MSAID: Automated Interference Detection for Multiple SDN Applications

Yahui Li*, Zhiliang Wang†‡, Jiangyuan Yao†‡, Xia Yin*‡, Xingang Shi†‡, and Jianping Wu*‡
*Department of Computer Science and Technology, Tsinghua University
†Institute for Network Sciences and Cyberspace, Tsinghua University
‡Tsinghua National Laboratory for Information Science and Technology (TNLIST)

***Abstract*—Multiple SDN applications can make several harmful interferences unintentionally, although each individual application may be properly developed. This paper proposes a Multiple SDN Applications Interference Detector (MSAID). To bridge the gap between the source code of applications and the actual interferences, we leverage symbolic execution and constraint solving to obtain how the event handler handles the input messages. We then analyze the complex interaction of multiple applications and present novel methods to identify the interferences. Finally, we evaluate its correctness and prove its usefulness with a series of SDN applications.**

I. Introduction

With the emergence of SDN applications stores, e.g. HP's SDN App Store, and the recent success of open-source controllers, e.g., ONOS, OpenDaylight, an increasing number of SDN applications are deployed. Unfortunately, the diversity of applications makes network control complex and error-prone. SDN applications are generally designed to be event-driven with the implementation comprising *event handlers*. These handlers (un)install rules for different network events, e.g., packet-in event. Generally speaking, these programs can perform arbitrary computation and maintain arbitrary state. Multiple applications can intentionally or unintentionally install conflicting rules on the same switch (e.g. rules that have intersecting match fields but conflicting action fields), resulting in harmful interferences [1]. For example, in Fig.1, two applications determine different actions for a flow from host A to host B. $APP1$ instructs the flow to go through the path, while $APP2$ installs rules to drop the flow. The priority and match fields of $APP1$'s rule unexpectedly equal $APP2$'s rule. Thus, only one application will install its rule on the switch, as specified in the OpenFlow specification.

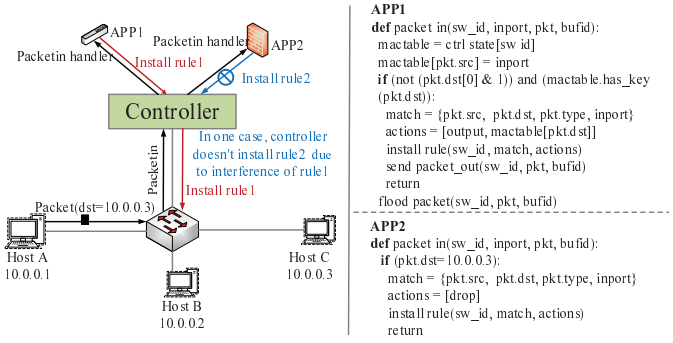

Fig. 1. An example of two applications, which have interferences.

Some systems rely on the application prioritizes to avoid conflicts, which assume that the application developers are aware of others, or all rule installations go through a module in the controller. Even though having clear priorities and explicit guidelines, how to combine the outcome of different applications is a long-term solution. In the absence of such explicit guidelines on how the rules generated by one application takes precedence over others, identifying bad interferences is the only way to avoid unwanted behavior. To date, as in Google's OpenFlow-based B4, developers design competing logic into a monolithic application to avoid conflicts. Unfortunately, these designs are not reusable, extensible, or reliably maintainable [2]. Efforts on SDN application verification, e.g. NICE [3], check each individual application with model checkers or theorem provers. However, they cannot capture the states of the multiple applications. Others, e.g. Frenetic [4], aim to build a correct by-design control plane. While worthwhile in the long term, those efforts cannot reason about existing applications.

II. Methodology

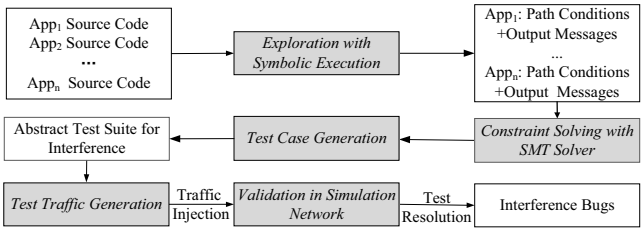

Fig. 2. Overview of how MSAID works.

To address above problems, we propose a systematic checking framework, called MSAID. As illustrated in Fig. 2, the high-level idea is to (i) explore the individual application to systematically find which inputs trigger which output OpenFlow messages, and then (ii) analyze the output messages of the multiple applications to detect the bad interference. It further employs the validation phase to confirm the interferences.

