described in Section III, there are a number of different faults that can cause a network to fail to provide a desired level of service. While fault diagnosis techniques are the most critical techniques for network maintenance, developing techniques to support recovery from error states is also important for improving network reliability and availability. We discuss fault recovery in SDN networks from two perspectives: state rollback (Section VI-A) and configuration fault repair (Section VI-B).

A. State Rollback

State rollback is a typical recovery approach that is widely used in distributed systems. As shown in Fig. 3, the main concept behind rollback is to periodically record the state of the system and store it as a snapshot with a timestamp in data storage or elsewhere. When a failure occurs, a previous correct snapshot will be chosen to which to roll the current faulty system back. The rollback process is similar to the record-and-replay mechanism for fault diagnosis discussed in Section V-B3, but here, the purpose of the replay is the recovery of a faulty system. This method can mitigate the impact of faults, especially for those faults that are difficult to repair [85], [165], [166].

NetRevert [165] is a checkpointing and rollback framework for SDN fault recovery. It provides a distributed approach to collecting system states in which each device (controllers and hardware/software switches) is responsible for checkpointing its own state independently. Each state change in a switch is tagged with a transaction identifier (ID) defined by the controller and stored as a snapshot ID in that switch. The controller is responsible for collecting all snapshots from the switches and selecting a set of network-wide consistent states to roll the whole system back to a globally consistent state at the time of recovery. Although the distributed approach to checkpointing can enable memory load balancing and high scalability for snapshot storage, it may also face issues of data synchronization and high computational overheads in highly dynamic networks. Sasaki *et al.* [166] also leveraged the rollback mechanism to revert compromised processes in network components (e.g., switches, controllers, channels, and the hypervisor) to pristine states. However, the primary goal of that work was to minimize the impact of an attack on the system as a whole.

In addition to recovering the CP and DP, LegoSDN [31] leverages rollback to address the recovery problem for crashed SDN apps. To mitigate the impact of faulty apps, LegoSDN relies on a fault-tolerant controller framework, in which each app or the controller is running separately in a sandbox. Thus, it can limit the cost of component recovery without influencing other apps or the controller. LegoSDN continuously records input events and their corresponding output messages from SDN apps as the snapshots or checkpoints that are used to ensure the consistency among different apps and the controller. When an app has crashed while processing some events, LegoSDN rolls back the changes made on the CP and DP by the crashing app and restores the app. Before allowing the crashed app to re-access the controller, LegoSDN leverages an event transformer to replay predefined events (different but equivalent to the event triggering the crash) to find an event with which the app can successfully realize its function. LegoSDN enables recovery from two types of faults: *fail-stops* due to invalid memory accesses or erroneous expressions and *invariant violations*.

B. Faulty Configuration Repair

Although rolling abnormal networks back to a previous correct state has been widely adopted for network recovery, it is often a time-consuming process and can incur high resource overheads for storing snapshots. Benefiting from the programmability of SDN, *faulty configuration repair* is also an efficient approach for recovering from abnormal network states. In this approach, suitable rule sets (i.e., repairs) are sought to reconfigure the affected switches. Traditionally, misconfiguration issues must be manually fixed by network administers, which is a tedious and error-prone process. The innovation of SDN provides opportunities to implement automated fault repair. The following discussion focuses on several proposed repair approaches for misconfiguration issues.

Incorrect forwarding rules are the major causes of abnormal network states (e.g., forwarding loops and blackholes) in OpenFlow networks. To address this problem, one simple approach is to delete the problematic forwarding rules. However, since a controller is typically responsible for several apps, each with its own policies, whose rules may potentially overlap with each other, deleting rules for the individual logic of one app is inefficient and may induce additional configuration issues. By separating the failure recovery mechanism from app-specific functionality, a run-time system for automatic failure recovery called AFRO [85] has been proposed. AFRO consists of three phases: record, replay and reconfiguration. In the record phase, AFRO keeps track of all Packet_In messages. When a failure is detected, AFRO first spawns a

TABLE XI
RELATED WORK ON FAULT RECOVERY AND REPAIR

Type	Reference	Goal	Proposed Solution
Rollback	NetRevert [165]	Recover an SDN system from failures, software bugs, or misconfigurations.	Set checkpoints in both switches and controllers to construct network-wide snapshots and roll the SDN system back to a pristine state when a failure occurs.
	Sasaki *et al.* [166]	Recover compromised components.	Design a secure SDN architecture that can revert the compromised processes in failed network components via rollback.
	LegoSDN [31]	Make SDN controllers tolerant to app failures (e.g., fail-stops and invariant violations).	Set checkpoints in apps to capture their transactions using OF messages as snapshots, roll back changes (made by the crashing app), and continuously transform events with a replay technique to find a safe sequence of events that will not cause the app to crash.
Repair	AFRO [85]	Automatically recover a network with a different set of rules.	Log OpenFlow messages, replay these messages in an emulated environment to find the rule inconsistencies between the emulated and actual forwarding states, and modify and recover the switches based on with the rule set.
	Marham [86]	Fix buggy configurations.	Provide an optimized SMT solver for finding a repaired set of network configurations with the objective of minimizing the number of switches that need to be modified.
	Wu *et al.* [87], [99]	Automatically generate repairs for fixing bugs in control programs.	Use the negative provenance to find the root causes of faults and infer possible suitable repairs and then test these repairs based on historical network information to deduce the smallest and plausible list of fixes with the minimal effect on the network.
	NEAt [100]	Automatically repair policy violations in real time.	Leverage Veriflow to verify rule updates and search for the minimum number of changes to the violating ECs forwarding graph via a clustering algorithm.

new controller instance in an emulated environment to replay the network state by feeding it the recorded messages. The replay results can be used to compute a minimal set of rule changes between the current and emulated forwarding states. By installing this rule set, AFRO can recover the network from its abnormal state.

For improved efficiency in repair generation, formal verification techniques can also be used to recover from network misconfigurations [86], [100]. In contrast to the configuration verification problem discussed in Section V, the network repair problem is as follows: given a set of network configurations (forwarding rules) that violate an invariant, a repaired set of configurations is sought such that the repair is optimal with respect to a given objective (e.g., a minimal scope of repair). Hojjat *et al.* [86] formulated the repair problem for SDN networks as a verification problem and solved this problem by designing an optimized SMT solver in which faulty network configurations are translated into a set of Horn clauses for checking the violated invariant. NEAt is a network repair solution that can automatically diagnose and repair violations in real time. It leverages Veriflow [20] to find violations. When a violation occurs, NEAt slices the network configuration into a set of ECs and computes the minimum number of changes necessary to repair the violating EC's forwarding graph.

In SDN networks, the root cause of a network misconfiguration is often a policy flaw in SDN apps. Although the above solutions can perform automated network repair generation, they are inefficient since they simply repair the faulty configurations in the DP. They cannot repair the bugs in the SDN apps that are responsible for these network faults, which will consequently arise again in the future. To address this issue, Wu *et al.* [87], [99] designed a tool that can produce a list of suggested program patches (repairs) for fixing identified faults in SDN apps. They had leveraged data provenance backtracing in their previous studies [132], [133], as discussed in Section V-A4, and they extended this work to model both control programs (written in NDlog, a declarative language) and data, based on a concept they called the meta provenance. By applying backtracing to the meta provenance graph, one can find which node (representing a network event, i.e., a rule and its related operations) in the graph induces a given fault and changes the node state in order to infer candidate repairs for the fault. To further reduce the side effects of a candidate repair, their tool backtests each candidate via replay using historical data from the network to narrow the set of suitable repairs suggested for fixing bugs in SDN apps.

C. Summary

The aforementioned recovery and repair solutions are summarized in Table XI. They represent preliminary attempts to endow networks with self-healing functions by leveraging the benefits of SDN, such as centralized management and network programmability. Fault recovery and repair constitute an indispensable part of fault management for guaranteeing SDN network reliability, but further research efforts are still required to improve the current fault recovery and repair capabilities in SDN.

VII. FAULT TOLERANCE

Fault diagnosis and recovery techniques have been identified above. By contrast, this section discusses fault tolerance techniques, which aim to reduce or avoid the effects of faults on SDN networks. Conflict resolution is addressed in Section VII-A, fault tolerance for traffic is considered in Section VII-B, and infrastructure planning is reviewed in Section VII-C.

A. Conflict Resolution

This section focuses on how to resolve *policy conflicts* over flow rules or high-level network resources among different apps in SDN networks. We categorize these solutions into three types: (1) *operation isolation* (Section VII-A1), in which virtualization techniques are leveraged to isolate each tenant's operations; (2) *policy composition* (Section VII-A2), which aims to combine multiple independent policies into a large policy; and (3) *module coordination* (Section VII-A3), in which a coordinator is deployed to reconcile resource competition among different SDN apps.

1) Operation Isolation: Networks often support multi-tenancy scenarios, in which multiple tenants operate on shared network resources. To support this scenario, network virtualization promises to be an effective method of resolving conflicts among different tenants while ensuring the correctness

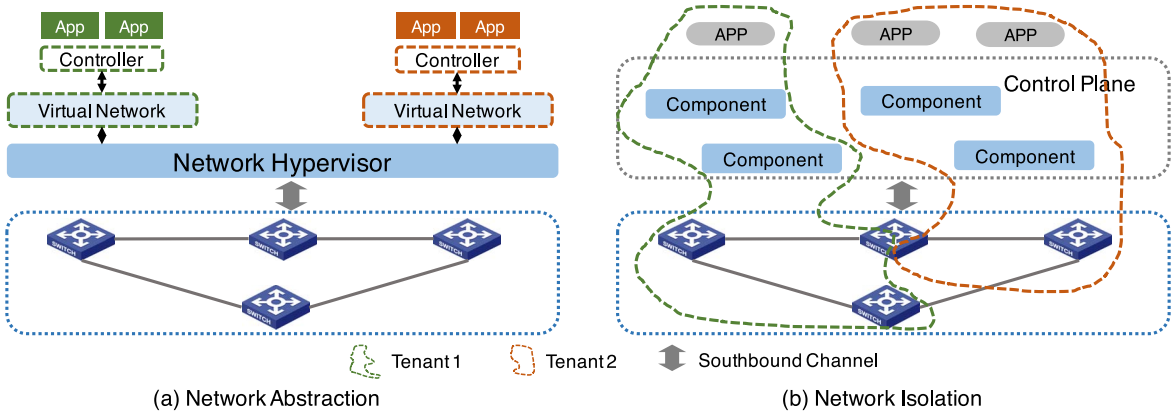


Fig. 4. Two types of network virtualization in SDN. Whereas *abstraction* attempts to abstract the details of the underlying network to allow tenants to operate on a virtual network, *isolation* attempts to isolate tenants' operations in both the CP and DP to avoid operation conflicts.

of each tenant's operations on the network. As shown in Fig. 4, network virtualization can provide two main functions, namely, *abstraction* and *isolation*. Whereas *abstraction* attempts to abstract the details of the underlying network to allow tenants to operate on an abstraction layer, *isolation* attempts to isolate tenants' operations in both the CP and DP to avoid operation conflicts. From low-level virtualization mechanisms (e.g., VxLAN, NvGRE, and STT) to high-level hypervisors acting on the SDN CP, a number of network virtualization techniques have been developed. In this subsection, we discuss these virtualization-based operation isolation techniques in terms of how they can be used to prevent policy conflicts to support multi-tenancy scenarios for SDN. Note that here, we simply select a few typical network virtualization techniques to illustrate how they support fault tolerance; a more detailed discussion of network virtualization can be found in a previous survey paper [167].

FlowVisor [161] was the first network virtualization technique proposed for OpenFlow networks. FlowVisor divides a physical network into a set of virtual networks, each being called a *slice*. To isolate each slice, FlowVisor acts as a network virtualization layer (more commonly called a hypervisor) interposed between the CP and DP to provide control isolation among network resources (e.g., link bandwidth, topology and switch CPUs), flow tables and OpenFlow control channels to control the access of different tenants' controllers to the switches. Slicing provides isolation to allow multiple potentially competing logical networks to share the same network resources, although rule-level conflicts on the DP are still not effectively resolved due to poor rule conflict detection, as discussed in Section V-B1.

Koponen *et al.* [26] implemented a network hypervisor (NVP) with the goal of serving a complementary role when applied together with mature computing hypervisors in modern multi-tenant data centers. Similar to computing hypervisors, it provides *abstractions* of the network resources using logical data paths in an overlay network implemented with OpenvSwitch (OVS) to allow for the creation, configuration and management of independent overlay networks for multiple tenants. The logical data paths have the same configuration models as the physical data paths and can be automatically installed into associated OVSs and connected to the physical network through network tunnels by NVP. By means of such an overlay network, packet forwarding can be implemented in the logical network without any changes to the physical network among the servers, which simply needs to ensure the connectivity among servers.

However, the two virtualization tools discussed above require the interposition of a hypervisor between the controller and the switches, which may introduce latency and errors. Splendid isolation [168] implements a language-level virtualization whereby network slices are directly specified by a programming language. Splendid isolation defines a network slice as consisting of three ingredients: a *topology*, a *mapping* (the topological relationship between the slice and the underlying network) and *predicates* (one for each port of the edge switches in the slice; these predicates specify access permissions for packets). Unlike a network hypervisor, Splendid isolation imposes a static isolation that allows different programs to be associated with different slices, which will then be translated into a global configuration for the whole network. By providing such a slice abstraction, it can simplify the implementation of traffic isolation and support multiple control programs without harmful interference. In addition, the high verifiability of language abstractions has come to be recognized as an advantage of slice abstraction because it allows a specific isolation implementation to be verified with formal verification tools.

2) Policy Composition: Although virtualization-based isolation techniques can enable tenant isolation based on network slices, they cannot address conflicts among multiple apps processing the same traffic, which may arise even in the single-tenant scenario. In fact, such hypervisors may be invalid in this situation since they can provide only slice-level isolation. Policy composition offers a different method of resolving conflicts, in which policies from different apps with various purposes are combined into one large policy.

To compose policies, three composition operators have been developed: *parallel* ($+$) [24], *sequential* ($\gg$) [48] and *override* ($\succ$) [169], [170], as shown in Fig. 5. Here, we use an example given in [169] and [170] to introduce these three composition operators. This example consists of three modules, namely,

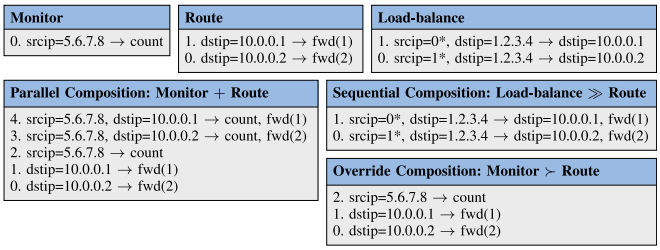


Fig. 5. Parallel, sequential and override composition [48], [169].

(*Monitor*, *Route* and *Load Balance*), each with different policies. Through parallel composition, network programmers can set different policies to be performed simultaneously, as indicated by the operator "+"; in this case, the overlapping match fields of the different rules are combined, and their actions are concatenated together. The override operator "≻" can also be used to combine conflicting policies, but it overrides the priority settings of the rules such that one module is always valid. For example, override composition can be used to specify that incoming packets should be processed by *Route* only when *Monitor* has failed, as shown in Fig. 5. As an alternative to the direct removal of overlapping, some modules can be combined sequentially in an order specified by the programmer with the sequential operator "≫", which indicates that incoming packets will be processed by the first policy (e.g., *Load Balance)* and that the outputs will be then processed by the second policy (e.g., *Route*).

