Generic Validation Criteria and Methodologies for SDN Applications

Ying-Dar Lin, *Fellow, IEEE*, Yu-Kuen Lai, *Senior Member, IEEE*, Yung-Liang Tsou, Yuan-Cheng Lai, *Member, IEEE*, En-Cheng Liou, and Yita Chiang

***Abstract*—Programmable control plane in software-defined networking (SDN), plays an essential role in the SDN architecture. The network function provided by the specialized hardware in a legacy network can be created in the form of software-based "SDN application" running on the controllers to manipulate entire network configurations. Therefore, the risk of having software bugs and errors in the SDN applications may threaten the normal operations of SDN networks. This paper presents systematic validation criteria and test cases based on the proposed novel methodologies for SDN application testing. The test framework can perform testbed build-up, generate desired packet sequences, and analyze results automatically. According to the results of a generic test suite, several issues are unveiled in the application under test (AUT). Some AUTs, which need to check all the incoming packets from OpenFlow switches, fail to meet the test criteria of burst packet-in and flow self-recycling. For most of the applications based on the Ryu controller, the evaluation results reveal that some are unable to recycle flow entries after they are unloaded. It is recommended that all flows populated by SDN applications must have timeout value specified to prevent unnecessary entries kept in the flow table.**

***Index Terms*—OpenFlow, performance evaluation, software defined network (SDN), system performance, testing, validation.**

I. Introduction

THE operation of software-defined networking (SDN) virtualizes the control plane by decoupling itself from the data plane in legacy network devices. In the virtualized control plane, controllers can maintain a global view of the entire network and dynamically manage network devices. Therefore, specific network functions provided by specialized hardware can be created in the form of software-based "SDN applications" and executed on top of a controller. Based on the OpenFlow [1] protocol standard, such "SDN applications" can be used to configure the switches in a data plane providing flexible and cost-effective benefits for network management and function deployment.

The controller, based on the operation of SDN applications, can perform various network functions and decide to manipulate switch behavior by adding flow entries as packets arrive. However, unexpected behaviors may occur caused by software bugs in the applications that jeopardize network operation. There are thus two major issues that need to be considered when programming SDN applications. The first is the issue of the correctness of functionality. It is crucial to avoid incorrect or unexpected network functions caused by the improper configuration of an application running on a controller. For example, one primary requirement for forwarding applications is to make sure that no packet carrying a control message is dropped in the controller. The second issue is related to the consumption of resources. Computation and storage resources in controllers and switches are limited. Therefore, if applications drain too many such resources, an SDN networks may eventually crash. As a result, the process of SDN applications development needs a set of well defined criteria and efficient ways to validate the integrity of applications so as to achieve high operational quality and reliability.

Open network foundation (ONF) is the most well-known organization which developed OpenFlow, a major SDN protocol standard, and benchmarking methodology for SDN environments [2]. Currently, available proposals are limited to controller benchmarking [3], switch conformance and interoperability benchmarking [4]. There is to date no benchmarking for SDN applications available. Recent studies are focusing on white-box testing [5], [6] for developers to identify bugs, concurrency violations [7], and verify the validity of flow rules [8] in applications. Typically, a white-box testing is commonly referring to the test methodology which is developed based on the DUT's internal design. On the contrary, black-box testing is applied where only the top-level interfaces and basic functionalities of the DUT are known. The complete testing is designed without knowing the DUT's internal architecture and implementation detail.

Although white-box testing can provide a thorough examination of applications, most of the source codes of these applications are not publicly available. Thus, the scale of testing may be restricted as a result of unavailability of source codes. Black-box testing methods can be applied to all applications.

Manuscript received July 27, 2018; revised November 14, 2018, February 17, 2019, and May 12, 2019; accepted May 23, 2019. This work was supported in part by the Ministry of Science and Technology (MoST), Taiwan, in part by Estinet Technologies Inc., and in part by Chunghua Telecom. *(Corresponding author: Yu-Kuen Lai.)*

Y.-D. Lin and Y.-L. Tsou are with the Department of Computer Science, National Chiao Tung University, Hsin-Chu 300, Taiwan, R.O.C (e-mail: ydlin@cs.nctu.edu.tw; yungliang1992@gmail.com).

Y.-K. Lai is with the Department of Electrical Engineering, Chung-Yuan Christian University, Chung-Li 32023, Taiwan, R.O.C (e-mail: ylai@cnsrl.cycu.edu.tw).

Y.-C. Lai is with the Department of Information Management, National Taiwan University of Science and Technology, Taipei 10607, Taiwan, R.O.C (e-mail: laiyc@cs.ntust.edu.tw).

E.-C. Liou and Y. Chiang are with the Network Benchmarking Lab, National National Chiao Tung University, Taoyuan 32023, Taiwan, R.O.C (e-mail: ecliou@nbl.org.tw; ytchiang@nbl.org.tw).

Digital Object Identifier 10.1109/JSYST.2019.2921599

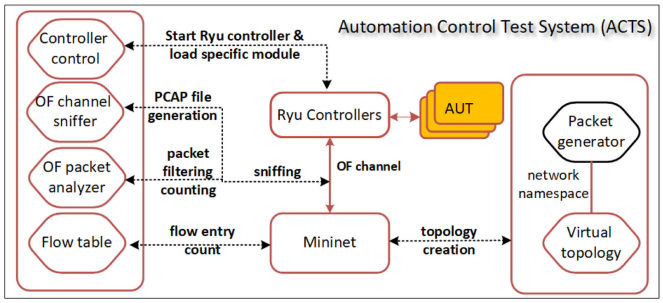


Fig. 1. Proposed architecture framework for the validation of SDN applications in the automation control test system (ACTS) [12]. The "*packet generator*" generates the network traffic and packets are injected into the *"virtual topology."* The "*OF channel sniffer*" and "*OF packet analyzer*" components are used to sniff and analyze the network traffic communicated between the controller and switches. The "*flow table*" component is designed to collect flow entries populated on switches for further analysis. Mininet is used to create the entire virtual network topology.

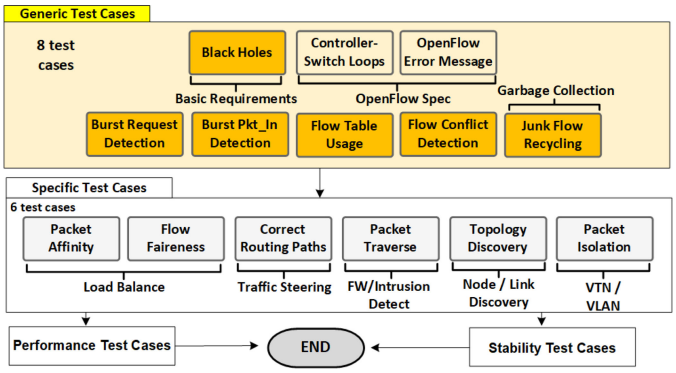


Fig. 2. The proposed flowchart, framework of methodologies, and systematic validation criteria for the application validation. The framework consists of three layers of four major suites: *Generic*, *specific*, *performance*, and *stability*. The implementation of principal methodologies in the *generic* test cases are discussed. The *performance*, *stability*, and *specific* tests are not included at this time and therefore scheduled for future work.

However, it is challenging to explore network variants by using black-box testing within an SDN paradigm because of the separation of control and data planes. Studies [9]–[11] show that only basic testing items are available. For instance, the reachability of packets from source to destination and ability to prevent loops in networks are analyzed and checked. Since such cases do not cover the two major issues mentioned, such limited coverage is not applicable as an application test for SDN networks.

As part of ongoing work on automation control test system (ACTS) [12] development, shown in Fig. 1, this paper presents selected test cases and validation criteria for the SDN application test. As illustrated in Fig. 2, the framework consists of sets of application-under-test (AUT) running on top of the Ryu controllers.

The main contributions of this paper are summarized below.

1) This paper presents test cases with systematic validation criteria of four suites: *generic*, *specific*, *performance*, and *stability.*
2) *Indirect packet triggering* and *scenario creation* are proposed as the principal methods for SDN applications testing.
3) Methodologies for *flow state checking*, *dual injection*, *tracking non-timeout flows*, and *burst detection* are proposed and supplemented with the principal methods in different test cases.
4) A novel three-layer validation procedure structure is presented. Applications must pass through suites of test cases layer by layer from generic cases to specific cases.
5) The system is designed based on an automatic test framework with multiple modules of different purposes to create the test environment, generate desired packet sequences, and analyze results for each test case automatically.

In this paper, we first give a brief review of the SDN architecture with an emphasis on northbound and southbound interfaces. Related works are summarized in Table I and covered in Section II. In Section III, we give our problem statement with notations. The proposed nontrivial test cases and methodologies are presented in details in Section IV. Implementation details and experimental results are discussed in Section V. Section VI concludes the paper with discussion and future research directions.

II. Background and Related Work

SDN architecture consists of several components. The centralized network control plane is programmable. The network functions provided by traditional network devices are abstracted as SDN applications, extending the capability of the SDN controller. The controller can also collect global network statistics from the network devices in order to analyze the packet and make decisions about the arriving events. As a result, networking devices at the layer of a data plane only have to take responsibility for packet forwarding and message processing according to the instruction assigned by the controller at the upper layer.

OpenFlow, providing southbound API in SDN architecture, is a significant communication interface that connects between the SDN controller and the OpenFlow-enabled devices. It defines the message exchange protocol between the control plane and data plane. It also defines the criteria for OpenFlow-enabled devices to process an incoming packet. Before an OpenFlow message can be delivered, a logic connection termed "OpenFlow channel" must be set up. Three categories of OpenFlow messages [1] are provided: one is the controller to manage or extract the state of the switch, the second is for a switch to call for help in order to solve unhandled events and to update network status for controllers, and the last is about connection set up communication.

In OpenFlow-enabled switches, packet processing is based on "flow entry." Each switch maintains one or more flow tables containing a list of flow entries. Each entry is composed of a set of match policy with desired instruction sets and other properties. The controller can thus manage and configure the switch by adding different flow entries to provide different network functions. Once a packet arrives in a switch, it is matched against the flow entry with the highest priority in the first flow table. At the same time, instructions of matched entry are directly applied to the packet. "Pipeline processing" is carried out for a switch with multiple flow tables, as a packet is directed to other flow tables

TABLE I
RELATED WORKS

Application	Type	Objective	Coverage	Test cases	Method
Veriflow [10]	Black box	Inspect network invariant	Functionality (forwarding)	Built-in, tester define	Forwarding graph
NetPlumber [11]	Black box	Inspect network invariant	Functionality (forwarding)	Built-in, tester define	Forwarding graph
Anomaly-free [13]	Black box	Flow conflict detect/recover	Functionality (flow conflict)	N/A	PSG model
Flow-based [14]	Black box	Flow conflict detect/recover	Functionality (flow conflict)	N/A	Forwarding graph
ADRS [15]	Black box	Flow conflict detect/recover	Functionality (flow conflict)	N/A	First-order logic
FortNOX [16]	Black box	Flow conflict detect/recover	Functionality (flow conflict)	N/A	Alias set rule reduction
Vericon [5]	White box	Application validation	N/A	Tester define	CSDN language
Verificare [6]	White box	Application validation	N/A	Tester define	VML model
High-Level Modeling [17]	White box	Application validation	N/A	Tester define	DSML
NICE [9][18][19]	White-Black box	Application validation	Functionality (forwarding)	Built-in, tester define	Model checking

for further processing. At the end of pipeline processing, all the actions which remain in action set of the packet are executed.

According to the executing environment, SDN applications can be classified into internal and external applications [20]. As its name implies, internal applications are executed inside the SDN controller and must be developed in the native programming language of a controller. Because of high interaction frequency in the controller, internal applications should have the characteristics of lower latency and complexity, otherwise, it may cause a deadlock in a controller as it is waiting for the response from the applications. In contrast, external applications running outside of the SDN controller communicate with the SDN controller according to application layer interface (or sometimes called northbound API). Therefore, the complexity of computation and the tolerance of processing latency can be more flexible than those of the internal applications. Furthermore, SDN applications can also be classified into reactive and proactive reactions, based on their behavior. The behaviors of responsive requests are frequently triggered by control packets. The proactive application performs its actions according to the initial configuration or external network events occurred.

Several works inspect network invariants, finding anomaly flows and validating SDN applications in SDN networks. Veriflow [10] and NetPlumber [11] check out network invariants during the operation of SDN. The method used in both of these maintain a forwarding graph based on flow entries populated by OpenFlow-enabled switches. The benefit of these two works [10], [11] is that real-time checking can be provided. But due to the methodology used, only forwarding-related behaviors can be validated. Also, both of these only consider limited parameters of matching fields in the flow entry, such as the IP address, the input port, and the output port. The coverage and quality of testing are not sufficient. Four other works focus on finding anomaly flow entries, detecting conflict flows, and attempt to resolve the peculiarities. Batista *et al.* [14] use first-order logic and ADRS [15] proposes a policy tree to inspect anomaly flows to resolve the problems. The coverage of match fields is, however, still limited. FortNOX [16], an extension of NOX controller is used to detect flow update contradictions in real time. The design is more focused on conflict detection of flows that use set operation to rewrite packet headers. With the same objective as previous works, Rezvani *et al.* [13] use a PSG model for detecting abnormal rules. It can support most of the matching fields and can provide much more accuracy in detecting results.

So far, the works covered above can verify only packet forwarding and rule anomalies during the operation of SDN networks. Other issues, such as SDN resource consumption and functionality, which may also directly impact the network operation are not taken into account in these studies. There are other four works presenting the same goal of SDN application validation as that in this paper. Vericon [5], Verificare [6], and high-level modeling [17] use white-box testing methodologies to verify SDN applications. Based on the novel identity-based signature scheme, PERM-GUARD [8] can be used for the controller to verify the validity of flow rules and reject fake flow rules created by applications. Although the validation can provide detailed inspection of application programs, the objectives are designed for developers to find bugs during the development stage of a program. The key is to obtain the source codes for each application under validation. Therefore, the notion of using white-box testing is not under our consideration. NICE [9] uses module checking and symbolic execution to verify invalid states of an SDN application program in different network states. It defines correct properties which are only related to network forwarding behaviors.

As summarized in Table I, most of the existent works concentrate more on developing a tool for developers to test their applications. Moreover, test cases covered in these works only analyze basic network forwarding requirements. Other network behaviors are not checked unless the developers design the

TABLE II
TABLE OF NOTATIONS

Category	Notation	Description
Entity	$C = \{c_i \mid i > 0\}$	Controllers
	$A = \{app_j \mid j \geq 0\}$	A set of Running app(s)
	AUT	The application under test. Category of Application Under Test (AUT), note as $AUT.cat$, where $cat \in CAT = \{cat_x \mid x > 0\}$
	$SW = \{sw_k \mid k > 0\}$	A set of switch(es), where $SW \subseteq TP$
	$H = \{h_m \mid m > 0\}$	A set of host(s), where $H \subseteq TP$
Process	$TP = \{tp_n \mid n > 1\}$	Topology for tc_q
	$PKTSQ = \{pktsq_{q,p} \mid p > 0\}$	A set of packet sequence for tc_q, where $pktsq_{q,p} = \{pkt_{L2,L3,L4}\}$ and the number of packet-in $pktsq_{q,p}$ denote as $N^{pktsq}_{q,p}$
	$TC = tc_q \mid q > 0$	A set of test case(s) for AUT
Evaluation	$T^{pktin-resp}$	The packet-in response time of AUT in specific tc_q
	$T^{discover}$, $T^{relearn}$	The topology discover time and relearning time with AUT installed in specific tc_q
	$R^{burst-pktin}$	Burst packet-in increase rate with AUT installed in specific tc_q
	R^{req} , R^{rep}	Average burst request/reply rate for AUT/SW with and without AUT installed in specific tc_q
	$R^{pktIn-freq}$	Packet-in frequency rate for AUT handling in specific tc_q
	$F = \{f_{k,y} \mid k > 0, y \geq 0\}$	The yth flow entry of switch k. The sum of flow entry usage S = all the flow entry $f_{k,y}$
	$M = \{m_k, y, z \mid y \geq 0, z \geq 0\}$	The yth flow's match field in kth switch
	$FN = \{fn_q \mid q > 0\}$	A set of false negative of AUT of the qth test case
	$FP = \{fp_q \mid q > 0\}$	A set of false positive of AUT of the qth test case
	$N^{packet-in}$	The number of packet-in messages
	N^{pre}, N^{L2},N^{L3}, N^{L4}	The number of pre-push flows and the number of L2, L3, and L4 flows

query procedures and network policies themselves for specific verification.