These operators have been widely implemented in various systems. Frenetic [24] and Pyretic [48] are domain-specific programming languages that provide a language-level hypervisor (more commonly called a compiler) that translates policies into OpenFlow rules for programming networks with composition operators. These composition operators are also applied in NetKat [171] to combine arbitrary forwarding policies with access control (ACL). Override composition was proposed in [169] and [170]. STN [169] is a distributed SDN CP that extends Pyretic to include the override operator and aims to solve conflicts among concurrent policy updates. RuleTris [170], [172] has been proposed to eliminate unnecessary priority updates in the high-level program compiler and supports policy composition.

In contrast to the Frenetic hypervisor, FlowBricks [173] and Compositional Hypervisor [174] provide an imperative interface with parallel and sequential composition options to allow all controllers (each consisting of individual apps) to directly process standard OpenFlow messages and generate policies, which are then compiled into a single policy. This approach can ensure the scalability of such policies for application in various SDN controllers. Compositional Hypervisor [174] has been extended to CoVisor [175], which offers performance improvements in terms of rule composition, e.g., the introduction of override composition. Both Compositional Hypervisor and CoVisor can allow any controller to update the network during run time by implementing the incremental composition of rules from different controllers based on the recalculation and rewriting of rule priorities. In addition to policy composition, additional fault tolerance mechanisms for controller failures or switch failures are also implemented in CoVisor [175], whereby the administrator can define a default app-dependent policy for each controller to execute corresponding operations when controller failure occurs. To address switch failures, CoVisor can remove all its rules and notify the relevant controllers.

The aforementioned policy composition solutions are efficient; however, they focus only on the composition of match fields through simple concatenations of actions, which may result in incorrect behaviors (in the case of parallel composition) or inefficient compositions (in the case of sequential composition) [176]. For example, if the two rules `{push_vlan(1), tcpdst=80 → fwd(1)}` and `{dstip=10.0.0.1, tcpdst=80 → fwd(2)}` are combined in parallel, the resulting rule is `{push_vlan(1), tcpdst=80 → fwd(1), dstip=10.0.0.1, tcpdst=80 → fwd(2)}`, which can forward packets with the appropriately modified IP destination address or an added VLAN header to port 2. To address this issue, Pan *et al.* [176] modeled the process of packet construction as a graph in which the vertices and edges represent packets and transformations between packets defined by actions, respectively, and they generated the correct actions for policy composition by searching for a Hamiltonian path in this graph.

PGA [49] is an approach that supports the automatic composition of independent network policies rather than the manual composition supported by the aforementioned approaches. PGA uses a graph-based abstraction to allow network programmers to specify their policies in the form of directed graphs in which each vertex represents an endpoint group (EPG) sharing common properties. PGA then decomposes these EPGs into a set of disjoint EPGs to identify the overlapping space and automatically recomposes them into a coherent composed policy (a conflict-free graph) based on verification with composition constraints. PGA can resolve, or flag, conflicts/errors based on the defined composition constraints and report them to users, possibly with suggested fixes.

3) Module Coordination: For conflict resolution, module coordination is also an important means of reducing the effect of policy conflicts. The main idea of this approach is to coordinate any conflicting operations on flow rules or coarser network resources (e.g., link bandwidth or access control) and reassign a different priority to each operation based on specific mechanisms to resolve conflicts.

PANE [177] attempts to provide participatory networks with a high-level configuration API whereby SDN apps can autonomously and dynamically invoke network resources without worrying about conflicts. It incrementally models the operations (e.g., requests, queries and hints) of authorized apps by adding their privileges and related flows into a shared tree. Thus, PANE can constrain the network policies constructed from the policy tree. An incoming request is first checked against the shared tree for admission and is then checked against the policy tree and the physical capabilities of the network using hierarchical flow tables to resolve conflicts. In this checking pass, the request is incorporated into the two trees, and OpenFlow rules are installed into the network.

Voting mechanisms for resolving conflicts between different SDN control modules have been proposed in Corybantic [69] and Athens [178]. In Corybantic, SDN modules first propose some topology changes (allocations or placements of network resources, called proposals), and then, each module evaluates every current proposal to assign a value to it that reflects any costs created by unfairness. Then, a global coordinator chooses the best proposal, and any module affected by the chosen proposal can materialize it directly. However, this approach places strict demands on each module's capability to evaluate proposals. To overcome this issue, Athens performs a family of evaluations of proposals instead of the particular evaluations performed in Corybantic. Upon receiving a new external request, Athens collects the proposals generated by each controller module, coordinates the family of voting evaluations in each module, and chooses the winning proposal to be implemented. These approaches are useful but can impose high overheads and code changes on SDN modules.

Instead of performing explicit coordination, as in the above two approaches, Statesman [25] achieves a different means of dynamically resolving conflicts by loosening the coupling between the modules and the system. It functions as an arbitration system between the apps and the SDN controllers to prevent conflicts and invariant violations. Statesman examines the applicability of state changes proposed by apps. Then, it merges all proposed changes into one target change by detecting conflicts among them (considering the dependencies on the network states) and resolving these conflicts with a last-writer-wins or priority-based locking mechanism. In addition to resolving conflicts, Statesman also checks for invariant violations by comparing the observed and target states against a network graph. This approach is beneficial for resolving conflicts at the network resource level. However, when a new app emerges, it may necessary to modify or even rewrite the apps or their coordination system.

Volpano *et al.* [93] addressed the issue of resource conflict detection and resolution using formal verification techniques. In their approach, each network control function acting on network resources is modeled as an FSM. Based on these FSMs, the intersections between controller functions can be identified to generate a combination of machines that can be deployed in the DP without conflicts. This approach enables the materialization of multiple proposals simultaneously rather than only the winning proposal, as in Corybantic and Athens, or of proposals in a specific order, as in Statesman.

4) Summary: Three different conflict resolution approaches have been discussed in this subsection. We summarize these techniques in Table XII. Network virtualization techniques can allow multiple tenants to share the same network infrastructure while also providing failure isolation among these tenants. Policy composition operators are commonly implemented in high-level programming languages for SDN, although most of them require manual decisions by programmers, which is an error-prone process. Module coordination is an indispensable function for SDN controllers that allows multiple apps to coexist even when there are potential conflicts among them. These three approaches can be used to resolve conflicts among tenants or apps to different extents, but all are important for the evolution of an open SDN ecosystem.

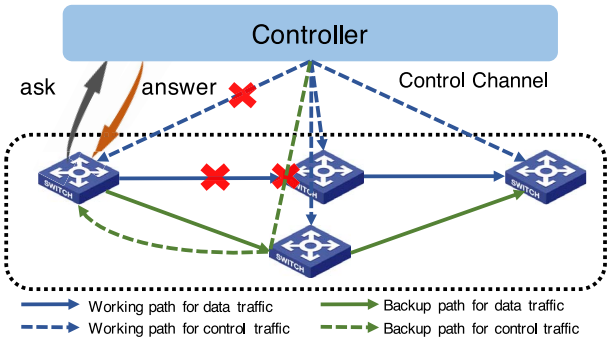


Fig. 6. Failover for the DP. *Restoration*: When there is a link failure, the switch will ask the controller how to forward the packet. *Protection*: Backup paths are preconfigured in the switch for use in the event of link failure without the need to consult the controller.

B. Fault Tolerance for Traffic

To ensure network service continuation, providing fault tolerance against traffic failures is essential. Network traffic in SDN-enabled networks involves both data traffic in the DP and control traffic between the CP and DP. The former is a basic network service for end points, and the latter is used to ensure normal network management. In this subsection, fault tolerance for network traffic is introduced from two perspectives: fault tolerance for data traffic (Section VII-B1) and for control traffic (Section VII-B2).

1) Fault Tolerance for Data Traffic: Link failures are the main reason for an unavailable DP. To address this issue, the failover mechanism, which is the process of choosing other paths on which to continue forwarding network traffic when a link failure occurs, has been developed. Traditionally, the failover mechanism is implemented by providing redundant links for network traffic, which can be achieved through persistent traffic mirroring and the custom configuration of individual forwarding devices. With the emergence of SDN/OpenFlow, more flexible failover mechanisms have been proposed. These mechanisms can be divided into two types, namely, *restoration* and *protection*, as shown in Fig. 6.

a) Restoration: In SDN networks, when a link failure occurs, the switch can send a port status message to the SDN controller, and the controller will recompute a new path for the affected network flows and move them to this new path by reinstalling the necessary flow rules in the switch. This process is known as restoration failover, and it benefits from SDN programmability. CORONET [179] is a restoration failover technique that uses the controller to replan the links for affected traffic. There are four components in CORONET, each responsible for a specific task: the topology discovery module periodically collects network topology information with which to construct a view of the latest global topology; the route planning module uses Dijkstra's algorithm to calculate a routing path as a backup path in the case of link failure; the VLAN switch configuration module configures the switch port with a VLAN ID by means of an OpenFlow API to enforce

TABLE XII
CONFLICT RESOLUTION

Type	Reference	Goal	Solution
Operation isolation	FlowVisor [161]	Slice networks to achieve strong isolation to support nonintrusive multi-tenancy scenarios.	Develop an abstraction layer with virtual flow tables for each slice to allow multiple logical networks to coexist.
	NVP [26]	Network virtualization for multi-tenant data centers.	Develop an abstraction-based end-host network virtualization technique that enables tenants to deploy individual network services on virtual networks.
	Splendid [168]	Slice networks at the language level without harmful interference.	Endow a programming language with the capability of isolating networks through traffic, physical and control isolation.
Policy composition	Frenetic [24]	Provide see-every-packet abstractions, composition properties and race-free semantics.	Design a declarative programming language that can disambiguate overlapping rules by re-assigning their integer priorities (i.e., parallel composition).
	Pyretic [48]	Combine multiple policies with which to build an integrated app rather than coordinating or isolating them.	Extend on Frenetic to support both parallel and sequential composition.
	FlowBricks [173]	Integrate network services from heterogeneous controllers into the same traffic.	Provide interfaces for composition configurations with parallel and sequential operators and combine flow tables based on these configurations at run time.
	CoVisor [175]	Allow multiple controllers to cooperate in managing the same shared traffic.	Develop a compositional hypervisor with tolerance to controller and switch failures for the multicontroller scenario, implemented with three composition types
	Action compositon [176]	Resolve the issue of action concatenation in flow rule composition.	Formalize the action composition problem as a Hamilton path search problem on a directed weighted graph for fast composition of action lists without action redundancy.
	PGA [49]	Implement automatic policy composition.	Design a graph-based model for specifying network policies with the capability of automatic policy composition.
Module coordination	PANE [177]	Decompose the control and visibility of the network and resolve conflicts among different apps.	Design a tree-based control mechanism for the invocation of network resources and provide a configuration API for implementing participatory networks.
	Corybantic [69]	Coordinate controller modules to achieve systemwide objectives under global constraints.	Implement a voting-based resource-level coordinator for modular composition and resolve potential resource conflicts to allow each module to propose its own goal.
	Athens [178]	Automatically resolve resource conflicts considering both precision and parity.	Develop a coordinator modified from Corybantic for modular composition, which is implemented based on a family of voting procedures.
	Statesman [25]	Provide a network-state management service for multiapplication scenarios.	Design a priority-based policy arbitration system and resolve resource conflicts dynamically by loosening the coupling between modules and SDN controllers.
	Volpano *et al.* [93]	Fine-grained control conflict resolution.	Model each network control function as a deterministic finite-state transducer and leverage standard proof techniques to control conflicts among these transducers.

this routing path; and the traffic assignment module allocates the traffic from the host to the corresponding routing path.

Since CORONET can calculate new paths from a global view of the entire network via disjoint path calculation algorithms, it can achieve optimal resource utilization. However, choosing new paths on the fly may result in unacceptable recovery times. A survey [180] has shown that due to the inefficient software and fundamental traits of switch hardware, the installation of new rules suffers from surprisingly high latency, e.g., 8 ms per packet on average in the in-bound mode and 3 ms and 30 ms per rule for insertion and modification, respectively, in the out-bound mode. When combined with the delay introduced by the path calculation algorithms, these latencies may cause network recovery times to increase to unacceptable levels [181] and result in difficulties in satisfying the carrier-grade recovery requirement of 50 ms [182]. Furthermore, the computational and memory resources required by the controller for handling recovery messages may be too high to permit scalability.

To reduce the time needed for path calculation, Li *et al.* [183] implemented a locally optimal approach for migrating the affected flows. This approach uses the connectivity matrix table and traffic statistic table in the failed link switch instead of the statuses of all paths to find a new path. It can find a locally optimal path and reroute a flow within 36 ms, which meets the carrier-grade requirement.

b) Protection: In protection failover, the backup paths are predefined and reserved before a link failure occurs, which can lead to a faster switching time. To implement this mechanism, introduced in version 1.1, the OpenFlow specification uses *group tables* to permit the predefinition of failure recovery policies on devices and supports forwarding behaviors that depend on the local states of switches [184] without the need for path calculations performed by controllers. The group table also contains entries consisting of the *group identifier*, *group type*, and *counters* and an ordered list of *action buckets*. Each bucket contains a set of actions that can apply more complex packet forwarding semantics (e.g., multipath routing, fast rerouting, and link aggregation) on packets that cannot be defined by a flow entry alone [40]. Once the *group type* has been set to fast failover, the switch forwards packets only in accordance with the first live action bucket, whose liveness for the associated parameters (a port or another flow group) is monitored through end-to-end liveness mechanisms (e.g., a spanning tree or keepalive mechanism). When the first bucket is down, the next live bucket is automatically chosen to continue traffic forwarding without consulting controllers. Similar to common flow entries, these group entries can also be installed by the controller, and network programmers can add their own failure recovery policies to the group tables by installing group entries. Protection failover possesses the advantages of lower reaction latencies, faster network restoration and lower load on the controllers, and it has been shown to be a more suitable recovery mechanism than restoration for traffic tolerance [185].

Since the fast failover tables are preplanned and the liveness monitoring is limited to local network elements, this mechanism can react only to local failures and may lead to the use of nonoptimal backup paths [186], [187]. Sahri and Okamura [186] combined the advantages of controller recomputing and fast failover groups. When a failure occurs, the affected flows will be forwarded via precomputed backup paths until a new optimal path based on the current network state has been recomputed and deployed in the switch

by the controller. A buffer is also implemented in the switch to minimize the effects of switching traffic among different paths. This approach can ensure the optimal utilization of network links, but it can also lead to high overhead in both controllers and switches. In contrast, DFRS [187] is a declarative failure recovery system that allows network operators to choose either failover protection or failover restoration independently for different flows. Its core idea is that when a failure occurs, backup paths will be used for delay-sensitive flows, whereas delay-tolerant flows will be forwarded to the controller. By providing a set of high-level interfaces based on Scala DSL, DFRS can serve as a feasible and low-memory-overhead recovery system.