III. PROBLEM STATEMENT

Currently, there are no well defined criteria for SDN application validation. As a result, we propose systematic criteria with broader coverage for SDN application validation. In this paper, 17 test cases are proposed to address the validation of SDN applications. These test cases are developed to test functionality, resources, performance and stability issues in SDN networks, both in physical and in virtual SDN environments. The first two categories are required for the validation procedure and are designed to evaluate whether an application achieves the basic requirements. The proposed schemes gain deeper insight into blackhole detection and flow conflict detection issues found in forwarding and resource consumption, respectively, and can provide more reliable and accurate verification. The three categories of notations that are used in this paper are listed in Table II. Given an application under test AUT, a set of applications which are running on the SDN controller are denoted as A and C, respectively. The set of test cases proposed is denoted as TC. Each case can be applied to a different topology TP and packet sequence $PKTSQ$ designed by testers required. The evaluation parameter can be divided into three parts—time T represents the timing-related performance of test cases; rate R states the difference impacted by AUT, and flows F assesses the flow table resource issues. The objective of this paper is to pursue the minimal false positive FP and false negative FN rates according to the real status of AUT among all the TC.

IV. METHODOLOGY

The framework of SDN application validation proposed in this paper is based on a set of test cases TC. Each of them represents different aspects of issues found in the SDN applications that need to be tested for each of AUT including packet forwarding, resource consumption, performance, and operation stability.

Before starting the validation, Each AUT must state its category and supporting protocol according to the developer's design concept. To construct a suitable execution environment for AUT, the corresponding controller C and applications A running on it also need to be provided. In order to narrow the scale of the problem statement of this work, three constraints are set. The testing procedure needs to be conducted automatically. We consider the AUT as one application instead of multiple ones even though there may actually be some applications running on the controller during testing. The validation time needs to be set under a threshold to avoid infinite waiting. As the example shown in Fig. 1, the OpenFlow channel is monitored, and the OpenFlow communication messages between the controller (or says AUT) and the switches are analyzed. The evaluation result regarding the time ($T^{\text{pktin-resp}}$) and rate (R^{req}, R^{rep}) parameters mentioned in Table II can then be presented. Moreover, the proposed methods also focus on the flow entries populated by the AUT inside each switch, in order to inspect the match fields M of each entry in the switch to identify flow conflict problems and to measure the flow table usage S. Finally, the validation report generated contains the detail result of each test case.

The design of proposed test cases, as shown in Fig. 2, can be broken down into three layers. The first layer is that of *generic test cases* in the data plane. One basic requirement is that packets can be forwarded to their destinations. Another concerns the southbound API in SDN network. Compatibility of OpenFlow specifications between the control and data planes also need to be considered. Furthermore, detecting usage problems in a data plane flow table is also an important issue which needs to be tested. This includes flow conflict problems and flow recycling problems. The second layer is that of *specific test cases* in the control plane, which focuses on validating the correctness of the network functions that AUT claims. As it is impossible to validate the functionality of each application, we divide

TABLE III
LIST OF PROPOSED KEY METHODOLOGIES IN THE SUITE OF GENERIC TEST CASES

Methodology	Test Case
Generic Method	Indirect packet triggering, Scenario creation (Applied in all test cases)
Foreground-with-Background	Flow table usage
Flow-State-Checking	Blackhole detection
Dual-Injection	Flow conflict detection
Tracking-Non-Timeout-Flows	Flow self-recycling
Burst Detection	Burst Detection (Packet-in, Request Messages)

possible functionalities into five categories of load balance, traffic steering, firewall with intrusion detection, node with link discovery, and virtual tenant network. The test cases are then designed according to these five categories instead of the application itself. The last one is composed of the *stability* and *performance* related test cases in the control plane. These test cases address most of the resource consumption issues. The purpose is to test whether any potential overheads occur in the controller. Each AUT must be validated through layers starting from the generic test cases, and then specific functionality test cases, followed by performance and stability cases for advanced testing. The overview of proposed key methodologies in the suite of generic test cases is listed in Table III, and the approaches are discussed in detail in the following sections.

A. Generic Method

It is difficult to extract the behaviors of the SDN application directly. For the application validation purpose, two major approaches of *indirect packet triggering* and *scenario creation* are applied to all the test cases in order to observe the behaviors of the AUT.

The *indirect packet triggering* method is proposed because the applications on controllers do not process packet sequences designed by testers directly. Instead, a packet needs to be sent via switches in the data plane and then indirectly passed within OpenFlow channel to the controller for further processing. Also, some scenarios need to be created to fit into different situations according to the characteristics of the applications. For example, some applications can be activated only under certain circumstances while the behavior of some functions is hard to predict. Thus, we need to create several scenarios to increase the effectiveness of the test. The proposed method can drive the AUT to present its processing reasoning and verify if the test criteria can be passed. During the test, test engineers send various packet sequences, depending on the protocols supported by the AUTs, to indirectly trigger the AUT running on the controller. The test case can then collect the corresponding messages on OpenFlow channel for further analysis. However, some behaviors of AUT are shown only when specific scenarios happen which is difficult to predict. In order to increase the effectiveness of the test, the creation of multiple network scenarios are needed to fit the characteristics of the AUT. The scenarios created in this paper can be briefly classified as connection setup, packet traversed, and topology changed. They are applied in the test cases of OpenFlow error detection and burst request detection.

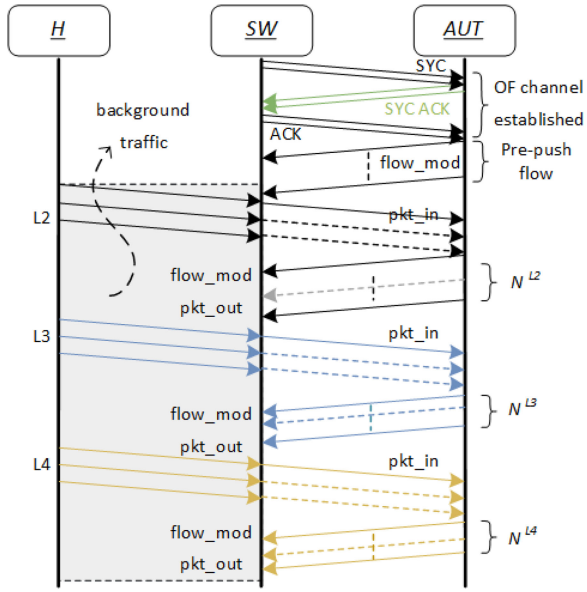


Fig. 3. Messages exchanged in the test of foreground-with-background process.

B. Foreground-With-Background

The methodology of foreground-with-background is applied to the test case of flow table usage. The purpose of this test case is to measure total flow entries used during the operation of AUT. It does not make sense to compare different results by just calculating the sum of flow entries populated in switches. As a result, the proposed method presents a tendency graph showing the increasing occupancy of flow entries while AUT is executed.