Another issue in failover protection is that the number of false positives for failure detection cannot be reduced since the liveness mechanisms, e.g., spanning tree or keepalive mechanisms, are managed by code outside of the OpenFlow specification [40]. The detection times of these protocols remain slow, and their accuracy is low. To reduce the detection time and the number of false positives, the Loss of Signal (LOS) and Bidirectional Forwarding Detection (BFD) protocols [188] have been implemented to work with OpenFlow fast failover groups [189], [190]. LOS can detect failures of a signal or connection due to a number of causes, e.g., a lost connection to the other end, an improper network configuration, or a bad cable connected to a network device. The BFD protocol can detect a link failure by detecting link loss by means of frequent control-echo sessions between each link. When per-link BFD or per-port LOS is used as the liveness mechanism, the time of failure detection for fast failover can be reduced to a sub-50 ms detection window.

The memory resource overhead is also a challenge in failover protection since it is necessary to deploy additional rules in switches. However, the memory in switches is often limited and expensive, especially for TCAM-based switches. To address this issue, Stephens *et al.* [191] first proposed a flow table compression algorithm to decrease the number of TCAM states consumed by forwarding table entries. However, this algorithm can only compress table entries with the same out port and the same modification actions. To improve the compression ratio, these authors further introduced the concept of *compression-aware routing* to reduce the number of flow table states without impacting resilience or performance. The authors then combined these techniques with Plinko, a new forwarding model in which the same action is used for every packet, to realize fast failover for multiple failures via forwarding table compression.

To facilitate the flexible deployment of failover policies, SDN programmability offers the possibility of integrating these policies into network programs. FatTire [95] is a high-level declarative programming language that attempts to implement this idea by allowing programmers to specify the abstractions of their failover policies. FatTire is a new programming technique based on regular expressions whereby programmers' failover requirements can be declaratively specified by specifying legal network paths. The specified policy is then compiled into low-level OpenFlow group entries. With FatTire, various failover policies (e.g., the modulo strategy, depth-first search strategy, and breadth-first search strategy in [192]) can be implemented more efficiently with high correctness and robustness. However, for now, this language can only deal with link-level failures; switch-level fault tolerance remains to be addressed.

c) Discussion: Although protection failover with the OpenFlow fast failover mechanism has been more widely studied than restoration failover has, the protection approach suffers from the two main drawbacks of nonoptimal path choice and resource utilization, especially for large-scale networks. Combining these two failover approaches is a good choice for achieving a balance between response time and path choice, as demonstrated by DFRS [187]. In addition to providing failure recovery for the DP, OpenFlow fast failover groups can also be used to implement several troubleshooting functions, e.g., *snapshot collection*, *anycast specification*, *blackhole identification*, and *critical node detection* [193].

2) Fault Tolerance for Control Traffic: In addition to fault tolerance for data traffic, providing fault tolerance mechanisms for control traffic (i.e., data traffic between controllers and switches) is crucial since a disconnection between a controller and switches may disable normal network processing. The control-switch channel can be implemented in two modes: in-band and out-band. In in-band control networks, the control traffic and data traffic are combined and share the same network resources. In the out-band mode, these two types of traffic are separated and implemented in different networks. Although an out-band network has obvious advantages in terms of reliability and security compared with an in-band network, building two independent networks is often too expensive and not feasible for large-scale networks [194]. Thus, in-band network control has become the preferred solution for deploying SDN networks. Related to the protection of control traffic to guarantee the reliability of networks, two issues have emerged: the protection of the control paths from switches and the placement of controllers to maximize connection reliability. In this section, we discuss fault tolerance mechanisms for control traffic failures due to link failures; the problem of controller placement will be discussed in the next Section VII-C.

Sharma *et al.* [194], [195] studied two failover mechanisms (i.e., restoration and protection) for control traffic in in-band SDN networks. In the *restoration* mechanism, all switches are preconfigured with a one-hop restoration path for the working path for control traffic. The control messages from a switch can be forwarded only by the neighbor on the working path. When a failure occurs, the controllers will collect the states of the other neighbors of the faulty switch and compute a new path on the restoration path to recover the control traffic affected by the faulty switch. However, this approach may result in considerable traffic loss due to the time consumed for restoration. Protection for control traffic was therefore also studied, with an approach similar to the protection mechanism for data traffic that is implemented by means of OpenFlow group tables. When a failure occurs, the switch can automatically forward control traffic over the backup path indicated by the group table without consulting the controller. These authors showed that the protection mechanism for control traffic can also satisfy the carrier-grade recovery requirement.

Hu *et al.* [196] and Obadia *et al.* [197] considered the problem of control traffic tolerance for a scenario with multiple controllers, rather than a single-controller scenario, as considered in [194], [195], and [198]. In the multiple-controller scenario, the network is divided into several domains, with each being controlled mainly by one of the controllers. Hu *et al.* [196] implemented a control traffic protection mechanism based on a combination of local rerouting protection and constrained reverse forwarding protection. Whereas the local rerouting protection mechanism attempts to forward control traffic to a neighboring switch during link failure, the reverse forwarding protection mechanism attempts to forward control traffic back to the downstream switch. For each link failure, one of these two mechanisms is chosen for reconnecting to the nearest controller, with the objective of recovering control traffic with the minimum number of hops to the controller.

Obadia *et al.* [197] focused on control traffic failover for distributed SDN controllers in different control domains. The question of how to quickly reconnect the switches in an invalid domain (called orphan switches) to other controllers was studied on the basis of two failover mechanisms: *greedy failover* and *prepartitioning failover*. In greedy failover, when a controller fails, each switch orphaned by that controller will automatically broadcast specific link layer discovery protocol (LLDP) messages, which are necessary to modify the switch software, and the controllers around these orphan switches will progressively take them over into their own domains. In contrast, in prepartitioning failover, the failed controller is responsible for choosing and informing its neighbors of which switches they should take over. This mechanism does not need to modify switches and can be fully compliant with OpenFlow; however, the coordination among controllers in the case of failure is an error-prone process.

C. Infrastructure Planning

The stability of the infrastructure determines the reliability of the upper services. Designing the infrastructure to provide fault tolerance is essential for improving the reliability of SDN networks. In this section, infrastructure planning is discussed from two perspectives: component redundancy (Section VII-C1) and controller placement (Section VII-C2).

1) Component Redundancy: Since the controller is the brain of an SDN network, guaranteeing its availability and survivability is essential. Once the controller is down, the network elements will be unavailable for normal network requests. An effective solution to this single point of failure is to use multiple controllers regardless of clustering or backup [199]. If the primary controller breaks, the slave controllers or backup controllers can take over the management of the whole network, which can allow the network to continue operating. In the multiple-controller scenario, a distributed data store with a synchronization mechanism is adopted to ensure the state consistency among these controllers to ensure the reliability of the whole network [199].

Li *et al.* [200] used a state machine replication mechanism implemented with the Byzantine fault tolerance (BFT) mechanism to ensure smooth network functioning. In this mechanism, each switch is connected to multiple controllers; if the primary controller fails, the next primary controller will be selected from among the backup controllers through a proper election algorithm. They also proposed the Requirement First Assignment algorithm to solve the controller assignment problem in fault-tolerant SDN. Similarly, Botelho *et al.* [32] employed the Paxos algorithm to implement a data store in the form of a replicated state machine (RSM), which was used to integrate fault detection and leader election algorithms without the need for additional coordination services. In their architecture, the controllers maintain a local data cache to reduce the read frequency of the RSM.

Although such a distributed storage system and RSM can be used to replicate durable states, the consistency between the controller and switches cannot be ensured. Thus, Ravana [201], a fault-tolerant CP, attempts not only to replicate the states of the controllers but also to ensure the consistency of the external switch states. The RSM in Ravana is extended to ensure control state replication, and an extension of the OpenFlow interface is adopted to ensure that each transaction can be executed in an ordered manner and exactly once across the switches. The main issue of concern is how to handle the switch consistency during controller failures based on maintaining consistent controller states.

2) Controller Placement: Resource redundancy is useful for enhancing CP survivability; however, it may not be sufficient to provide fault tolerance against both network disruption and controller overload and can cause additional problems such as route flapping and prolonged route convergence times. To address these issues, the problem of finding the optimal controller placement has received significant attention [202]–[209]. The controller placement problem has two aspects: the *number* and locations of controllers [210]. The first aspect concerns how many controllers need to be deployed to implement a reliable and resilient network. The second aspect concerns where these controllers should be deployed, which affects several important metrics, e.g., the resilience of control traffic, the quality of network services, and controller performance. Finding the optimal controller placement for specific metrics is studied in this subsection.

Several studies have focused on intelligent controller placement according to various metrics to achieve improved design and performance of SDN networks. Zhang *et al.* [202] focused on a metric concerning the invalidation of nodes, links, and connectivity between controllers and switches. They formulated the placement problem in the SDN network and proposed a min-cut-based graph partitioning algorithm for controller placement to maximize the resilience of the network. Hu *et al.* [203] considered a metric reflecting the expected percentage of control path loss. They proposed several controller placement algorithms to minimize this metric, e.g., random placement, *l-w-greedy* placement, and simulated annealing. Guo and Bhattacharya [204] leveraged an interdependence graph to analyze the cascading behavior of a failure, with the steady state used to define a resilience metric for controller placement, and applied a greedy algorithm to partition the network into a set of subnetworks and a selection algorithm to choose the controller position in each subnetwork.

The above approaches are useful but may have some limitations, as analyzed by [205]: they often use single paths to model the connections between controllers and switches, handle traffic load changes on demand, and overlook the effects of predefined failover mechanisms. To address these issues, Survivor [205] has been proposed as a controller placement strategy that considers path diversity, thus achieving capacity awareness in controller placement and improving the performance of failover mechanisms.

Bari *et al.* [206] attempted to implement dynamic controller provisioning rather than offline planning by minimizing the metrics of flow setup time and communication overhead. To this end, they formulated the dynamic controller provisioning problem (DCPP) as an integer linear program (ILP), which they solved with two heuristic algorithms to dynamically adjust the number and locations of controllers according to the current number of flows. Ros and Ruiz [207] also proposed a heuristic algorithm for solving the fault-tolerant controller placement problem, but with the objective of achieving at least five-nines southbound reliability.

These ILP- and heuristic-based controller placement methods can address the resilience problem only in terms of specific metrics and not for all objectives simultaneously. POCO [208] is a framework for controller placement that considers a trade-off among several performance and resilience metrics, e.g., the latency between nodes and controllers, resilience and load balancing. A Pareto-based optimal controller placement approach that can evaluate the entire solution space and provide a comprehensive placement based on all objectives is implemented. This solution has been extended with a heuristic approach (Pareto simulated annealing) to achieve a trade-off between calculation time and placement accuracy for supporting large-scale and dynamic WANs, as reported in [209].

D. Summary

In this section, we have discussed fault tolerance techniques for SDN from three perspectives: conflict resolution, which enables multiple tenants with various independent apps to coexist on the same network; fault tolerance for network traffic, which provides fault-tolerance capabilities for both data traffic and control traffic in SDN-enabled networks; and infrastructure planning, which focuses on how to design infrastructure deployments to satisfy reliability and other requirements. The main conflict resolution approaches are summarized in Table XII. We further summarize the fault tolerance techniques for traffic and infrastructure planning in Table XIII to conclude this section.

VIII. Fault Management Gap Analysis

While the academic literature indicates that researchers have comprehensively addressed SDN fault management problems, the extent of implementation of fault management in existing SDN-related frameworks remains unknown. Thus, we survey fault management work in popular SDN-related frameworks and attempt to analyze the gap in fault management between academia and industry. We survey open-source SDN controllers in Section VIII-A. Many SDN controllers have been developed since the proposal of SDN [6]. Some SDN controllers, e.g., OpenDaylight (ODL) [220], have evolved into SDN ecosystems. Numerous projects and new features are continuously being added in the controller community. These projects are integrated with core projects to function as complete SDN controllers. Our survey of SDN controllers focuses on the fault-management-related projects (summarized in Table XIV) in SDN controller ecosystems. In addition, we analyze the gap between solutions developed in an academic research context and practical deployments for SDN fault management.

A. Current SDN Controller Platforms

The first SDN controller, NOX [221], was introduced together with OpenFlow in 2008 [6]. Since then, many SDN controllers, e.g., POX [222], ONIX [223], Beacon [224], Floodlight [225], ODL, and ONOS [226], have been developed in both academia and industry. However, some of these controllers are no longer being actively maintained. The first SDN controller, NOX, is no longer under active development due to its difficulty in scaling, while its Python sibling, POX, remains in limited use by the research community. The Beacon controller was popular in 2010 but was replaced by Floodlight in 2013. Only a few controllers are currently under active development. We present a brief survey of some important development communities and commercial SDN controllers in Table XV. We find that while some companies build their own proprietary controllers, more than half of them build controllers based on open-source software, e.g., ODL. Based on this survey, we select controllers that are still under active development, including ODL, ONOS and OpenContrail, as representatives for evaluation.

1) OpenDaylight: ODL is currently the largest open-source SDN controller. It is a collaborative open-source project hosted by the Linux Foundation. The members of the ODL community include Cisco, Ericsson, Intel, Brocade, Google, Huawei and other Internet and telecommunication companies. ODL was launched in February 2013 and was announced as a community-led project in April 2013. Currently, ODL has grown to be the largest open-source SDN controller. According to its project list,[8] there are 65 approved projects, including 6 kernel projects, 18 protocol and service projects, 33 application projects and 8 support projects. In a survey of each of these projects, we find three projects related to fault management: Cardinal [211], TSDR [213] and Centinel [212].

Cardinal [211] was proposed to provide monitoring-as-a-service in ODL by serving as a monitoring proxy for the centralized network management system (NMS), as shown in Fig. 7. In legacy networks, the NMS is a centralized system that monitors and manages devices throughout the network via standard protocols, e.g., SNMP. With the advent of SDN, the need for monitoring has become a whole-network issue, including controllers, devices, and deployed features. Cardinal collects statistics from devices and deployed feature statistics from other services in the controller, and it reports these data

[8]OpenDaylight projects - https://wiki.opendaylight.org/view/Project_Proposals.