We assume there are multiple hosts in the testing network topology. For each test iteration, a set of hosts are chosen to be the target hosts, and the remainder are the backups. To simulate the real situation of network operation, two types of traffics were deigned. The foreground traffics, also named as target traffics, were composed of packets in different network layers that AUT supports. For example, Ethernet, IP, and TCP packets belong to link layer (L2), network layer (L3), and transport layer (L4) protocols. The background traffics used to communicate between backup hosts to occupy the link bandwidth for simulating real network status.

Fig. 3 shows the example of the test iteration. The background traffics (shadow part in Fig. 3) were generated to occupy link bandwidth with desired percentage. Foreground traffics were sent to switch one type after another from lower to upper network layer to trigger the AUT indirectly. Therefore, AUT may respond to those target packets by using flow-mod and packet-out messages. In this case, only flows $f_{k,y}$ populated by the flow-mod messages that AUT sent out are concerned. The goal is to calculate the sum of prepush flows N^{pre} and average number of flow entries, denoted as N^{L2}, N^{L3}, and N^{L4}, in different networks layer for all switches. The sum of average flow entry

usage can be derived as

$$S = N^{\text{pre}} + N^{L2} + N^{L3} + N^{L4}. \quad (1)$$

Test engineers can change the topology with switch and host to construct various scenarios to validate the AUT. Finally, the tendency graph can be constructed with different topologies in the x-axis and average number of flow entry S per switch in the y-axis.

C. Flow-State-Checking

Flow-state-checking methodology was used to validate the issue of blackhole for an AUT during its operation. The blackhole is defined as packets which are dropped silently and do not reach their destination. According to the specifications of OpenFlow, packet processing in OpenFlow-enabled switches is based on pipeline processing. Activities of an incoming packet are determined by the action set it contains. The action set can be modified based on the results of its matching fields. Two possible ways may lead to packets being dropped in a switch, one as a result of an empty action set, and the second one as a result of action sets with nonexistent output ports. Based on our observations, things need to be checked in the flow entries are those instructions that produce "empty action set" and "invalid output port" responses once a flow is detected and identified as "potential blackhole". The testing procedure continues to check whether there are any flows with the same matching fields but have higher priority specified. If yes, the flow is no longer reported as "potential blackhole" because no packets match this specific flow.

Furthermore, most of the flows may have their idle and hard timeout values specified. Flow entries are removed as timeout values expired. As we noted above, the test case checks each flow entry in the switch. The accuracy of the proposed method is affected because the checks to those flow entries were missed. A way to fix the issue is proposed in the test case. The frequency of checking is set to be shorter than the lowest timeout value in all flow entries to ensure the total coverage of checking in the switch.

The flow chart of the proposed checking methodology is shown in Fig. 4. Initially, flow entries related to the AUT are sampled periodically. Then flow entries are checked for potential blackholes based on the rules illustrated in the flow chart. The flowchart is designed according to the pipeline defined in the OpenFlow specification. The checking procedure presented in the flowchart is also suited for multiple flow tables in the OpenFlow switch. Furthermore, the proposed method can also detect network blackholes on meter table and group table.

D. Dual-Injection

Based on the specifications of OpenFlow, applications running in the controller can manage and configure switches to provide network functionality by populating different flow entries. Therefore, there is a possibility for a potential issue of flow conflict, resulting from the creation of multiple flow entries which consist of overlapping match fields with the same priority but distinct instructions. Once a packet arrives in a switch, all

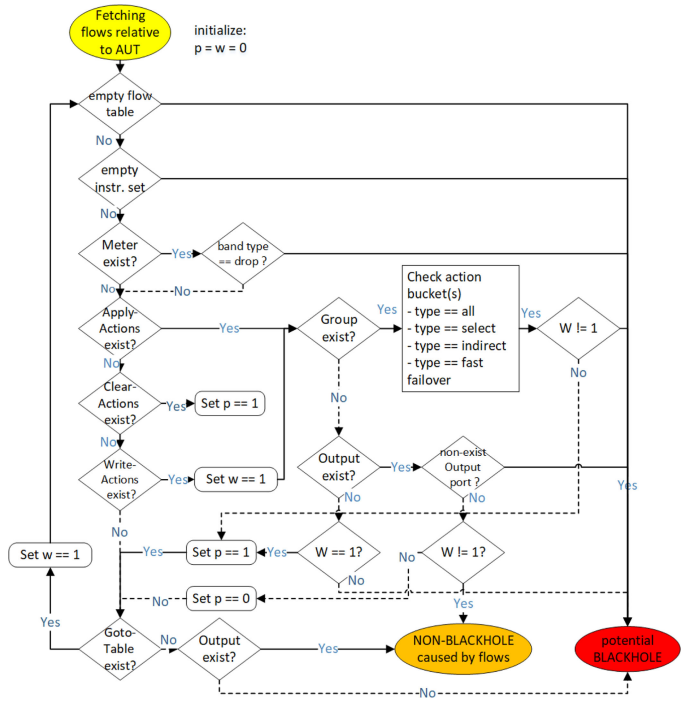


Fig. 4. Flowchart of the flow-state-checking for blackhole detection. The "Blackhole" is referring to an outcome where packets are dropped silently and cannot reach to its destination. In the beginning, flow entries related to the AUT are sampled periodically. Then, flow entries are checked based on the rules illustrated in the flowchart. As documented in the specification, packets with "Empty flow table", "empty instruction set," and "Meter table with drop band type" must be dropped. For those flow entries with "Apply-Action" and "Write-Action" instructions, further check is needed. The purpose is to verify the existence of "Group table" and "nonexist output port" which may cause potential packet drop according to the OpenFlow switch specification.

the conflict flow entries are matched. Only one packet can be assigned once with an action with the matched entries. Therefore, the issue of flow conflict leads to ambiguous behaviors in the network operation.

As shown in Fig. 5, a dual injection method is proposed to validate flow conflict problem. In the first phase, packet injection is used to trigger the AUT for populating flows indirectly. In the second phase, overlapping parts among those flow entries are checked. After that, the test case generates packets that are associated with the rules of overlap flows and observes the status of flow counters. For example, assume that two applications are working together and generating flow entries in a switch with overlapping match fields. In the second phase shown in Fig. 5, the value in one of the flow counter is increased, and the other stays the same.

The proposed method can check the parameters of matching fields in network layers two, three, and four. We evaluate the priority value in each of the flow entries because the practical higher priority operators are capable of covering the flows of lower priority. The related works [13]–[16] are able to fully detect overlap in the matching fields of port and IPv4 addresses. The major difference, compared to the other proposed methods, is that we can validate the flows of layer two and layer three with the support of IPv6. For example, one of the flows is matched

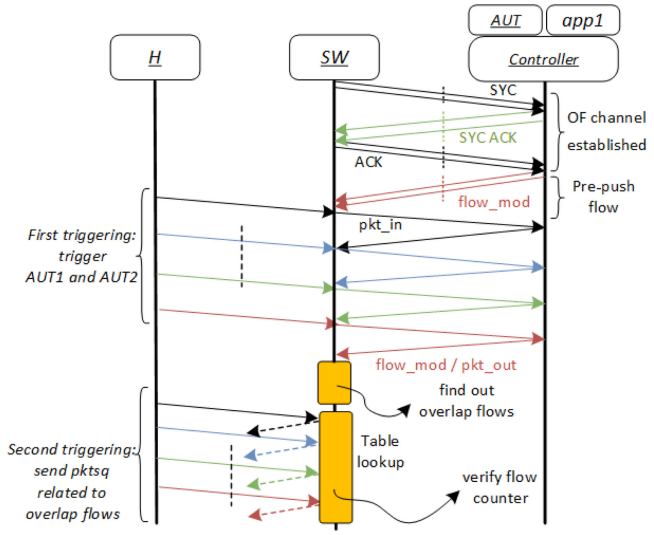


Fig. 5. Transaction messages in the method of dual-injection. In the first phase, packet injection is used to trigger the AUT for populating flows indirectly. In the second phase, overlapping parts among those flow entries are checked by verifying the flow counters.