TABLE XIII
FAULT TOLERANCE FOR TRAFFIC AND INFRASTRUCTURE

Technique	Type	Reference	Goal	Proposed Solution
Traffic tolerance	Data traffic tolerance	Kim *et al.* [179]	Recover from link failures in the DP.	Propose CORONET, which uses the controller to recompute paths in the case of link failures.
		Li *et al.* [183]	Recover from link failures in a locally optimal manner.	Migrate the affected flows according to connectivity matrix tables and traffic statistics.
		Sahri *et al.* [186]	Optimize path protection failover.	Forward traffic via precomputed backup paths and then reroute traffic according to controller-recomputed paths.
		DFRS [187]	Optimize path protection failover.	Forward delay-sensitive flows via backup paths and delay-tolerant flows to the controller.
		Sharma *et al.* [189]	Recover from link failures within a 50 ms interval.	Implement a LOS-based restoration mechanism and a BFD-based protection mechanism to satisfy the 50 ms recovery requirement in OpenFlow networks.
		Van *et al.* [190]	Reduce false positives and time overhead in failure detection.	Introduce BFD as the liveness mechanism to work together with OpenFlow group tables.
		Stephens *et al.* [191]	Improve the utilization of memory resources for fast failover.	Propose a flow table compression algorithm to decrease the number of TCAM states consumed by the forwarding table entries.
		FatTire [95]	Simplify the deployment of failure recovery policies.	Propose a new programming construction mechanism based on regular expressions to specify legal forwarding paths.
		Borokhovich *et al.* [192]	Improve the robustness of DP connectivity with local fast failover.	Provide three graph algorithms for fast failover implemented with FatTire.
	Control traffic tolerance	Sharma *et al.* [194], [195]	Implement failure recovery for in-band control traffic.	Implement two failover mechanisms (i.e., restoration and protection) for control traffic.
		Hu *et al.* [196]	Achieve control traffic protection with multiple controllers.	Combine local rerouting protection with constrained reverse forwarding protection.
		Obadia *et al.* [197]	Achieve control traffic failover for distributed SDN controllers.	Develop two failover mechanisms (greedy incorporation and prepartitioning) to migrate the control of orphan switches to other active controllers.
Infrastructure planning	Component redundancy	Tootoonchian *et al.* [199]	Improve the scalability of OpenFlow controller.	Present a distributed event-based CP with the capability of network partitioning and component failure recovery.
		Li *et al.* [200]	Provide the fault tolerance capability for SDN network control.	Implement a state machine replication approach based on a Byzantine mechanism for multiple-controller scenarios.
		Smartlight [32]	Provide a fault-tolerant SDN controller.	Formalize the data store as a replicated state machine (RSM) and allow the primary controller to maintain a data cache.
		Ravana [201]	Incorporate switch states into a fault-tolerant SDN controller.	Extend RSM and OpenFlow interfaces to ensure the consistency of the controller states and switch states.
	Controller placement	Zhang *et al.* [202]	Maximize the resilience of networks.	Propose a min-cut-based graph partitioning algorithm considering the outage of nodes, links, or connections between devices and controllers for controller placement.
		Hu *et al.* [203]	Maximize the reliability of control networks.	Provide several placement algorithms to minimize the expected percentage of control path loss.
		Guo *et al.* [204]	Consider a metric of cascading failure relations for controller placement.	Analyze controller placement using an interdependence graph and two algorithms for partitioning the network and placing controllers.
		Survivor [205]	Provide a more realistic controller placement strategy.	Propose a controller placement strategy that considers path diversity, capacity and failover mechanisms to improve SDN survivability.
		DCPP [206]	Decrease the flow setup time and communication overhead in controller placement.	Dynamically adjust the number and locations of controllers dynamically according to the number of current flows.
		Ros *et al.* [207]	Achieve five nines reliability of SDN network control.	Formalize the fault-tolerant controller placement problem and solve it with a heuristic algorithm to achieve the required reliability.
		POCO [208], [209]	Develop a trade-off controller placement algorithm.	Implement a Pareto-based optimal controller placement algorithm to balance all considered objectives.

to the NMS above. Currently, Cardinal Boron is only able to provide monitoring and basic traps.

The Time Series Data Repository (TSDR) [213] is a distributed time-series data collector that collects data via standard protocols, e.g., OpenFlow counters, and sinks them into a distributed database, e.g., HBase, with timestamps. Currently, TSDR is able to collect controller metrics, NetFlow statistics, OpenFlow statistics, sFlow statistics, SNMP statistics and SysLog statistics. Although TSDR can be used to improve the scalability and performance of ODL controllers, its primary objective is to help to create an intelligent and "smart" controller.

Centinel [212] is another project in ODL, also focusing on streaming data. It is a distributed framework for collecting, aggregating and sinking streaming data. It enables SDN controllers to receive events from multiple streaming sources, e.g., SysLog, and execute batch processing or real-time analytics. Centinel has some overlap with TSDR in the data collection aspect, as shown in Fig. 8. TSDR and Centinel both provide network data collection and analytics. The collected data can be used to monitor the status of the SDN network or for further potential fault analysis.

2) ONOS: The Open Network Operating System (ONOS) is a carrier-grade SDN network operating system designed to provide high availability, performance and scalability. The members of the ONOS community include AT&T, Cisco, Ericsson, Google, Huawei, Samsung, and Verizon. In December 2014, the Open Networking Lab, along with its industry partners, released the ONOS source code to the open-source community. In October 2015, ONOS joined the Linux Foundation as a collaborative project. The first released version was Avocet. ONOS is now on its sixth version: Falcon. There were 30 projects as of November 2016, including core projects, incubation projects and projects in proposal. Five of these projects are related to fault management: OPEN-TAM [214], Fault Management [215], Composition Mode [216], Network TroubleShooting Module [217] and Network Artificial Intelligence [218].

TABLE XIV
FAULT-MANAGEMENT-RELATED PROJECTS IN SDN

Platform	System	Project	Key Words	Focus
SDN	ODL	Cardinal [211]	Monitoring-as-a-Service and Network Management System.	Enable an SDN network to be remotely monitored by the NMS.
		Centinel [212]	Streaming Data, Batch Processing and Real-Time Processing.	Collect and analyze SDN system data.
		TSDR [213]	Time Series Data and Data Repository.	Collect, store and model time-series data.
	ONOS	OPEN-TAM [214]	Traffic Analysis and Monitoring, Flow Sampling and Deep Packet Inspection.	Analyze and monitor various types of network traffic.
		Fault Management [215]	Network Element, Alarm, Fault and Event.	Provide ONOS with fault management features.
		Composition Mode [216]	Flow Rule, Conflict Composition and CoVisor.	Implement policy composition features to support multi-tenancy scenarios.
		Network TroubleShooting Module [217]	Routing Loop, Routing Black Hole and app Conflict.	Troubleshoot network faults.
		Network Artificial Intelligence [218]	Machine Learning, Self-Adjustment, Self-Optimization and Self-Recovery.	Troubleshoot network problems, predict network traffic and defend against network attacks.
	OpenContrail	Analytics Node [219]	Data Collection, Analytics and Sandesh.	Gather traffic information from the DP.

TABLE XV
SDN CONTROLLERS

Controller	Developer	Language	Base	Open Source?	Active?
NOX [221]	Nicira	C++/Python	/	Yes	No
POX [222]	Nicira, Berkeley	Python	/	Yes	No
ONIX [223]	Nicira, Google, NTT	C/Python	/	No	/
Beacon [224]	Stanford	Java	/	Yes	No
Trema [227]	NEC	Cucumber/Ruby	/	Yes	Yes
Ryu [228]	NTT	Python	/	Yes	Yes
Floodlight [225]	Big Switch	Java	Beacon	Yes	Yes
OpenContrail [229]	Juniper	C++/Python	/	Yes	Yes
OpenDaylight [230]	Cisco, Ericsson, HP, Intel, Brocade ...	Java	/	Yes	Yes
ONOS [226]	AT&T, Cisco, Ericsson, Huawei, Google ...	Java	Floodlight	Yes	Yes
OpenIRIS [231]	ETRI	Java	/	Yes	No
OpenMUL [232]	KulCloud	C	/	Yes	No
Kandoo [233]	U Toronto	Go	/	Yes	No

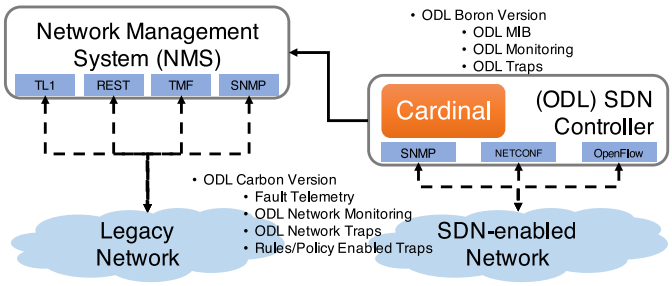


Fig. 7. OpenDaylight Cardinal. It enables the monitoring of an SDN network by the NMS.

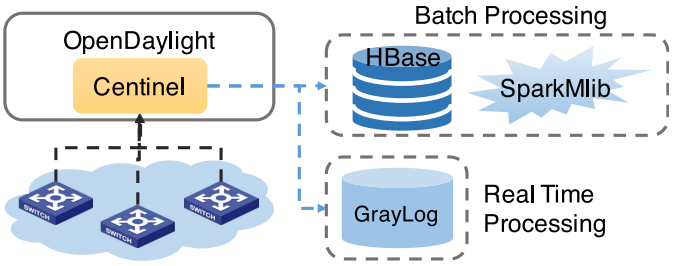


Fig. 8. OpenDaylight Centinel. A distributed, reliable framework for efficiently collecting, aggregating and sinking streaming data across a persistence database and stream analyzers.

OPEN-TAM [214] is a traffic analysis and monitoring project that enables the analysis and monitoring of various types of network traffic. Network traffic monitoring is a function that is crucial for various other functions, including traffic engineering and fault management. OPEN-TAM has two subsystems: the Adaptive Flow Sampling Service and the Open Selective-DPI Service. The Adaptive Flow Sampling Service can adaptively sample flow statistics to overcome the problems of performance degradation and low accuracy in current FlowRule services. The Open Selective-DPI Service can filter users' data traffic in the DP and classify it with app-level granularity using open-source DPI software.

Fault Management [215] is intended to support alarms from network devices. When a fault or event occurs, a network device typically sends a notification to network operators via certain protocols. Fault Management is designed to receive such notifications or alarms, store them, and make them externally visible. There are two components in Fault Management: the Protocol Provider (e.g., SNMP or NETCONF) and the Fault Management Application (which stores and displays notifications or alarms).

Composition Mode [216] allows ONOS to run multiple apps concurrently and automatically resolves flow conflicts. It supports the parallel, sequential and override composition operators, as described in Section VII-A2. Composition Mode, as shown in Fig. 9, has 4 components: Policy Interface Definition, FlowRuleService Implementation, Composition Library and

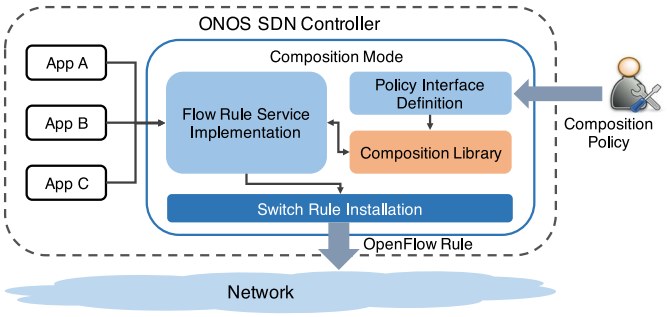


Fig. 9. ONOS Composition Mode. It can resolve conflicts among different apps using three composition operators (as illustrated in Fig. 5).

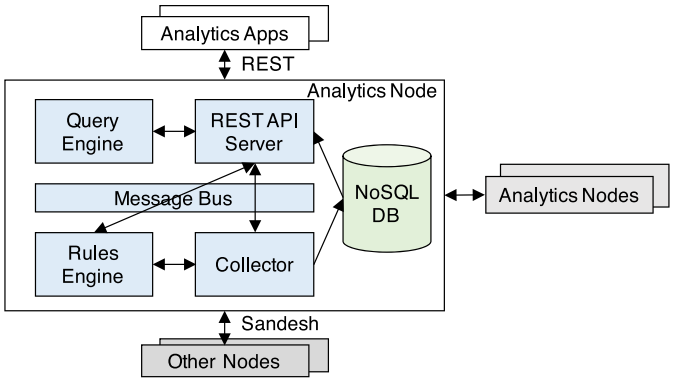


Fig. 10. OpenContrail Analytics Node. An analytical framework for the OpenContrail system.

Switch Rule Installation. Policy Interface Definition interprets the composition policies defined by network operators and configures ONOS to apply these policies. FlowRuleService Implementation maintains flow tables for each switch and is responsible for the details of composition. Composition Library is a stateless library that has access to apps, intermediate flow tables and policies. Switch Rule Installation installs OpenFlow rules into physical switches.

Two additional fault-management-related projects, Network Trouble Shooting Module [217] and Network Artificial Intelligence [218], have also been proposed in ONOS. They are in their initial stages. Network Trouble Shooting Module aims to improve the reliability of SDN, mainly by solving the problems of routing loops, routing blackholes and app conflicts, as indicated in the proposal. However, at present, only two algorithms have been developed: the Routing Loop Detection Algorithm and the Routing Black Hole Detection Algorithm. The app conflict problem remains to be addressed. The Network Artificial Intelligence project in ONOS is similar to TSDR and Centinel in OpenDaylight and utilizes Apache Flume and Kafka for streaming data collection and analytics. However, no detailed information about this project is available.

3) OpenContrail: OpenContrail is an open-source network virtualization platform for the cloud that supports secure multitenancy and enables dynamic service chaining in private, public and hybrid clouds using SDN and NFV techniques. Juniper acquired this technology in 2012 and began building on its SDN capabilities. It was first released in September 2013 and is mainly supported by Juniper.

OpenContrail includes the Analytics Node project [219] for data collection and analytics, as shown in Fig. 10. Analytics Node uses an XML-based protocol called Sandesh for high-volume data collection. It collects asynchronous messages from other nodes, such as logs, events and traces. It can also collect synchronous messages by sending requests for the collection of specific operational states from other nodes. All information is persistently stored in a NoSQL database. Analytics Node also provides a northbound REST API for other analytics apps.

B. Gap Analysis

Although a large number of fault management solutions have been proposed and evaluated in academic studies, we find that few of them have been applied in practical commercial deployments, and the projects listed in Table XIV are still in their initial stages, with limited fault management capabilities. Most of the projects in Table XIV enable only the monitoring of SDN networks and the collection of statistical data; some projects, such as Network TroubleShooting Module and Network Artificial Intelligence in ONOS, cannot be directly applied. Leveraging the centralized nature of SDN to collect statistical data for network maintenance is highly feasible since SDN can simplify data collection by virtue of its global overview of the network. In addition, the programmability of SDN allows network operators to design automated fault diagnosis and repair solutions, which can relieve operators of the task of having to analyze large amounts of data. However, as found in our survey, applying such features in practical network deployments is currently still quite difficult. In this subsection, we attempt to identify the reasons for this gap between academic research and practical deployments in terms of the state of development of SDN fault management techniques from two perspectives.

1) Issues in Academic Research:

a) Complexity of the production network environment: The main issue hindering the practical application of many solutions developed in an academic research context is that the experimental environments in which they have been tested are often small in scale, with a limited amount of equipment, and most solutions are actually simulated. In addition, many solutions depend to some extent on certain assumptions, such as a fixed network topology, a fault-free DP in the case of control-message-based configuration verification, and unlimited hardware resources. However, practical operating environments are diverse and complex since practical networks often need to support many network services, e.g., streaming media, IP voice, L2/L3 VPNs, 3GPP mobile backhaul and core transmission, and cloud services. Some unexpected cases may exist that cannot be extensively considered when network vendors are designing fault management mechanisms for their productions. Therefore, the simple environments and strong assumptions considered in academic experiments often make many solutions difficult to apply in production environments.

b) Diversity of network devices: Another issue is that many academic research solutions have been proposed only

for pure SDN networks based on a single protocol. However, many practical networks need to include legacy or proprietary devices, which may be beyond the control of SDN. Even in a pure SDN network, various protocols (e.g., OpenFlow, BGP and SNMP) may be used simultaneously to manage the DP. These networks are often heterogeneous and contain various devices. For example, in many cloud data centers, the virtual switches for virtual machines are controlled via SDN technology with OpenFlow and OVSDB, whereas the physical switches connecting servers still use CLI solutions, which are configured by SDN controllers or operators. Furthermore, even in a network based on a single protocol, interoperability problems arise among different versions of OpenFlow; e.g., OpenFlow 1.0 is not fully compatible with OpenFlow 1.3. The hybrid network paradigm, in which SDN and traditional network devices are integrated in a single network, is expected to persist for a very long time. Thus, providing a solution for interoperability testing and management to address the diversity of network devices will be necessary for future network development [55], [234].