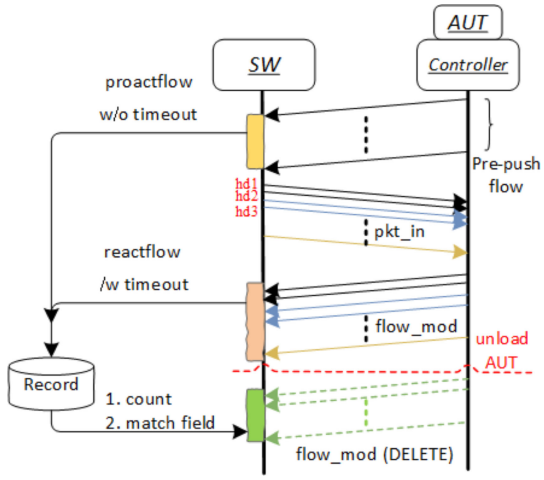


Fig. 6. Proposed processes for tracking the non-timeout-flows. The test checks whether there are flow-mod messages with deleted type sent out from a controller (or AUT) to "recycle" all the non-timeout flow entries.

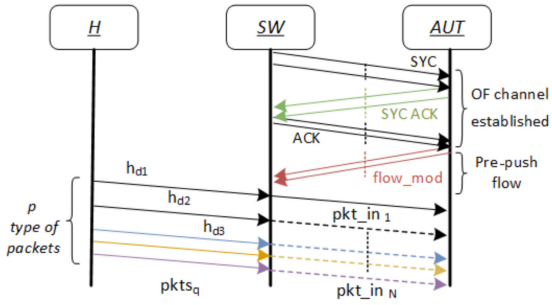


Fig. 7. Process of burst-packet-in-detection. The checking criteria only allow one packet-in message to be sent to the controller with the same type of packet.

with a MAC address, and the other is matched with an IP address. When packets become matched with both MAC and IP addresses and are directed to the same host, such a conflict scenario can be identified by our proposed method.

E. Tracking-Non-Timeout-Flows

In the criteria of mobile application validation, there is a standard test item called "garbage collection". Garbage collection is used to recycle unused resources to protect limited system resources. The same idea can be applied in an SDN environment. Flow entries in an OpenFlow-enabled switch represent the abstraction of the desired network functions provided by the applications. Once the application is unloaded from an SDN controller, the flow entries set by the applications are also required to be removed. Thus, the purpose of this test case is to verify whether an AUT is capable of removing the flow entries after it has been unloaded.

Typically, flow entries with timeout values are automatically removed from the switch when expired. As shown in Fig. 6, the proposed method first tracks the flows that have no timeout value specified. The test case forces the AUT to be unloaded from an SDN controller. At the same time, it checks whether there are flow-mod messages with deleted-type sent out from a controller (or AUT) to "recycle" all the non-timeout flow entries. The APP itself has the knowledge of rules being installed. Moreover, the controller can keep the status of rules being installed by which APPs. Thus, once the APP is stopped, the controller has the capability to clear the rules. During the test, the framework has the full observability of the transactions in the OpenFlow channel. Therefore, the flow-remove messages sent can be checked and distinguished based on different running applications. This checking process is composed of two parts. First, the value of total count needs to be set the same as those flows we want to track; second, we need to compare the matching fields in the recorded database to guarantee that all flows in the switch have actually been removed.

F. Burst Detection

1) Burst-Packet-in-Detection: In a test case of burst detection, the purpose is to inspect the potential problem of generating burst packet-in messages from switches in SDN applications. The checking criteria only allow one packet-in message to be sent to the controller with the same type of packet. This is because after the first packet-in message request from the switch, the application typically needs to send out a corresponding flow-mod message to avoid the same type of packet-in message again for better efficiency.

As illustrated in Fig. 7, the test case sends p different types of packets each with a fixed number N_p to trigger the AUT indirectly. We then inspect how many packet-in messages ($N^{\text{packet-in}}$) generated by k switches are a result of the prepush flow populated by the AUT. We also re-iterate the test with and without the AUT installed to compare the difference ratios between them. According to the criteria we noted above, the formula is derived as

$$R^{\text{burst-pktin}} = \frac{\frac{N^{\text{packet-in}}}{k}}{p \times N_p}, \quad \text{where} \quad R^{\text{burst-pktin}} \geq 0. \tag{2}$$

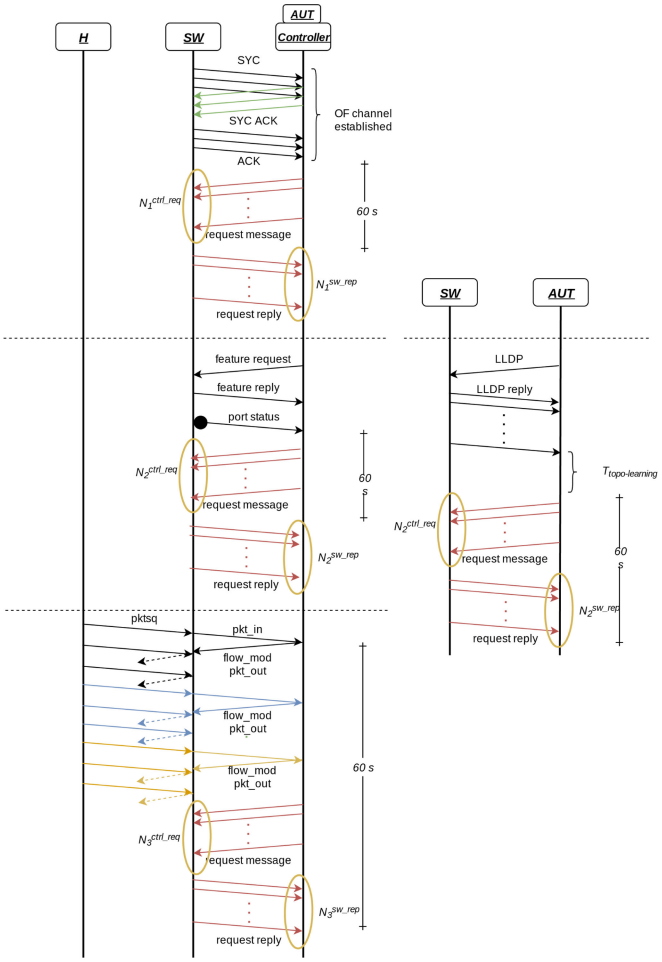


Fig. 8. Procedures of the burst-request-detection process. Three different scenarios are constructed to inspect the burst request messages from the controller.

When the value of $R^{\text{burst-pktin}}$ is greater than one, it means there is more than one packet-in message generated by the same type of packets, and the test fails.

2) Burst-Request-Detection: The purpose of this test case is to inspect the ratio of requested messages sent out from a controller and reply messages from switches in the case of the AUT installation and un-installation. It is known that some OpenFlow messages, such as multipart request and reply messages, may produce an unequal number of request and reply messages. Sometimes the size of a reply message is larger than that of the request messages. This behavior potentially increases the controller processing overhead due to larger messages being sent.