2) Issues in Practical Deployments:

a) Hindrances to new network implementations: The slow adoption of SDN is one of the factors that is limiting the development of related fault management techniques in practical networks. While SDN has seen many successful deployments, it still suffers from many issues preventing its widespread adoption [13], e.g., issues of reliability, security, performance and scalability. For example, when considering the implementation of SDN networks, network vendors and users first need to consider the benefits and the necessity of updating their products [13]. The benefits are the cost savings of implementing new technologies, and the necessity to update concerns how and why the existing products are insufficient. In addition, the process of updating products often suffers from a long cycle time since a wide variety of updates may be needed concerning, e.g., compatibility with old devices, the reliability and security of the new software system, and infrastructure maintenance. These factors are among the major hindrances to the adoption of SDN and the development of related fault management projects.

b) Disunity of architecture and interfaces: The hierarchical SDN architecture and the interfaces between different planes have yet to be effectively unified and standardized [234], which complicates the development of associated projects, including fault management. First, with the increasing requirements for and sizes of networks, the SDN CP must be scaled in terms of horizontal expansion and vertical stratification. This has resulted in various hierarchical CP architectures in different network domains. For example, while a CP with a single-layer architecture is used in data centers, a CP with a multilayer architecture is needed for mobility core networks [235]. Second, the SDN SBI and NBI have become diversified, and there is also no consensus on the development of EBIs/WBIs. Therefore, the SDN architecture and its interfaces will require further study and clarity for standardization and unification. This will allow rapid development of related technologies, which can, in turn, promote the development of SDN itself.

c) Changes in management patterns: The emergence of SDN has affected both network architectures and network management patterns. A single SDN network involves multiple vendors (e.g., app vendors, controller vendors and device vendors), and the network management pattern must coordinate the products from all these vendors. However, this is challenging to achieve [234], especially with regard to network reliability. Fault management solutions need to address not only issues of horizontal incompatibility and interoperability in the same plane (e.g., the interoperability among network devices, as discussed above) but also issues of vertical (i.e., cross-layer) collaboration, such as cross-layer diagnosis, as addressed in [92]. We believe that designing new techniques to address the interoperability problems that arise in multivendor integration is highly necessary for reliability, and we also suggest that providing an incremental plan for SDN deployment (such as those considered in [236] and [237]) is another efficient approach for relieving interoperability issues, which can balance the benefit of replacing legacy devices with the cost of addressing interoperability issues when integrating multiple network techniques.

d) Changes in certifications: The last, but no less important, source of practical hindrances to fault management is the changes in network certifications caused by the emergence of SDN. Since the software-centric nature of SDN can fundamentally alter network engineering and management, network engineers must know not only how to configure networks but also, and more importantly, how to program them. To deploy a reliable SDN network, network engineers must become familiar with more elements (e.g., controllers, apps, programming languages, and new switch architectures), in addition to command-line interfaces (CLIs), and must acquire more skills (e.g., troubleshooting, basic software tool debugging and automation), based on this new understanding of networks. Most network engineers will require retraining for this purpose. In addition, current SDN certifications for validating the skills of engineering professionals remain in an imperfect state, and only a few organizations, e.g., ONF and Cisco, provide such certifications. Thus, there is a need for more investment in training and certifications [234].

C. Summary

We have surveyed fault management projects related to open-source SDN controllers and presented a gap analysis between the solutions that have emerged from academic research and practical deployments. Unfortunately, these projects are in their initial stages; some have only simple implementations, and some even lack detailed proposals. The development of SDN fault management has been slowed by the current state of adoption of SDN, and this immaturity of fault management undermines the reliability of SDN, which, in turn, affects SDN development. We believe that to push SDN techniques forward, the industry will need to put greater effort into SDN fault management.

IX. Future Research Directions

By analyzing and comparing current solutions for SDN fault management, we have presented a comprehensive study of SDN reliability issues. However, several issues remain challenging and will require greater attention in future research. Thus, we attempt to identify several open challenges and potential directions for future research in this section.

A. Data Plane Programming

As analyzed in Sections III-A and V-A, most of the thorny issues regarding fault diagnosis and repair in SDN concern the DP, and they can have diverse causes, e.g., software bugs, hardware failures and external interference. Current probe-based testing solutions [15], [70] for these issues suffer from a long probe packet generation time, and solutions based on traffic statistics [73] require the addition of new pipelines in the switch datapath, which is also an error-prone process. The emergence of new DP specifications (e.g., P4 and POF) has inspired device manufacturers to develop new products that allow network operators to customize the DP and modify its features at maintenance time as well as at run time. Recently, much academic research has been conducted on P4-based fault management [30], [126], [238]–[240], and several commercial products[9] have emerged. Academic experiments show that P4 can optimize fault management for the DP and further improve SDN reliability, and commercial products also strongly prove the performance of P4. Thus, we believe that the design of more powerful programming protocols for the SDN DP, including network management, reliability, and security, is a promising future direction for SDN development, and we additionally believe that these DP programming languages can further upset the ecological balance of the current networking world and provide greater opportunities for many white card manufacturers.

B. Diverse Network Protocols

In current SDN frameworks, OpenFlow interfaces are the most popular type of interfaces for network devices. However, as SDN has evolved, many weaknesses of OpenFlow (e.g., its scalability, security and compatibility) have become amplified [241], making it difficult for OpenFlow to remain the only SDN protocol in widespread use. Thus, vendors are using other protocols, such as NETCONF, OVSDB, MPLS-TP, BGP, PCEP, ForCES, P4, and POF, to fill the voids in SDN network management capabilities left by OpenFlow or to directly replace it. These protocols are also used in various combinations, such as OpenFlow and NETCONF or OVSDB, to manage switches. As this trend has emerged, greater concerns about SDN reliability have also been exposed. This is because the existence of multiple protocols in an SDN network makes the DP more diverse and complex, and more complex programs also need to be provided in the CP to support multiprotocol network management. However, existing fault management solutions are all focused on networks based on only one protocol, typically OpenFlow. Thus, we believe that greater research efforts will be needed to address the diversity of network protocols used in SDN networks.

[9]One example is Barefoot Tofino (https://barefootnetworks.com/technology/#tofino).

C. Complex Software Systems

SDN has come to have a broader meaning, i.e., not simply CP/DP separation but also automation, virtualization and programmability. Behind this meaning is the need for complex software systems. In addition to normal network management, these systems also need to provide many other functions, such as modular collaboration, distribution, state synchronization, backup and restoration, and load balancing, to maintain the whole network. While these complex software systems require elaborate design, more effort also needs to be focused on monitoring, testing, evaluating and diagnosing these systems. However, most existing fault diagnosis solutions [20], [23], [107], [134] do not consider the complexity of the CP and assume that the controllers are failure-free. Only works such as [16], [17], and [153] provide solutions for testing and troubleshooting controller software, and we believe that greater effort is still needed to accelerate the software development and evolution of SDN controllers.

D. Network Engineer Training

As analyzed in Section VIII-B, network engineers need retraining to be able to integrate the SDN architecture. This is especially important for network maintenance. As surveyed in the preceding sections, many elements are involved in maintaining an SDN-enabled network, such as newly defined switches and various software entities, and network engineers need to know how to monitor and test these elements, log their states, and detect and localize potential faults throughout the system. In addition, they need to deploy fault tolerance mechanisms, including mechanisms for data and control traffic as well as infrastructure planning, on the switches and controllers. While the declarative programming languages that have been proposed for this purpose offer effective methods of simplifying network deployment in terms of both traffic configuration [24] and fault tolerance policies [95], we believe that greater effort needs to be focused on helping network engineers to understand SDN and how to deploy networks; one example of such work can be found in [242], where an automatic suggestion mechanism for writing test codes is proposed, which can be very useful for inexperienced network engineers.

E. Scalability

As a logically centralized network architecture, SDN faces scalability issues [243], [244]. These issues arise from several causes, including communication delays between the CP and DP [199], limited computational resources in controllers [244], and inconsistent data transmission rates [244]. To design a scalable SDN network architecture, in addition to resolving these issues, providing reliable software systems and stable connections between the CP and DP is the most important factor to be considered. Additional challenges related to SDN reliability that concern scalability, such as controller

failures, the distribution and consistency of controller states, and infrastructure planning [244], still require further research.

F. Intelligent Network Management

The future of networking lies in network automation, which is also one of the main premises of SDN. Recently, there has been a trend of introducing artificial intelligence (AI) and machine learning (ML) techniques into SDN to take over network management to achieve network automation [245], [246]. The logically centralized network control and the global visibility of the network provided by SDN allow AI/ML techniques to automatically make and adjust network decisions. Several academic research works and engineering projects [213], [218], [246] have attempted to provide such a combination of techniques. In [246], a new network paradigm based on a combination of SDN and AI/ML, called Knowledge-Defined Networking, was proposed by several universities and enterprises, e.g., Broade, HP, Intel, NTT, Cisco, and UC Berkeley. In this paradigm, a *knowledge plane* (KP) is established on top of the SDN architecture and is responsible for analyzing the network on the basis of the data collected by the management plane; ML is used to transform these data into knowledge and to make network decisions based on this knowledge. Various open-source projects, such as ODL TSDR [213] and Centinel [212], also provide similar functions of collecting system data from each SDN element and using ML tools (e.g., Spark) for automatic network management. As discussed in [245] and [246], the combination of SDN and AI/ML truly has the potential to enable automated network provisioning and management and to make networks more reliable and secure. We also believe that with greater effort, the combination of SDN and AI/ML will simplify network implementations to meet various demands and improve network reliability to an acceptable level for most users.

G. Self-Healing Networks

As discussed in the above section, network automation is a current trend. In addition to designing intelligent mechanisms to achieve automated network management, designing self-healing networks to ensure reliability is another important direction of research. A self-healing network is a network that has the ability to perceive incorrect states in its components and automatically recover itself to a normal state without human intervention. Implementing a self-healing mechanism for networks requires a comprehensive fault management solution, including an online monitoring system, a fault detection and localization mechanism for finding faults, a fault repair and recovery mechanism to restore the network, and a fault tolerance framework to maintain normal operations. SDN has enabled this innovation for networks, and several preliminary attempts, such as integrated fault troubleshooting systems [92], [155], [156], automated fault diagnosis and repair mechanisms [83], [85], [87], [100], and fault tolerance platforms with automatic recovery capabilities [31], have been presented in academic experiments. However, these designs are still in an initial stage and have many shortcomings, such as reliance on strong assumptions concerning the network states and software systems, incomplete repair mechanisms, and high overhead for recovery.

X. Conclusion

Although SDN promises network innovation, its reliability issues have demanded widespread attention from academia and industry. We surveyed academic publications on SDN fault management from the period of 2008-2017 and information on projects undertaken in open-source communities. We found that although the available academic solutions and projects address most SDN reliability issues, few can provide a complete solution for SDN fault management, and many faults encountered in SDN continue to be challenging. To address these issues, a systematic and comprehensive survey of fault analysis and an evaluation of existing solutions for and challenges facing SDN fault management will be necessary to guide future research. Unfortunately, this literature was still lacking.

In this paper, we conducted such a survey to present a deep and comprehensive understanding and analysis of SDN reliability issues. We started with an introduction of the characteristics of SDN, considering its current state of development, and provided a two-dimensional taxonomy of SDN fault management solutions as an overview. We then classified the types of faults that can occur in the SDN architecture by means of a deep analysis of their symptoms and root causes. Next, the core of the survey was presented from four perspectives, i.e., system measurement, fault diagnosis, fault recovery and fault tolerance, along with in-depth discussions and comparisons of existing solutions. After the discussion of the existing solutions that have been developed in an academic research context, a study of open-source fault management projects concerning SDN controllers was also presented. We found that many projects are still in their initial stages and will need greater effort to develop further. We therefore analyzed the gap of SDN fault management solutions between academia and industry and uncovered deep differences with corresponding reasons as suggestions for future work. Finally, we have noted several research challenges and emerging trends as future directions for pursuing advancements in SDN. We believe that these open issues must be addressed before the maturity of SDN can reach an acceptable level.

References

[1] D. Kreutz *et al.*, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

[2] Y. Jarraya, T. Madi, and M. Debbabi, "A survey and a layered taxonomy of software-defined networking," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 4, pp. 1955–1980, 4th Quart., 2014.

[3] F. Hu, Q. Hao, and K. Bao, "A survey on software-defined network and OpenFlow: From concept to implementation," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 4, pp. 2181–2206, 4th Quart., 2014.

[4] ONF. (2014). *SDN Architecture*. Accessed: Jun. 28, 2018. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf

[5] E. Haleplidis *et al.*, "Software-defined networking (SDN): Layers and architecture terminology," Internet Eng. Task Force, Fremont, CA, USA, RFC 7426, Jan. 2015. Accessed: Jun. 28, 2018. [Online]. Available: https://tools.ietf.org/rfc/rfc7426.txt

[6] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," in *Proc. ACM SIGCOMM*, 2008, pp. 69–74.

[7] R. Jain and S. Paul, "Network virtualization and software defined networking for cloud computing: A survey," *IEEE Commun. Mag.*, vol. 51, no. 11, pp. 24–31, Nov. 2013.

[8] Y. Li and M. Chen, "Software-defined network function virtualization: A survey," *IEEE Access*, vol. 3, pp. 2542–2553, 2015.

[9] N. Bizanis and F. A. Kuipers, "SDN and virtualization solutions for the Internet of Things: A survey," *IEEE Access*, vol. 4, pp. 5591–5606, 2016.

[10] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *Proc. ACM SIGCOMM*, Hong Kong, 2013, pp. 3–14.

[11] P. Patel *et al.*, "Ananta: Cloud scale load balancing," in *Proc. ACM SIGCOMM*, Hong Kong, 2013, pp. 207–218.

[12] S. Natarajan, A. Ramaiah, and M. Mathen, "A software defined cloud-gateway automation system using OpenFlow," in *Proc. IEEE 2nd Conf. CloudNet*, San Francisco, CA, USA, 2013, pp. 219–226.

[13] Juniper Networks. (2014). *Readiness, Benefits, and Barriers: An SDN Progress Report*. Accessed: Jun. 28, 2018. [Online]. Available: https://www.usebackpack.com/resources/7178/download?1451715494

[14] A. M. Johnson, Jr., and M. Malek, "Survey of software tools for evaluating reliability, availability, and serviceability," *ACM Comput. Surveys*, vol. 20, no. 4, pp. 227–269, 1988.

[15] P. Perešíni, M. Kuźniar, and D. Kostić, "Monocle: Dynamic, fine-grained data plane monitoring," in *Proc. ACM CoNEXT*, Heidelberg, Germany, 2015, Art. no. 32.

[16] C. Scott *et al.*, "Troubleshooting blackbox SDN control software with minimal causal sequences," in *Proc. ACM SIGCOMM*, Chicago, IL, USA, 2014, pp. 395–406.

[17] K. Mahajan, R. Poddar, M. Dhawan, and V. Mann, "JURY: Validating controller actions in software-defined networks," in *Proc. 46th IEEE/IFIP DSN*, Toulouse, France, 2016, pp. 109–120.

[18] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Passive and Active Network Measurement*. Cham, Switzerland: Springer, 2015, pp. 347–359.

[19] P. Kazemian *et al.*, "Real time network policy checking using header space analysis," in *Proc. 10th USENIX NSDI*, Lombard, IL, USA, 2013, pp. 99–112.

[20] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," in *Proc. 10th USENIX NSDI*, Lombard, IL, USA, 2013, pp. 15–27.

[21] L. Paradis and Q. Han, "A survey of fault management in wireless sensor networks," *J. Netw. Syst. Manag.*, vol. 15, no. 2, pp. 171–190, 2007.

[22] M. Yu, H. Mokhtar, and M. Merabti, "Fault management in wireless sensor networks," *IEEE Wireless Commun.*, vol. 14, no. 6, pp. 13–19, Dec. 2007.

[23] M. Canini, D. Venzano, P. Perešíni, D. Kostic, and J. Rexford, "A NICE way to test OpenFlow applications," in *Proc. 9th USENIX NSDI*, San Jose, CA, USA, 2012, pp. 127–140.

[24] N. Foster *et al.*, "Frenetic: A network programming language," in *Proc. ACM PLDI*, Tokyo, Japan, 2011, pp. 279–291.

[25] P. Sun *et al.*, "A network-state management service," in *Proc. ACM SIGCOMM*, Chicago, IL, USA, 2014, pp. 563–574.

[26] T. Koponen *et al.*, "Network virtualization in multi-tenant datacenters," in *Proc. 11th USENIX NSDI*, Seattle, WA, USA, 2014, pp. 203–216.

[27] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. 9th USENIX NSDI*, San Jose, CA, USA, 2012, pp. 113–126.

[28] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *Proc. 8th ACM CoNEXT*, Nice, France, 2012, pp. 241–252.

[29] A. Wang, Y. Guo, F. Hao, T. Lakshman, and S. Chen, "UMON: Flexible and fine grained traffic monitoring in open vSwitch," in *Proc. ACM CoNEXT*, Heidelberg, Germany, 2015, Art. no. 15.

[30] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *Proc. ACM SIGCOMM*, Florianópolis, Brazil, 2016, pp. 101–114.

[31] B. Chandrasekaran, B. Tschaen, and T. Benson, "Isolating and tolerating SDN application failures with LegoSDN," in *Proc. ACM SOSR*, Santa Clara, CA, USA, 2016, Art. no. 7.

[32] F. Botelho, A. Bessani, F. Ramos, and P. Ferreira, "SMaRtLight: A practical fault-tolerant SDN controller," *CoRR*, vol. abs/1407.6062, 2014. [Online]. Available: http://arxiv.org/abs/1407.6062

[33] I. Ahmad, S. Namal, M. Ylianttila, and A. Gurtov, "Security in software defined networks: A survey," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 4, pp. 2317–2346, 4th Quart., 2015.

[34] S. Scott-Hayward, S. Natarajan, and S. Sezer, "A survey of security in software defined networks," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 1, pp. 623–654, 1st Quart., 2015.

[35] S. T. Ali, V. Sivaraman, A. Radford, and S. Jha, "A survey of securing networks using software defined networking," *IEEE Trans. Rel.*, vol. 64, no. 3, pp. 1086–1097, Sep. 2015.

[36] A. Yassine, H. Rahimi, and S. Shirmohammadi, "Software defined network traffic measurement: Current trends and challenges," *IEEE Instrum. Meas. Mag.*, vol. 18, no. 2, pp. 42–50, Apr. 2015.

[37] J. Chen, J. Chen, F. Xu, M. Yin, and W. Zhang, "When software defined networks meet fault tolerance: A survey," in *Proc. Int. Conf. Algorithms Archit. Parallel Process.*, 2015, pp. 351–368.

[38] J. Qadir and O. Hasan, "Applying formal methods to networking: Theory, techniques, and applications," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 1, pp. 256–291, 1st Quart., 2015.

[39] T. Dargahi, A. Caponi, M. Ambrosin, G. Bianchi, and M. Conti, "A survey on the security of stateful SDN data planes," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1701–1725, 3rd Quart., 2017.

[40] ONF. *OpenFlow Switch Specification Version 1.5.1*. Accessed: Jun. 28, 2018. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf

[41] A. Doria *et al.*, "Forwarding and control element separation (ForCES) protocol specification," Internet Eng. Task Force, Fremont, CA, USA, RFC 5810, Mar. 2010. Accessed: Jun. 28, 2018. [Online]. Available: https://tools.ietf.org/rfc/rfc5810.txt

[42] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[43] X. Wen *et al.*, "SDNShield: Reconciliating configurable application permissions for SDN app markets," in *Proc. 46th IEEE/IFIP DSN*, Toulouse, France, 2016, pp. 121–132.

[44] H. Yin *et al.*, "SDNi: A message exchange protocol for software defined networks (SDNS) across multiple domains," IETF, Fremont, CA, USA, Internet Draft, Jun. 2012. Accessed: Jun. 28, 2018. [Online]. Available: https://tools.ietf.org/id/draft-yin-sdn-sdni-00.txt

[45] R. Skowyra, A. Lapets, A. Bestavros, and A. Kfoury, "A verification platform for SDN-enabled applications," in *Proc. IEEE IC2E*, Boston, MA, USA, 2014, pp. 337–342.

[46] T. Ball *et al.*, "VeriCon: Towards verifying controller programs in software-defined networks," in *Proc. 35th ACM PLDI*, Edinburgh, U.K., 2014, pp. 282–293.

[47] F. A. Lopes, M. Santos, R. Fidalgo, and S. Fernandes, "A software engineering perspective on SDN programmability," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 2, pp. 1255–1272, 2nd Quart., 2016.

[48] C. Monsanto *et al.*, "Composing software defined networks," in *Proc. 10th USENIX NSDI*, 2013, pp. 1–13.

[49] C. Prakash *et al.*, "PGA: Using graphs to express and automatically reconcile network policies," in *Proc. ACM SIGCOMM*, London, U.K., 2015, pp. 29–42.

[50] E. Haleplidis *et al.*, "Network programmability with ForCES," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 3, pp. 1423–1440, 3rd Quart., 2015.

[51] J. Halpern, "Forwarding and control element separation (ForCES) forwarding element model," Internet Eng. Task Force, Fremont, CA, USA, RFC 5812, Mar. 2010. Accessed: Jun. 28, 2018. [Online]. Available: https://tools.ietf.org/rfc/rfc5812.txt

[52] B. Pfaff and B. Davie, "The open vSwitch database management protocol," Internet Eng. Task Force, Fremont, CA, USA, RFC 7047, Dec. 2013. Accessed: Jun. 28, 2018. [Online]. Available: https://rfc-editor.org/rfc/rfc7047.txt

[53] R. Enns, M. Bjorklund, and J. Schoenwaelder, "Network configuration protocol (NETCONF)," Internet Eng. Task Force, Fremont, CA, USA, RFC 6241, Jun. 2011. Accessed: Jun. 28, 2018. [Online]. Available: https://tools.ietf.org/rfc/rfc6241.txt

[54] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proc. 2nd ACM HotSDN*, Hong Kong, 2013, pp. 127–132.

[55] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic, "A SOFT way for OpenFlow switch interoperability testing," in *Proc. ACM CoNEXT*, Nice, France, 2012, pp. 265–276.

[56] J. Yao, Z. Wang, X. Yin, X. Shiyz, and J. Wu, "Formal modeling and systematic black-box testing of SDN data plane," in *Proc. ICNP*, Raleigh, NC, USA, 2014, pp. 179–190.

[57] J. Sherry and S. Ratnasamy, "A survey of enterprise middlebox deployments," EECS Dept., Univ. California at Berkeley, Berkeley, CA, USA, Rep. UCB/EECS-2012-24, 2012. Accessed: Jun. 28, 2018. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-24.pdf

[58] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming platform-independent stateful OpenFlow applications inside the switch," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, 2014.

[59] S. Zhu, J. Bi, and C. Sun, "SFA: Stateful forwarding abstraction in SDN data plane," in *Proc. Open Netw. Summit (ONS)*, 2014, pp. 1–2.

[60] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for SDN," in *Proc. 3rd ACM HotSDN*, Chicago, IL, USA, 2014, pp. 61–66.

[61] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul, "FlowTags: Enforcing network-wide policies in the presence of dynamic middlebox actions," in *Proc. 2nd ACM HotSDN*, 2013, pp. 19–24.

[62] Z. A. Qazi *et al.*, "SIMPLE-fying middlebox policy enforcement using SDN," in *Proc. ACM SIGCOMM*, Hong Kong, 2013, pp. 27–38.

[63] G. G. Xie *et al.*, "On static reachability analysis of IP networks," in *Proc. 24th IEEE INFOCOM*, vol. 3. Miami, FL, USA, 2005, pp. 2170–2183.

[64] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Trans. Netw.*, vol. 24, no. 2, pp. 887–900, Apr. 2016.

[65] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *Proc. 12th USENIX NSDI*, Oakland, CA, USA, 2015, pp. 499–512.

[66] H. Yang and S. S. Lam, "Scalable verification of networks with packet transformers using atomic predicates," *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, pp. 2900–2915, Oct. 2017.

[67] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "SymNet: Scalable symbolic execution for modern networks," in *Proc. ACM SIGCOMM*, Florianópolis, Brazil, 2016, pp. 314–327.

[68] H. Zeng *et al.*, "Libra: Divide and conquer to verify forwarding tables in huge networks," in *Proc. 11th USENIX NSDI*, Seattle, WA, USA, 2014, pp. 87–99.

[69] J. C. Mogul *et al.*, "Corybantic: Towards the modular composition of SDN control programs," in *Proc. 12th ACM HotNets*, College Park, MD, USA, 2013, p. 1.

[70] K. Bu *et al.*, "Is every flow on the right track? Inspect SDN forwarding with RuleScope," in *Proc. IEEE INFOCOM*, San Francisco, CA, USA, 2016, pp. 1–9.

[71] K. Benton, L. J. Camp, and C. Small, "OpenFlow vulnerability assessment," in *Proc. 2nd ACM HotSDN*, Hong Kong, 2013, pp. 151–152.

[72] Z. Peng, "Towards rule enforcement verification for software defined networks," in *Proc. IEEE INFOCOM*, Atlanta, GA, USA, 2017, pp. 1–9.

[73] P. Zhang *et al.*, "Mind the gap: Monitoring the control-data plane consistency in software defined networks," in *Proc. ACM CoNEXT*, Irvine, CA, USA, 2016, pp. 19–33.

[74] J. Miserez, P. Bielik, A. El-Hassany, L. Vanbever, and M. Vechev, "SDNRacer: Detecting concurrency violations in software-defined networks," in *Proc. ACM SOSR*, Santa Barbara, CA, USA, 2015, p. 22.

[75] M. Kuzniar, P. Peresini, and D. Kostić, "Providing reliable FIB update acknowledgments in SDN," in *Proc. ACM CoNEXT*, Sydney, NSW, Australia, 2014, pp. 415–422.

[76] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An open framework for OpenFlow switch evaluation," in *Proc. Passive Active Netw. Meas.*, Vienna, Austria, 2012, pp. 85–95.

[77] M. Kuzniar, P. Peresini, and D. Kostic, "ProboScope: Data plane probe packet generation," EPFL, Lausanne, Switzerland, Rep. EPFL-REPORT-201824, 2014.

[78] E. Rojas, "From software-defined to human-defined networking: Challenges and opportunities," *IEEE Netw.*, vol. 32, no. 1, pp. 179–185, Jan./Feb. 2018.

[79] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.

[80] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Proc. 10th USENIX NSDI*, Lombard, IL, USA, 2013, pp. 227–240.

[81] A. Gember-Jacobson *et al.*, "OpenNF: Enabling innovation in network function control," in *Proc. ACM SIGCOMM*, Chicago, IL, USA, 2014, pp. 163–174.

[82] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proc. 14th USENIX NSDI*, Boston, MA, USA, 2017, pp. 97–112.

[83] A. Wundsam, D. Levin, S. Seetharaman, A. Feldmann, "OFRewind: Enabling record and replay troubleshooting for networks," in *Proc. USENIX ATC*, Portland, OR, USA, 2011, p. 29.

[84] R. W. Skowyra, A. Lapets, A. Bestavros, and A. Kfoury, "Verifiably-safe software-defined networks for CPS," in *Proc. 2nd ACM HiCoNS*, Philadelphia, PA, USA, 2013, pp. 101–110.

[85] M. Kuźniar, P. Perešíni, N. Vasić, M. Canini, and D. Kostić, "Automatic failure recovery for software-defined networks," in *Proc. 2nd ACM HotSDN*, Hong Kong, 2013, pp. 159–160.

[86] H. Hojjat, P. Rümmer, J. McClurg, P. Černỳ, and N. Foster, "Optimizing horn solvers for network repair," in *Proc. 16th IEEE FMCAD*, Mountain View, CA, USA, 2016, pp. 73–80.

[87] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, "Automated bug removal for software-defined networks," in *Proc. 14th USENIX NSDI*, Boston, MA, USA, 2017, pp. 719–733.

[88] J. Yao *et al.*, "Model based black-box testing of SDN applications," in *Proc. ACM CoNEXT Student Workshop*, Sydney, NSW, Australia, 2014, pp. 37–39.

[89] P. Porras *et al.*, "A security enforcement kernel for OpenFlow networks," in *Proc. 1st ACM HotSDN*, Helsinki, Finland, 2012, pp. 121–126.

[90] S. Natarajan, X. Huang, and T. Wolf, "Efficient conflict detection in flow-based virtualized networks," in *Proc. IEEE ICNC*, 2012, pp. 690–696.

[91] A. Guha, M. Reitblatt, and N. Foster, "Machine-verified network controllers," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 483–494, 2013.

[92] B. Heller *et al.*, "Leveraging SDN layering to systematically troubleshoot networks," in *Proc. 2nd ACM HotSDN*, Hong Kong, 2013, pp. 37–42.

[93] D. M. Volpano, X. Sun, and G. G. Xie, "Towards systematic detection and resolution of network control conflicts," in *Proc. 3rd ACM HotSDN*, Chicago, IL, USA, 2014, pp. 67–72.

[94] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi, "Tierless programming and reasoning for software-defined networks," in *Proc. 11th USENIX NSDI*, Seattle, WA, USA, 2014, pp. 519–531.

[95] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "FatTire: Declarative fault tolerance for software-defined networks," in *Proc. 2nd ACM HotSDN*, Hong Kong, 2013, pp. 109–114.

[96] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Apr. 2016.

[97] R. Durairajan, J. Sommers, and P. Barford, "Controller-agnostic SDN debugging," in *Proc. 10th ACM CoNEXT*, 2014, pp. 227–234.

[98] T. Nelson, D. Yu, Y. Li, R. Fonseca, and S. Krishnamurthi, "Simon: Scriptable interactive monitoring for SDNs," in *Proc. ACM SOSR*, Sydney, NSW, Australia, 2015, p. 19.

[99] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, "Automated network repair with meta provenance," in *Proc. 14th ACM HotNets*, Philadelphia, PA, USA, 2015, p. 26.

[100] W. Zhou, J. Croft, B. Liu, and M. Caesar, "NEAt: Network error autocorrect," in *Proc. ACM SOSR*, Renton, WA, USA, 2017, pp. 157–163.

[101] S. K. Fayaz and V. Sekar, "Testing stateful and dynamic data planes with FlowTest," in *Proc. 3rd ACM HotSDN.*, 2014, pp. 79–84.