However, the behavior of requested messages from a controller is hard to predict. Therefore, three different scenarios were constructed to inspect the burst request messages from the controller, especially for those messages causing high switch loading and controller loading. Such scenarios are connection setup, topology change, and traffic traverse. As an illustration of these three scenarios in Fig. 8, we collected the number of *request* and *reply* messages from the controller and switches denoted as $N_i^{\text{ctrl-req-aut}}$, $N_i^{\text{sw-rep-aut}}$, $N_i^{\text{ctrl-req-nonaut}}$, and $N_i^{\text{sw-rep-nonaut}}$. The controller request rate and switch reply rate can be derived as

$$R^{\text{req}} = \frac{\sum_{i=0}^{n} N_i^{\text{ctrl-req-aut}}}{\sum_{i=0}^{n} N_i^{\text{ctrl-req-nonaut}}} \tag{3}$$

and

$$R^{\text{rep}} = \frac{\sum_{i=0}^{n} N_i^{\text{sw-rep-aut}}}{\sum_{i=0}^{n} N_i^{\text{sw-rep-nonaut}}} \tag{4}$$

respectively.

TABLE IV
SEVERAL PARAMETERS ARE CHANGED DYNAMICALLY IN ORDER TO CONDUCT SIMULATIONS OF DIFFERENT SCENARIO IN THE TEST EXPERIMENTS

Test Cases	Parameter Changed in Each Test Iteration
Blackhole detection (flow-state checking)	Topology: Linear, Ring, Tree, Fully Mesh (with single table and multi-table)
Flow table usage (foreground-with-background)	Traffic type: L2/L3/L4 packets + Different number of switches and hosts
Flow conflict detection (dual-injection)	Traffic type: L2/L3/L4 packets + Different Topology
Junk Flow Recycling (checking-non-timeout-flows)	Traffic type: L2/L3/L4 packets

The traffic type of L2, L3, and L3 represents the link layer (L2), network layer (L3), and transport layer (L4) packets.

V. SYSTEM EVALUATION

We built an automatic test framework for the validation of SDN applications. The test cases proposed are implemented on top of a Ryu controller. The proposed architecture is shown in Fig. 1. For each test case, the test steps consist of modules of different functionalities written in Python scripts. Those modules can be grouped into five different categories of controller control, OpenFlow channel sniffer, OpenFlow packet analyzer, flow table sniffer, and virtual topology. Those modules can manipulate testing component according to various test requirements, such as testbed configuration, packet generation, and result analysis.

A. Experiment Setup

The available configurations for each test case consist of virtual switch type, packet sequence, and network topology. We typically utilize the Open vSwitch (v 2.5.0) in our experiment. The OFSoftswitch can also be used if needed. Scapy (v 2.3.2) is used as the packet generator in the framework. The available variables for generating packet sequences are composed of packet size, packet count, and different network protocols from layers two to four. By default, basic linear, ring, tree, and fully mesh topology for each test can be further extended to any custom topology based on Mininet (v 2.2.1) for all the test cases. The major setting parameters of the experiment that changed over time are shown in Table IV.

Table V, lists the the proposed test cases and methods that we used to validate the AUTs. The applications were collected from the official Ryu (v 4.8) controller modules [21] and the Github. Most of the collected applications do not use advanced OpenFlow functionality, such as multiple flow table, group table,

TABLE V
COLLECTED OPEN-SOURCE SDN APPLICATIONS UNDER TEST

Category	Application	Description
Controller Built-in Applications	Simple Switch 13	Ryu built-in module [21]
	Simple SwitchSTP 13	
	Simple Monitor 13	
Open Source Applications	RoutingFlow [22]	OF1.0 Applications (Github)
	Network Awareness	
	Shortest Forwarding [23]	
	Source NAT [24]	
	Ryu-DHCP [25]	
Simple Python Scripts	Multipath Routing	Group, Meter Table
	Access Control	Multiple flow table
	Broadcasting	Group table
	DNS Director	

The controller application of dynamic host configuration protocol (DHCP) is adopted from the Ryu framework.

and meter table. In order to check the proper operations of the proposed method in all the OpenFlow-enabled environments, some specific scenarios were created for an auxiliary test.

B. Test Results and Discussions

This section presents the experimental results obtained by applying the proposed test cases with its corresponding methodology to the SDN applications listed in Table V. The summary of all the experiment results of each AUT is listed in Table VI. In each subsection below, the representative numerical results are presented in detail for specific test cases.

1) Flow Table Usage: The purpose of this test case was to determine the space consumption of the flow table for a specific SDN application. The test was conducted based on the given foreground and background traffic. The foreground traffic consisted of different network layer protocols supported by the AUT. For each test iteration, background traffic was generated with protocols from layers two to four and sent to selected target hosts.

Three types of results were observed in the experiments. The most common case was that the number of flow entries (e.g., simple_switch_13) was proportional to the number of hosts. For the same destination, there were no flows populated when L4 packets were sent because of the previous L2 and L3 packets had already triggered the AUT to generate corresponding flow entries. This scenario was commonly seen in the forwarding applications. No matter what protocol the incoming packet is and what topology exists in the data plane, these AUTs have the similar behavior to process packets based on packet's destination. Another test was observed in the application of network address translation (NAT) [24] where only TCP and UDP packets were generated to test the "source NAT" application because of its implementation with limited capability. A flow entry was populated when a packet was sent from a host for flow connection. The number of flow entries was proportional to the number of hosts and different types of protocol. Finally, there were some applications (e.g., simple_monitor_13) that used only a fixed number of flow entries. It is easy to conclude that this is the best scenario where few flow entries consume just a small amount of memory space. In fact, from our observation, this type of application may direct all incoming packets to the controller causing extra processing overheads.

2) Blackhole Detection: For the test case of blackhole detection, a flow state flowchart based on the OpenFlow specification was constructed. During the test operation, the metadata of flow entries were fetched from the switch with the time period determined by the minimal timeout value of all entries. The results of this experiment are shown in Table VII. According to the experiment results, no blackholes were found in most of the applications except those with spanning tree. The reason is that when spanning tree encounters loop topology, specific ports may be blocked to prevent broadcast storm. For a broadcasting scenario, it is considered as blackhole because an invalid output port exists in one of the action buckets in the group table. The test of access control also failed because of their functionality requirements. Those AUTs without populating flow entries are marked as non-tested (N/T) in Table VI. Instead of using original flow entry to check for reachability validation as found in other works [10], [11], the proposed method can be used to detect the occurrence of blackholes accurately in both group and meter table (broadcasting and multipath routing scenario) defined in OpenFlow specification.

3) Flow Conflict Detection: The test case of flow conflict detection was used to inspect flow entries for ambiguous behaviors of matched packets. Different from those real-time checking proposed by other works, the main objective was to detect conflict flows before the application was deployed. Therefore, as shown in Table VIII, some scenarios were created to verify the effectiveness of the proposed method. The conflict flow entries in the first two scenarios were found because there were no mutually exclusive parameters. That is, the input-port, IP address, and ARP address appeared simultaneously. The last one is a particular case. Assuming there were two flow entries, both of them were set with different actions without proper priority. As a packet arrives at the switch with the MAC address of 00:00:00:00:00:01 and IP address of 10.0.0.1, it is possible to match these two flow entries resulting in potential conflict. The proposed method can detect all the conflicts in these three scenarios while the other can only detect partial scenarios.

4) Flow Self-Recycling: All of the Ryu built-in modules failed the flow self-recycling test. The main reason is that the timeout values were not specified in the flow entries generated by the Ryu module. Furthermore, according to our observation, flow entries in all of Ryu modules could not be recycled and deleted after the application was unloaded, because of a "stop" event handler was not provided in the application program. Once the application was unloaded, it could not achieve the recycling requirement. Furthermore, the feature of dynamic load and unload of applications was not supported in the Ryu controller. For the other applications with timeout value specified in each of its flows, the tests passed in our verification. It is obvious that the flow is automatically invalid when timeout.

5) Burst Packet-in Detection: For the test case of burst packet-in detection, the packet-in rate for each AUT was measured. In the test, packets of different flows were sent to the switch. A constant number larger than one (e.g., 100) was

TABLE VI
SUMMARY OF THE TEST EXPERIMENT RESULTS

Application Under Test	Blackholes	Flow Conflict	Burst Packet-in	Burst Request	Flow Table Usage	Flow Self-recycling	Remark
Simple Switch 13	none	none	≤ 1	$R^{req} = 1, R^{rep} = 1$	proportional to host	fail (no timeout)	
Simple Switch stp	1 potential	none	1, except stp	$R^{req} = 1, R^{rep} = 1$	proportional to host	fail (no timeout)	
Simple Monitor 13	N/T	N/T	N/T	$R^{req} = 10$, $R^{rep} = 10$	constant (default flow)	fail (no timeout)	multipart request
DNS Director	N/T	N/T	> 1	N/T	constant (default flow)	fail (no timeout)	only DNS
Shortest Forwarding	none	none	≤ 1, except lldp	$R^{req} = 10$, $R^{rep} = 10$	proportional to host	pass (timeout)	
Network Awareness	N/T	N/T	≤ 1, except lldp	$R^{req} = 1, R^{rep} = 1$	N/T	N/T	Func. error
RoutingFlow	none	none	>1,except icmp	$R^{req} = 1, R^{rep} = 1$	proportional to host	pass (timeout)	OF1.0
Source NAT	none	none	≤ 1	$R^{req} = 1, R^{rep} = 1$	proportional to flow #	fail (no timeout)	TCP, UDP
Ryu-DHCP	N/T	N/T	> 1	N/T	N/T	fail (no timeout)	UDP only
Multipath Routing	none	2	N/T	N/T	N/T	N/T	a scenario
Broadcasting	1 potential	N/T	N/T	N/T	N/T	N/T	a scenario
Access Control	1 potential	1	N/T	N/T	N/T	N/T	a scenario

The AUTs without populating flow entries are marked as non-tested (N/T).

TABLE VII
TEST RESULTS OF AUTS IN BLACKHOLE DETECTION

AUT	Simple Switch 13	Broadcasting	Access Control	Multipath Routing
Exist Black-hole	in single table	in group table	in multiple tables	meter table but no black-holes
Other Meth-ods	Blackhole Detected	can not detect group table	Blackhole Detected	cannot detect meter table
Proposed	s2: {table=0, priority= 0, inport:2, actions=drop}	s1: {group_id=11, type=all, bucket=output:1, bucket=output:2, bucket=output:10}	s1: {table=0, priority=0, actions= drop}	no blackholes
Remark	flow contain drop action (due to stp)	non-existent output port in group bucket action	table miss flow contain drop action	

TABLE VIII
FAILURE SCENARIOS FOR FLOW CONFLICT DETECTION

AUT	Multipath routing (ipv4)	Multipath routing (ipv6)	Cross layer
Exist conflicts	2 conflict	1 conflict	1 conflict
Other methods	2 conflicts detected	not support IPv6	not supported cross layer checking
Rule Entry	{in_port: 4} v.s. {dl_type: 2048, nw_dst: 10.0.0.1} {in_port: 4} v.s. {dl_type: 2054, arp_tpa: 10.0.0.1}	{in_port: 4} v.s. {dl_type: 32545, ipv6_dst: "::ffff:10.0.0.1"}	{dl_type: 2048, nw_dst: 10.0.0.1} v.s {dl_dst: 00:00:00:00:00:01}

chosen to be the packet count threshold for each flow. Thus, the packet-in threshold could be represented as 1/100. This meant that for each flow, the AUT could only generate one packet-in message among 100 packets. If more than one packet-in message was generated, the AUT could potentially cause processing overheads on the controller. According to the results, the *DNS director* and *Ryu-DHCP* applications utilized a fixed number of flow entries to direct all packets related to protocols of Domain Name System (DNS) and Dynamic Host Configuration Protocol (DHCP) to the controller. For official Ryu modules, such as "simple switch 13", "simple switch stp 13", and other open source applications, only the first packets of the same type was directed to the controller. For the RoutingFlow application, only Internet control message protocol (ICMP) flow entries were populated in the switch, no matter what types of packets were generated in the test. The three applications covered above may create burst packet-in messages causing extra processing overheads in the controller when a large number of flows arrived.

6) Burst Request Detection: In the burst request detection test case, the numbers of the request and reply messages were counted based on the method mentioned in Section IV while the AUT was running. As the results shown in Table VI, the ratio of the request and reply count was equal to one for most of the applications. That is, no further request messages were shown for querying data plane information. However, two applications (e.g., Simple Monitor 13, Shortest Forwarding) with the ratio of ten were observed. These two applications have the functionality to collect network status. Constant polling of ten seconds was utilized to grab data plane information periodically.

VI. CONCLUSION

In this paper, we have proposed systematic validation criteria for the evaluation of SDN applications. As illustrated in Fig. 2, the purpose of this novel three-layer systematic validation structure is to make sure that all SDN applications under test are functioned properly before being deployed. The proposed measures consist of four significant suites—*generic*, *specific*, *performance*, and *stability*. SDN applications must pass through suites of test cases layer by layer from top to bottom. Based on these criteria, 17 test cases and methodologies are proposed to the ONF [2], with the pending review for standardization process. In particular, key methodologies in the suite of generic test

cases (e.g., *flow table usage*, *flow conflict detection*, *blackhole detection*, *flow recycling,* and *burst detection*) are discussed in detail with experiment results presented. Currently, among all the test cases proposed, six of them have been implemented and integrated into the ACTS [12].

A. Lessons Learned

To sum up, we have observed that the request messages are mostly used in network-awareness-related applications. Those applications have the capability of monitoring the status of the data plane based on the scheme of fixed-time polling. In the evaluation results of flow table usage, the AUTs which always populate default and fixed flow entries consume the minimum amount of space. In flow conflict detection, the technique proposed covers most of the parameters defined in OpenFlow specifications. It also supports cross-layer checking between two flow entries. With the priority value being considered, the proposed method provides better accuracy and coverage on the application validation. The proposed method utilizes the instruction set of flow entry instead of forty parameters in match field to check for blackholes. It can also support checking in both group and meter tables. In the validation results, no blackholes are found except for the application running the spanning tree protocol and unique access control.

Flow Self-Recycling: In order to forward all of the packets from a switch to the controller, fixed flow entries are kept in the switch without timeout values specified. An inability to recycle the flow entries was observed when an application was unloaded.

Timeout Values: According to our evaluation, the flow entries populated by official built-in modules in Ryu controller had no timeout values specified. As a result, it is recommended that each flow entry needs to be specified with a timeout value to prevent flows from occupying flow table when the corresponding application is unloaded.

Performance Issue: High CPU loading in the controllers was observed for AUTs populating default and fixed flow entries, because each incoming packet was directed to the controller waiting for further processing.

B. Future Work

Currently, the coverage of test suites other than the generic one is not provided due to the limited time available. In the near future, we plan to classify SDN applications (e.g., load balance, link discovery, and intrusion detection) based on their possible functionality to enhance the coverage in specific functionality validation shown in Fig. 2.

There are roughly two types of NIDS, i.e., the signature-based and behavior-based intrusion detection systems. We plan to focus on the APPs related to the behavior-based NIDS. As most of them constantly push and pull command and statistical metadata between switches and controllers. The communication and processing overheads not only consume the extra bandwidth but also increase CPU loading in switches and controllers. Therefore, the performance characterization of the application under tests (AUTs) in regard to the processing latency, bandwidth consumption, and accuracy will be the main design goal in the future works. The response time and frequency capacity for the *packet_in* messages are now being implemented based on the NIC's hardware timestamp for better measurement precision in the performance and stability test suites. The final goal is to enrich application validation procedures and make them more reliable for all test cases.