[102] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar, "BUZZ: Testing context-dependent policies in stateful networks," in *Proc. 13th USENIX NSDI*, 2016, pp. 275–289.

[103] H. Zhang *et al.*, "Enabling layer 2 pathlet tracing through context encoding in software-defined networking," in *Proc. 3rd ACM HotSDN*, 2014, pp. 169–174.

[104] P. Tammana and R. Agarwal, and M. Lee, "Cherrypick: Tracing packet trajectory in software-defined datacenter networks," in *Proc. ACM SOSR*, 2015, p. 23.

[105] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker, "Compiling path queries," in *Proc. 13th USENIX NSDI*, 2016, pp. 207–222.

[106] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, "Where is the debugger for my software-defined network?" in *Proc. 1st ACM HotSDN*, 2012, pp. 55–60.

[107] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proc. 11th USENIX NSDI*, 2014, pp. 71–85.

[108] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "OpenTM: Traffic matrix estimator for OpenFlow networks," in *Proc. PAM*, 2010, pp. 201–210.

[109] C. Yu *et al.*, "FlowSense: Monitoring network utilization with zero measurement cost," in *Passive and Active Network Measurement*. Heidelberg, Germany: Springer, 2013, pp. 31–41.

[110] J. Yang *et al.*, "Rethinking the design of OpenFlow switch counters," in *Proc. ACM SIGCOMM*, 2016, pp. 589–590.

[111] A. R. Curtis *et al.*, "DevoFlow: Scaling flow management for high-performance networks," in *Proc. ACM SIGCOMM*, Toronto, ON, Canada, 2011, pp. 254–265.

[112] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proc. ACM SIGCOMM*, 2016, pp. 129–143.

[113] Z. Hu and J. Luo, "Cracking network monitoring in DCNs with SDN," in *Proc. IEEE INFOCOM*, 2015, pp. 199–207.

[114] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "OpenSample: A low-latency, sampling-based measurement platform for commodity SDN," in *Proc. 34th IEEE ICDCS*, 2014, pp. 228–237.

[115] W. Han *et al.*, "State-aware network access management for software-defined networks," in *Proc. 21st ACM SACMAT*, 2016, pp. 1–11.

[116] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with OpenSketch," in *Proc. 10th USENIX NSDI*, 2013, pp. 29–42.

[117] Q. Huang *et al.*, "SketchVisor: Robust network measurement for so ware packet processing," in *Proc. ACM SIGCOMM*, 2017, pp. 113–126.

[118] J. Rasley *et al.*, "Planck: Millisecond-scale monitoring and control for commodity networks," in *Proc. ACM SIGCOMM*, 2014, pp. 407–418.

[119] Y. Zhu *et al.*, "Packet-level telemetry in large datacenter networks," in *Proc. ACM SIGCOMM*, 2015, pp. 479–491.

[120] P. Sun, M. Yu, M. J. Freedman, J. Rexford, and D. Walker, "HONE: Joint host-network traffic management in software-defined networks," *J. Netw. Syst. Manag.*, vol. 23, no. 2, pp. 374–399, 2015.

[121] H. Chen *et al.*, "Felix: Implementing traffic measurement on end hosts using program analysis," in *Proc. ACM SOSR*, 2016, Art. no. 14.

[122] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures," in *Proc. 3rd ACM SaftConfig*, Chicago, IL, USA, 2010, pp. 37–44.

[123] H. Mai *et al.*, "Debugging the data plane with anteater," in *Proc. ACM SIGCOMM*, 2011, pp. 290–301.

[124] Y. Xu, Y. Liu, R. Singh, and S. Tao, "Identifying SDN state inconsistency in OpenStack," in *Proc. ACM SOSR*, 2015, p. 11.

[125] A. Horn, A. Kheradmand, and M. R. Prasad, "Delta-net: Real-time network verification using atoms," in *Proc. 14th USENIX NSDI.*, 2017, pp. 735–749.

[126] L. Ryzhyk *et al.*, "Correct by construction networks using stepwise refinement," in *Proc. 14th USENIX NSDI*, 2017, pp. 683–698.

[127] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying isolation properties in the presence of middle-boxes," *CoRR*, vol. abs/1409.7687, 2014. [Online]. Available: http://arxiv.org/abs/1409.7687

[128] X. Wen *et al.*, "RuleScope: Inspecting forwarding faults for software-defined networking," *IEEE/ACM Trans. Netw.*, vol. 25, no. 4, pp. 2347–2360, Aug. 2017.

[129] K. Agarwal, E. Rozner, C. Dixon, and J. Carter, "SDN traceroute: Tracing SDN forwarding without changing network behavior," in *Proc. 3rd ACM HotSDN*, 2014, pp. 145–150.

[130] S. Narayana, J. Rexford, and D. Walker, "Compiling path queries in software-defined networks," in *Proc. 3rd ACM HotSDN*, 2014, pp. 181–186.

[131] P. Tammana, R. Agarwal, and M. Lee, "Simplifying datacenter network debugging with pathDump," in *Proc. 12th USENIX OSDI*, 2016, pp. 233–248.

[132] Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, "Answering why-not queries in software-defined networks with negative provenance," in *Proc. 12th ACM HotNets*, 2013, p. 3.

[133] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo, "Diagnosing missing events in distributed systems with negative provenance," in *Proc. ACM SIGCOMM*, 2014, pp. 383–394.

[134] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, "The good, the bad, and the differences: Better network diagnostics with differential provenance," in *Proc. ACM SIGCOMM*, 2016, pp. 115–128.

[135] *OFTest OpenFlow Switch Testing Framework*. Accessed: Jun. 28, 2018. [Online]. Available: http://www.projectfloodlight.org/oftest/

[136] M. Kuzniar, M. Canini, and D. Kostic, "OFTEN testing OpenFlow networks," in *Proc. Eur. Workshop SDN*, 2012, pp. 54–60.

[137] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Model checking invariant security properties in OpenFlow," in *Proc. IEEE ICC*, 2013, pp. 1974–1979.

[138] B. L. A. Batista, G. A. L. de Campos, and M. P. Fernandez, "Flow-based conflict detection in OpenFlow networks using first-order logic," in *Proc. IEEE ISCC*, 2014, pp. 1–6.

[139] J. Wang *et al.*, "Towards a security-enhanced firewall application for OpenFlow networks," in *Cyberspace Safety Security*. Cham, Switzerland: Springer, 2013, pp. 92–103.

[140] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "FLOWGUARD: Building robust firewalls for software-defined networks," in *Proc. 3rd ACM HotSDN*, 2014, pp. 97–102.

[141] A. El-Hassany, J. Miserez, P. Bielik, L. Vanbever, and M. Vechev, "SDNRacer: Concurrency analysis for software-defined networks," in *Proc. ACM PLDI*, 2016, pp. 402–415.

[142] R. May, A. El-Hassany, L. Vanbever, and M. Vechev, "BigBug: Practical concurrency analysis for SDN," in *Proc. ACM SOSR*, 2017, pp. 88–94.

[143] R. C. Scott, A. Wundsam, K. Zarifis, and S. Shenker, "What, where, and when: Software fault localization for SDN," Elect. Eng. Comput. Sci. Dept., Univ. California at Berkeley, Berkeley, CA, USA, Rep. UCB/EECS-2012-178, Jul. 2012. Accessed: Jun. 28, 2018. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-178.pdf

[144] C. Scott *et al.*, "How did we get into this mess? Isolating fault-inducing inputs to SDN control software," Elect. Eng. Comput. Sci. Dept., Univ. California at Berkeley, Berkeley, CA, USA, Rep. UCB/EECS-2013-8, 2013. Accessed: Jun. 28, 2018. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-8.pdf

[145] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *Proc. 2nd USENIX Hot-ICE*, 2012, pp. 1–6.

[146] M. Jarschel, F. Lehrieder, Z. Magyari, and R. Pries, "A flexible OpenFlow-controller benchmark," in *Proc. Eur. Workshop SDN*, 2012, pp. 48–53.

[147] N. Laurent, S. Vissicchio, and M. Canini, "SDLoad: An extensible framework for SDN workload generation," in *Proc. 3rd ACM HotSDN*, 2014, pp. 215–216.

[148] M. Jarschel, C. Metter, T. Zinner, S. Gebert, and P. Tran-Gia, "OFCProbe: A platform-independent tool for OpenFlow controller analysis," in *Proc. IEEE ICCE*, 2014, pp. 182–187.

[149] Z. K. Khattak, M. Awais, and A. Iqbal, "Performance evaluation of OpenDaylight SDN controller," in *Proc. 20th IEEE ICPADS*, 2014, pp. 671–676.

[150] P. Perešíni and M. Canini, "Is your OpenFlow application correct?" in *Proc. ACM CoNEXT Student Workshop*, 2011, p. 18.

[151] M. Canini, D. Kostic, J. Rexford, and D. Venzano, "Automating the testing of OpenFlow applications," in *Proc. 1st Int. Workshop Rigorous Protocol Eng. (WRiPE)*, 2011, pp. 1–6.

[152] D. Sethi, S. Narayana, and S. Malik, "Abstractions for model checking SDN controllers," in *Proc. 13th IEEE FMCAD*, Portland, OR, USA, 2013, pp. 145–148.

[153] T. Nelson, A. D. Ferguson, and S. Krishnamurthi, "Static differential program analysis for software-defined networks," in *Proc. Int. Symp. Formal Methods*, 2015, pp. 395–413.

[154] R. Beckett *et al.*, "An assertion language for debugging SDN applications," in *Proc. 3rd ACM HotSDN*, 2014, pp. 91–96.

[155] I. Pelle, T. Lévai, F. Németh, and A. Gulyás, "One tool to rule them all: A modular troubleshooting framework for SDN (and other) networks," in *Proc. ACM SOSR*, 2015, p. 24.

[156] T. Lévai, I. Pelle, F. Németh, and A. Gulyás, "EPOXIDE: A modular prototype for SDN troubleshooting," in *Proc. ACM SIGCOMM*, 2015, pp. 359–360.

[157] A. Ko and B. Myers, "Debugging reinvented," in *Proc. ACM/IEEE ICSE*, 2008, pp. 301–310.

[158] B. H. Sigelman *et al.*, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Mountain View, CA, USA, Rep. dapper-2010-1, 2010.

[159] A. Nandi, A. Mandal, S. Atreja, G. B. Dasgupta, and S. Bhattacharya, "Anomaly detection using program control flow graph mining from execution logs," in *Proc. ACM KDD*, 2016, pp. 215–224.

[160] X. Yu *et al.*, "CloudSeer: Workflow monitoring of cloud infrastructures via interleaved logs," in *Proc. ACM ASPLOS*, vol. 50, 2016, pp. 489–502.

[161] R. Sherwood *et al.*, "FlowVisor: A network virtualization layer," OpenFlow Switch Consortium, Rep. OPENFLOW-TR-2009-1, pp. 1–13, 2009. [Online]. Available: https://pdfs.semanticscholar.org/64f3/a81fff495ac336dccdd63136d451852eb1c9.pdf

[162] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *Proc. ACM POPL*, vol. 47, 2012, pp. 217–230.

[163] R. Majumdar, S. D. Tetali, and Z. Wang, "Kuai: A model checker for software-defined networks," in *Proc. 14th IEEE FMCAD*, 2014, pp. 163–170.

[164] M. Gupta, J. Sommers, and P. Barford, "Fast, accurate simulation for SDN prototyping," in *Proc. 2nd ACM HotSDN*, 2013, pp. 31–36.

[165] Y. Zhang, N. Beheshti, and R. Manghirmalani, "NetRevert: Rollback recovery in SDN," in *Proc. 3rd ACM HotSDN*, 2014, pp. 231–232.

[166] T. Sasaki, A. Perrig, and D. E. Asoni, "Control-plane isolation and recovery for a secure SDN architecture," in *Proc. IEEE NetSoft*, 2016, pp. 459–464.

[167] A. Blenk, A. Basta, M. Reisslein, and W. Kellerer, "Survey on network virtualization hypervisors for software defined networking," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 1, pp. 655–685, 1st Quart., 2016.

[168] S. Gutz, A. Story, C. Schlesinger, and N. Foster, "Splendid isolation: A slice abstraction for software-defined networks," in *Proc. 1st ACM HotSDN*, 2012, pp. 79–84.

[169] M. Canini *et al.*, "STN: A robust and distributed SDN control plane," in *Proc. Open Netw. Summit*, 2014, pp. 1–2.

[170] X. Wen *et al.*, "Compiling minimum incremental update for modular SDN languages," in *Proc. 3rd ACM HotSDN*, 2014, pp. 193–198.

[171] C. J. Anderson *et al.*, "NetKAT: Semantic foundations for networks," in *Proc. ACM POPL*, vol. 49, 2014, pp. 113–126.

[172] X. Wen *et al.*, "RuleTris: Minimizing rule update latency for TCAM-based SDN switches," in *Proc. IEEE ICDCS*, 2016, pp. 179–188.

[173] A. Dixit, K. Kogan, and P. Eugster, "Composing heterogeneous SDN controllers with flowbricks," in *Proc. IEEE ICNP*, 2014, pp. 287–292.

[174] X. Jin, J. Rexford, and D. Walker, "Incremental update for a compositional SDN hypervisor," in *Proc. 3rd ACM HotSDN*, 2014, pp. 187–192.

[175] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A compositional hypervisor for software-defined networks," in *Proc. 12th USENIX NSDI*, 2015, pp. 87–101.

[176] H. Pan, G. Xie, P. He, Z. Li, and L. Mathy, "Action computation for compositional software-defined networking," in *Proc. IFIP Netw.*, 2016, pp. 19–27.

[177] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory networking: An API for application control of SDNs," in *Proc. ACM SIGCOMM*, 2013, pp. 327–338.

[178] A. AuYoung *et al.*, "Democratic resolution of resource conflicts between SDN control programs," in *Proc. 10th ACM CoNEXT*, 2014, pp. 391–402.

[179] H. Kim *et al.*, "CORONET: Fault tolerance for software defined networks," in *Proc. 20th IEEE ICNP*, 2012, pp. 1–2.

[180] K. He *et al.*, "Latency in software defined networks: Measurements and mitigation techniques," in *Proc. ACM SIGMETRICS*, vol. 43, 2015, pp. 435–436.

[181] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Enabling fast failure recovery in OpenFlow networks," in *Proc. IEEE DRCN*, 2011, pp. 164–171.

[182] D. Brungard, M. Betts, S. Ueno, B. Niven-Jenkins, and N. Sprecher, "Requirements of an MPLS transport profile," Internet Eng. Task Force, Fremont, CA, USA, RFC 5654, Sep. 2009. Accessed: Jun. 28, 2018. [Online]. Available: https://tools.ietf.org/rfc/rfc5654.txt

[183] J. Li, J. Hyun, J.-H. Yoo, S. Baik, and J. W.-K. Hong, "Scalable failover method for data center networks using OpenFlow," in *Proc. IEEE NOMS*, 2014, pp. 1–6.

[184] ONF. *OpenFlow Switch Specification Version 1.1.0 Implemented*. Accessed: Jun. 28, 2018. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf

[185] R. Pujar and C. Icaro. 2016. *Path Protection and Failover Strategies in SDN Networks*. Accessed: Jun. 6, 2018. [Online]. Available: http://events17.linuxfoundation.org/sites/events/files/slides/Path%20protection%20and%20failover%20strategies%20in%20SDN%20networks.pdf

[186] N. Sahri and K. Okamura, "Fast failover mechanism for software defined networking: OpenFlow based," in *Proc. 9th Int. Conf. Future Internet Technol.*, 2014, p. 16.