REFERENCES

[1] "Openflow switch specification," Accessed: Mar. 14, 2018. [Online]. Available: https://www.opennetworking.org/software-defined-standards/

[2] "Open network foundation," Accessed: Jun. 4, 2017. [Online]. Available: https://www.opennetworking.org/

[3] "OpenFlow controller benchmarking methodologies V1.0 ONF TR-539," Accessed: Feb. 12, 2017. [Online]. Available: https://www.opennetworking.org/

[4] "Conformance test specification for OpenFlow switch specification v1.3.4, basic single table conformance test profile," Accessed: Jun. 4, 2016. [Online]. Available: https://www.opennetworking.org/

[5] T. Ball *et al.*, "VeriCon: Towards verifying controller programs in software-defined networks," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 282–293.

[6] R. Skowyra, A. Lapets, A. Bestavros, and A. Kfoury, "A verification platform for SDN-Enabled applications," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2014, pp. 337–342.

[7] A. El-Hassany, J. Miserez, P. Bielik, L. Vanbever, and M. Vechev, "SDNRacer: Concurrency analysis for software-defined networks," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Des. Implementation.*, 2016, pp. 402–415.

[8] M. Wang, J. Liu, J. Chen, X. Liu, and J. Mao, "PERM-GUARD: Authenticating the validity of flow rules in software defined networking," *J. Signal Process. Syst.*, vol. 86, no. 2/3, pp. 157–173, Mar. 2017.

[9] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test openflow applications," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, San Jose, CA, USA, 2012, pp. 127–140.

[10] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 467–472, Sep. 2012. [Online]. Available: http://doi.acm.org/10.1145/2377677.2377766

[11] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, Lombard, IL, USA, 2013, pp. 99–112.

[12] "Automation Control Test System," Network Benchmarking Lab. Accessed: Jun. 30, 2018. [Online]. Available: https://www.oprueba.com/products#prod-acts

[13] M. Rezvani, A. Ignjatovic, M. Pagnucco, and S. Jha, "Anomaly-free policy composition in software-defined networks," in *Proc. IFIP Netw. Conf. (IFIP Netw.) Workshops*, 2016, pp. 28–36.

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

[15] P. Wang, L. Huang, H. Xu, B. Leng, and H. Guo, "Rule anomalies detecting and resolving for software defined networks," in *Proc. IEEE Global Commun. Conf.*, 2015, pp. 1–6.

[16] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for openflow networks," in *Proc. First Workshop Hot Topics Softw. Defined Netw.*, New York, NY, USA, 2012, pp. 121–126.

[17] F. A. Lopes, L. Lima, M. Santos, R. Fidalgo, and S. Fernandes, "High-level modeling and application validation for SDN," in *Proc. IEEE Netw. Oper. Manag. Symp.*, 2016, pp. 197–205.

[18] P. Perešíni and M. Canini, "Is your openflow application correct?" in *Proc. ACM CoNEXT Student Workshop*. New York, NY, USA, 2011, pp. 18:1–18:2.

[19] M. Canini, D. Kostic, J. Rexford, and D. Venzano, "Automating the testing of openflow applications," in *Proc. 1st Int. Workshop Rigorous Protocol Eng.*, 2011.

[20] "Brocade SDN controller developer wiki," Accessed: Jan. 14, 2017. [Online]. Available: https://github.com/BRCDcomm/BVC/wiki/

[21] "Ryu controller," Accessed: May 12, 2017. [Online]. Available: https://osrg.github.io/ryu/

[22] Y.-H. Hung, "Routingflow," Accessed: Jun. 5, 2016. [Online]. Available: https://github.com/hungys/RoutingFlow

[23] "EPCC-Networking Shortest Forwarding," [Online]. Available: https://github.com/epcc-networking/ryu-event. Accessed on: Jun. 6, 2016.
[24] C.-W. Lin, "Source NAT," Accessed: Nov. 3, 2016. [Online]. Available: https://github.com/John-Lin/nat
[25] A. Hill, "Ryu-DHCP," Accessed: Oct. 4, 2016. [Online]. Available: https://github.com/andyhky/ryu-dhcp

Ying-Dar Lin (F'93) received the Ph.D. in computer science from the University of California at Los Angeles (UCLA), Los Angeles, CA, USA, in 1993.

He is a Distinguished Professor of computer science at National Chiao Tung University (NCTU), Taiwan. He was a Visiting Scholar at Cisco Systems in San Jose, CA, USA, during 2007–2008, CEO at Telecom Technology Center, Taiwan, during 2010–2011, and the Vice President of National Applied Research Labs (NARLabs), Taiwan, during 2017–2018. Since 2002, he has been the Founder and Director of Network Benchmarking Lab, Taiwan, which reviews network products with real traffic and automated tools, and has been an approved test lab of the Open Networking Foundation (ONF) since July 2014. He also cofounded L7 Networks Inc., Taiwan in 2002, later acquired by D-Link Corp, and O' Prueba Inc. in 2018. His work on multihop cellular was the first along this line, and has been cited over 850 times and standardized into IEEE 802.11s, IEEE 802.15.5, IEEE 802.16j, and 3GPP LTE-Advanced. He authored a textbook, *Computer Networks: An Open Source Approach* (McGraw-Hill, 2011) (www.mhhe.com/lin), with Ren-Hung Hwang and Fred Baker. His research interests include network security, wireless communications, and network softwarization.

Dr. Lin is an IEEE Distinguished Lecturer (2014–2017), ONF Research Associate, and was the recipient of 2017 Research Excellence Award and K. T. Li Breakthrough Award. He has served or is serving on the editorial boards of several IEEE journals and magazines, and is the Editor-in-Chief of IEEE Communications Surveys and Tutorials (COMST).

Yu-Kuen Lai (S'03–M'06–S M'14) received the M.S. and Ph.D. degrees in electrical and computer engineering from North Carolina State University (NCSU) at Raleigh, NC, USA in 1997 and 2006, respectively.

He is an Associate Professor with the Electrical Engineering Department, Chung-Yuan Christian University (CYCU), Chung-Li, Taiwan. From 1997 to 2002, he was a Senior ASIC Design Engineer with Delta Networks, Inc., and Applied Micro Circuit Corporation (AMCC) at Research Triangle Park, NC, USA. He served as the Electronic Communication Officer for the IEEE Taipei Section in 2013. He is currently the Director of the Information Technology Division at C.Y. Chang Memorial Library at CYCU, Taoyuan City, Taiwan. His research interests include network processor architecture, streaming data processing, network traffic analysis, FPGA systems design, and computer network security.

Dr. Lai was the recipient of the CYCU Distinguished Teaching Award and CYCU Outstanding Teaching Award in 2011 and 2017, respectively.

Yung-Liang Tsou received the M.S. degree in computer science from National Chiao Tung University, Chung-Li, Taiwan in 2017.

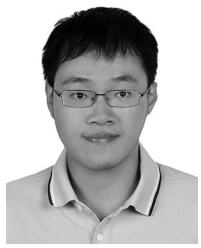
His research interests include software-defined networking (SDN), network testing, and 4G LTE.

Yuan-Cheng Lai (A'05–M'13) received the Ph.D. degree in computer science from National Chiao Tung University, Hsinchu, Taiwan, in 1997.

In August 1998, he joined the faculty of the Department of Computer Science and Information Science, National Cheng Kung University, Tainan, Taiwan. In August 2001, he joined the faculty of the Department of Information Management, National Taiwan University of Science and Technology (NTUST), Taipei, Taiwan, where he has been a Professor since February 2008. His research interests include performance analysis, protocol design, wireless networks, and web-based applications.

En-Cheng Liou received the Ph.D. degree in communication engineering from Yuan Ze University, Taiwan, in 2016.

He is a Project Manager with Network Benchmarking Lab at National National Chiao Tung University, Taiwan. His research interests include wireless networking, multimedia communication, cross-layer design, 4G LTE downlink scheduler, and software-defined networking (SDN).



Yita Chiang received the M.S. degree in network engineering from National Chiao Tung University in 2010.

Since then, he worked at Network Benchmarking Lab, Taiwan, where he is currently a Senior Engineer. His research interests include test automation of network devices.