[187] H. Li, Q. Li, Y. Jiang, T. Zhang, and L. Wang, "A declarative failure recovery system in software defined networks," in *Proc. IEEE ICC*, 2016, pp. 1–6.

[188] D. Katz and D. Ward, "Bidirectional forwarding detection (BFD)," Internet Eng. Task Force, Fremont, CA, USA, RFC 5880, Jun. 2010. Accessed: Jun. 6, 2018. [Online]. Available: https://tools.ietf.org/rfc/rfc5880.txt

[189] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "OpenFlow: Meeting carrier-grade recovery requirements," *Comput. Commun.*, vol. 36, no. 6, pp. 656–665, 2013.

[190] N. L. Van Adrichem, B. J. Van Asten, and F. A. Kuipers, "Fast recovery in software-defined networks," in *Proc. 3rd IEEE Eur. Workshop SDN*, 2014, pp. 61–66.

[191] B. Stephens, A. L. Cox, and S. Rixner, "Scalable multi-failure fast failover via forwarding table compression," in *Proc. ACM SOSR*, 2016, Art. no. 9.

[192] M. Borokhovich, L. Schiff, and S. Schmid, "Provable data plane connectivity with local fast failover: Introducing OpenFlow graph algorithms," in *Proc. 3rd ACM HotSDN*, 2014, pp. 121–126.

[193] L. Schiff, M. Borokhovich, and S. Schmid, "Reclaiming the brain: Useful OpenFlow functions in the data plane," in *Proc. 13th ACM HotNets*, 2014, p. 7.

[194] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "In-band control, queuing, and failure recovery functionalities for OpenFlow," *IEEE Netw.*, vol. 30, no. 1, pp. 106–112, Jan./Feb. 2016.

[195] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Fast failure recovery for in-band OpenFlow networks," in *Proc. 9th IEEE Design Rel. Commun. Netw. (DRCN)*, 2013, pp. 52–59.

[196] Y. Hu *et al.*, "Control traffic protection in software-defined networks," in *Proc. IEEE Globecom*, Austin, TX, USA, 2014, pp. 1878–1883.

[197] M. Obadia, M. Bouet, J. Leguay, K. Phemius, and L. Iannone, "Failover mechanisms for distributed SDN controllers," in *Proc. IEEE Conf. NOF*, 2014, pp. 1–6.

[198] N. Beheshti and Y. Zhang, "Fast failover for control traffic in software-defined networks," in *Proc. IEEE GLOBECOM*, Anaheim, CA, USA, 2012, pp. 2665–2670.

[199] A. Tootoonchian and Y. Ganjali, "HyperFlow: A distributed control plane for OpenFlow," in *Proc. USENIX INM/WREN*, 2010, p. 3.

[200] H. Li, P. Li, S. Guo, and A. Nayak, "Byzantine-resilient secure software-defined networks with multiple controllers in cloud," *IEEE Trans. Cloud Comput.*, vol. 2, no. 4, pp. 436–447, Oct./Dec. 2014.

[201] N. Katta, H. Zhang, M. Freedman, and J. Rexford, "Ravana: Controller fault-tolerance in software-defined networking," in *Proc. ACM SOSR*, 2015, p. 4.

[202] Y. Zhang, N. Beheshti, and M. Tatipamula, "On resilience of split-architecture networks," in *Proc. IEEE Globecom*, Kathmandu, Nepal, 2011, pp. 1–6.

[203] Y. Hu, W. Wendong, X. Gong, X. Que, and C. Shiduan, "Reliability-aware controller placement for software-defined networks," in *Proc. IFIP/IEEE IM*, Ghent, Belgium, 2013, pp. 672–675.

[204] M. Guo and P. Bhattacharya, "Controller placement for improving resilience of software-defined networks," in *Proc. 4th IEEE ICNDC*, Los Angeles, CA, USA, 2013, pp. 23–27.

[205] L. F. Müller *et al.*, "Survivor: An enhanced controller placement strategy for improving SDN survivability," in *Proc. IEEE Globecom*, Austin, TX, USA, 2014, pp. 1909–1915.

[206] M. F. Bari *et al.*, "Dynamic controller provisioning in software defined networks," in *Proc. 9th IEEE Int. CNSM*, Zürich, Switzerland, 2013, pp. 18–25.

[207] F. J. Ros and P. M. Ruiz, "Five nines of southbound reliability in software-defined networks," in *Proc. 3rd ACM HotSDN*, Chicago, IL, USA, 2014, pp. 31–36.

[208] D. Hock *et al.*, "Pareto-optimal resilient controller placement in SDN-based core networks," in *Proc. IEEE ITC*, Shanghai, China, 2013, pp. 1–9.

[209] S. Lange *et al.*, "Heuristic approaches to the controller placement problem in large scale SDN networks," *IEEE Trans. Netw. Service Manag.*, vol. 12, no. 1, pp. 4–17, Mar. 2015.

[210] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *Proc. 1st ACM HotSDN*, Helsinki, Finland, 2012, pp. 7–12.

[211] OpenDayLight. *Cardinal*. Accessed: Jun. 28, 2018. [Online]. Available: https://wiki.opendaylight.org/view/Cardinal:Main

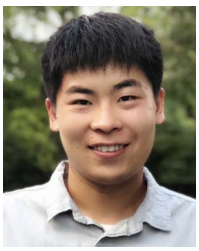
[212] OpenDayLight. *Centinel*. Accessed: Jun. 28, 2018. [Online]. Available: https://wiki.opendaylight.org/view/Centinel:Main

[213] OpenDayLight. *Project Proposals:Time Series Data Repository*. Accessed: Jun. 28, 2018. [Online]. Available: https://wiki.opendaylight.org/view/Project_Proposals:Time_Series_Data_Repository

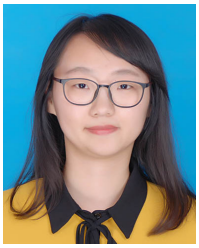
[214] ONOS. *OPEN-TAM: Traffic Analysis and Monitoring*. Accessed: Jun. 28, 2018. [Online]. Available: https://wiki.onosproject.org/display/ONOS/OPEN-TAM%3A+Traffic+Analysis+and+Monitoring

[215] ONOS. *Fault Management*. Accessed: Jun. 28, 2018. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Fault+Management
[216] ONOS. *Composition Mode*. Accessed: Jun. 28, 2018. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Composition+Mode
[217] ONOS. *Network TroubleShooting Module*. Accessed: Jun. 28, 2018. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Network+TroubleShooting+Module
[218] ONOS. *Network Artificial Intelligence*. Accessed: Jun. 28, 2018. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Network+Artificial+Intelligence
[219] OpenContrail. *A SDN Analytics Interface*. Accessed: Jun. 28, 2018. [Online]. Available: http://www.opencontrail.org/sandesh-a-sdn-analytics-interface/
[220] J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a model-driven SDN controller architecture," in *Proc. IEEE 15th Int. Symp. World Wireless Mobile Multimedia Networks (WoWMoM)*, Sydney, NSW, Australia, 2014, pp. 1–6.
[221] N. Gude *et al.*, "NOX: Towards an operating system for networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, 2008.
[222] J. Mccauley. (2014). *Pox: A Python-Based OpenFlow Controller*. Accessed: Sep. 11, 2018. [Online]. Available: https://noxrepo.github.io/pox-doc/html/
[223] T. Koponen *et al.*, "Onix: A distributed control platform for large-scale production networks," in *Proc. 9th USENIX Symp. OSDI*, vol. 10, 2010, pp. 1–6.
[224] D. Erickson, "The beacon OpenFlow controller," in *Proc. 2nd ACM HotSDN*, 2013, pp. 13–18.
[225] Big Switch. *FloodLight Is an Open SDN Controller*. Accessed: Jun. 28, 2018. [Online]. Available: http://www.projectfloodlight.org/floodlight/
[226] P. Berde *et al.*, "ONOS: Towards an open, distributed SDN OS," in *Proc. 3rd ACM HotSDN*, 2014, pp. 1–6.
[227] NEC. *Trema: Full-Stack OpenFlow Framework in Ruby and C*. Accessed: Jun. 28, 2018. [Online]. Available: https://trema.github.io/trema/
[228] NTT. *Ryu SDN Framework: Build SDN Agilely*. Accessed: Jun. 6, 2018. [Online]. Available: https://osrg.github.io/ryu/
[229] *OpenContrail: An Open-Source Network Virtualization Platform for the Cloud*, Cisco, San Jose, CA, USA, Accessed: Jun. 6, 2018. [Online]. Available: http://www.opencontrail.org/
[230] OpenDaylight. *OpenDaylight: A Linux Foundation Collaborative Project*. Accessed: Jun. 6, 2018. [Online]. Available: http://www.opendaylight.org
[231] ETRI. *OpenIRIS: The Recursive SDN Openflow Controller*. Accessed: Jun. 6, 2018. [Online]. Available: http://openiris.etri.re.kr/
[232] KulCloud. *OpenMUL*. Accessed: Jun. 6, 2018. [Online]. Available: http://www.openmul.org/
[233] S. H. Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proc. 1st ACM HotSDN*, 2012, pp. 19–24.
[234] Cartesian. (2017). *The Future of Networks: Dealing With Transformation in a Virtualized World*. Accessed: Jun. 6, 2018. [Online]. Available: https://content.cartesian.com/the-future-of-networks-report
[235] V. Yazıcı, U. C. Kozat, and M. O. Sunay, "A new control plane for 5G network architecture with a case study on unified handoff, mobility, and routing management," *IEEE Commun. Mag.*, vol. 52, no. 11, pp. 76–85, Nov. 2014.
[236] D. Levin, M. Canini, S. Schmid, and A. Feldmann, "Incremental SDN deployment in enterprise networks," in *Proc. ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, 2013, pp. 473–474.
[237] H. Xu *et al.*, "Incremental deployment and throughput maximization routing for a hybrid SDN," *IEEE/ACM Trans. Netw.*, vol. 25, no. 3, pp. 1861–1875, Jun. 2017.
[238] D. Hancock and J. van der Merwe, "HyPer4: Using P4 to virtualize the programmable data plane," in *Proc. 12th ACM CoNEXT*, 2016, pp. 35–49.
[239] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data plane performance diagnosis of TCP," in *Proc. ACM SOSR*, 2017, pp. 61–74.
[240] S. Narayana *et al.*, "Language-directed hardware design for network performance monitoring," in *Proc. ACM SIGCOMM*, 2017, pp. 85–98.
[241] ONF. (2016). *Special Report: OpenFlow and SDN—State of the Union*. Accessed: Jun. 28, 2018. [Online]. Available: http://www.opennetworking.org/wp-content/uploads/2013/05/Special-Report-OpenFlow-and-SDN-State-of-the-Union-B.pdf
[242] R. Pham, Y. Stoliar, and K. Schneider, "Automatically recommending test code examples to inexperienced developers," in *Proc. 10th ACM ESEC/FSE*, 2015, pp. 890–893.
[243] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Commun. Mag.*, vol. 51, no. 2, pp. 136–141, Feb. 2013.
[244] M. Karakus and A. Durresi, "A survey: Control plane scalability issues and approaches in software-defined networking (SDN)," *Comput. Netw.*, vol. 112, pp. 279–293, Jan. 2017.
[245] Z. M. Fadlullah *et al.*, "State-of-the-art deep learning: Evolving machine intelligence toward tomorrow's intelligent network traffic control systems," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2432–2455, 4th Quart., 2017.
[246] A. Mestres *et al.*, "Knowledge-defined networking," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 47, no. 3, pp. 2–10, 2017.

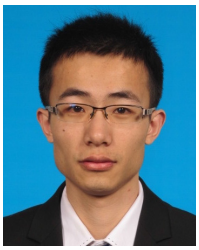
Yinbo Yu received the B.E. degree in electronic information engineering from Wuhan University, Wuhan, China, in 2014, where he is currently pursuing the Ph.D. degree with the School of Electronic Information. He is also a visiting Ph.D. student with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, USA. His research interests include SDN, NFV, cellular network, and networking security and measurement.



Xing Li received the B.E. degree in software engineering from Shandong University, Jinan, China, in 2016. He is currently pursuing the Ph.D. degree with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. His research interests include SDN, NFV, and cyberspace security.



Xue Leng (S'18) received the B.S. degree in computer science and technology from Harbin Engineering University, Harbin, China, in 2015. She is currently pursuing the Ph.D. degree major in computer science and technology with Zhejiang University, Hangzhou, China. Her research interests are SDN, NFV, and 5G protocol verification. She is a Student Member of CCF.



Libin Song received the B.S. degree in automation from Tsinghua University, Beijing, China, in 2015 and the M.Sc. degree in computer science from Northwestern University, Evanston, IL, USA, in 2017. He is currently a Software Engineer with TuSimple, San Diego, CA, USA. His research interests span on the area of distributed systems, networking and security, with a current focus on computing resources orchestration in enterprise data center.

Kai Bu (M'17) received the B.Sc. and M.Sc. degrees in computer science from the Nanjing University of Posts and Telecommunications, Nanjing, China, in 2006 and 2009, respectively, and the Ph.D. degree in computer science from Hong Kong Polytechnic University, Hong Kong, in 2013. He is currently an Assistant Professor with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. His research interests include networking and security. He was a recipient of the Best Paper Award of IEEE/IFIP EUC 2011 and the Best Paper Nominees of IEEE ICDCS 2016. He is a member of ACM and CCF.

Yan Chen (F'17) received the Ph.D. degree in computer science from the University of California at Berkeley, Berkeley, CA, USA, in 2003. He is currently a Professor with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, USA. He has over 10 000 Google Scholar citation with an H-index of 49. His research interests include network security, measurement, and diagnosis for large-scale networks and distributed systems. He was a recipient of the Department of Energy Early CAREER Award in 2005, the Department of Defense Young Investigator Award in 2007, the Best Paper nomination in ACM SIGCOMM 2010, and the Most Influential Paper Award in ASPLOS 2018.

Jianfeng Yang received the bachelor's, master's, and Ph.D. degrees in information and communication engineering from Wuhan University, China, in 1998, 2002, and 2009, respectively, where he is currently an Associate Professor. He was a Visiting Scholar with Intel Company in 2012 and Northwestern University from 2015 to 2016. His research interests are in security and measurement for networking, edge computing, and high-reliability real-time wireless communication.

Liang Zhang received the Ph.D. degree in circuit and system from Southeast University, Nanjing, China, in 2010. He is currently a Leader Research Engineer with Huawei Technologies Company Ltd. He is currently leading a big data analysis team, focus on the intelligent fault analysis, network health evaluation, and network automation.

Kang Cheng received the Ph.D. degree in control theory and engineering from Southeast University, Nanjing, China, in 2013. He is currently a Senior Research Engineer with Huawei Technologies Company Ltd. His research interests include network fault diagnosis and optimization, optimal control, and machine learning.

Xin Xiao received the Ph.D. degree in control and computer engineering from the Politecnico di Torino, Turin, Italy, in 2016. She is currently a Senior Research Engineer with Huawei Technologies Company Ltd. Her research interests include data mining, network fault diagnosis and optimization, time series analysis, and machine learning.