A. Symbolic Execution of SDN Applications

MSAID leverages recent advances in program analysis techniques, *symbolic execution*, to explore the source code of individual application. Symbolic execution runs a program with symbolic variables as inputs. It exercises all possible code paths and records the path condition of the symbolic variables

978-1-5090-6501-1/17/$31.00 ©2017 IEEE

on each path. To make the execution more tractable, we optimize symbolic inputs with domain knowledge: (i) structuring inputs to leverage the lazily-initialization technique; (ii) minimizing number and scope of symbolic variables; (iii) choosing the appropriate number of inputs to provide satisfactory coverage. More formally, for applications $\{A_1, A_2, ..., A_i, ..., A_n\}$, we denote the execution results with $SE_{A_i} = \{< c_{i1}, r_{i1} >, < c_{i2}, r_{i2} >, ..., < c_{ik}, r_{ik} > ..., < c_{im_i}, r_{im_i} >\}$ $(i = 1, 2, 3, ..., n)$, $(k = 1, 2, 3, ..., m_i)$, where n is the total number of the analyzed applications and m_i is the number of the code paths of the ith application A_i. c_{ik} are the path conditions and r_{ik} are the output messages, when executing the kth code path of the ith application. In this way, we can obtain which inputs cause which output messages for each application. Table I shows the execution results of the applications in Fig.1. E.g., if the packet satisfies condition c_{11} (its destination is a broadcast address), $APP1$ outputs message r_{11} (flooding this packet).

TABLE I
PART OF SYMBOLIC EXECUTION RESULTS OF $APP1$ AND $APP2$ (IN FIG.1). λ REPRESENTS THE SYMBOLIC PACKET.

Application	Path condition	Output message
$APP1$	$c_{11} : \lambda.dst \in broadcast$	$r_{11} : flood\ packet(sw\_id, \lambda, bufid)$
	$c_{12} : \lambda.dst \notin broadcast \wedge !(mactable.haskey(\lambda.dst))$	$r_{12} : flood\ packet(sw\_id, \lambda, bufid)$
	$c_{13} : \lambda.dst \notin broadcast \wedge mactable.haskey(\lambda.dst)$	$r_{13} : install\ rule(sw\_id, match, output, mactable[\lambda.dst])$
$APP2$	$c_{21} : \lambda.dst = 10.0.0.3$	$r_{21} : install\ rule(sw\_id, match, drop)$
	$c_{22} : \lambda.dst \neq 10.0.0.3$	$r_{22} : \phi\ (do\ nothing)$

B. Interference Detection

We then explore the solution to identify interferences by leveraging the results of symbolic execution. When checking application A_i and application A_j, MSAID detects interferences as follows: (i) For $\forall < c_{ik}, r_{ik} > \in SE_{A_i}$, it solves c_{ik} with off-line satisfiability solver (e.g., Z3bitolver). If the solver can obtain a concrete value cv that satisfies the constraints, there exists at least one packet can be processed by the application. It then concretes the symbolic fields of the corresponding output messages r_{ik} with cv. SE_{A_j} are similarly processed. (ii) It extracts and solves each conjunction of path constraints $c_{ik} \wedge c_{jp}$, and detects whether r_{ik} and r_{jp} instruct the same switch install rules with intersecting math and conflicting action fields, where $\forall < c_{ik}, r_{ik} > \in SE_{A_i}$, $\forall < c_{jp}, r_{jp} > \in SE_{A_j}$. If it detects an interference between r_{ik} and r_{jp}, it adds the concrete value which satisfies the path condition $c_{ik} \wedge c_{jp}$ to the abstract test suite. (iii) To verify whether the test cases trigger the bad interferences, it further translates them into concrete test traffic and employs empirical testing. E.g., after processing the results in Table I, MSAID obtains test cases {packet (src=10.0.0.1, dst=10.0.0.3), packet (src=10.0.0.1, dst=10.0.0.2)}. Then, in validation, it uncovers an interference with test traffic from 10.0.0.1 to 10.0.0.3.

III. IMPLEMENTATION AND EVALUATION

We have implemented the above framework in Java. All experiments are performed on a machine with 16GB of RAM and Intel Core CPU running at 3.40 GHz. All analyzed applications are included in Floodlight and ONOS. MSAID detects a broad spectrum of interferences, a few of which are presented in Table II. We also evaluate how it scales with the different applications. Fig. 3(a) shows that it can find interferences within a few minutes even for applications with thousands of lines of code. As a bug finding tool, MSAID does not produce false positives: each identified interference is evidence of divergent behavior. It might have false negatives because of the incomplete symbolic execution. In Fig. 3(b), the code coverage of more than 70% applications is over 72.6%. This is a worthwhile trade-off as operators can increase the number of input variables or repeat tests for greater coverage.

MSAID allows checking of unmodified applications to identify interferences automatically. It is a combination of program analysis, testing and domain knowledge, which can help build a highly reliable network.

TABLE II
EXAMPLES OF KNOWN AND NEW INTERFERENCES OF APPLICATIONS.

No.	Description
1	In [1]: Rules in routing application make the flow from external host reach internal web server, rendering firewall application ineffective.
2	In [2]: The preferred server in load balancer application is unexpectedly quarantined by firewall application.
3	New Interference in Floodlight: Clients cannot connect with the server pool in load balancer application, due to the wrong logic rules in virtual network filter (VNF) application.
4	New Interference in ONOS: Rules in Virtual Private LAN Service (VPLS) unexpectedly divide host A and host B into two virtual networks, remending SDNIP's policy (A to B forwarding) ineffective.

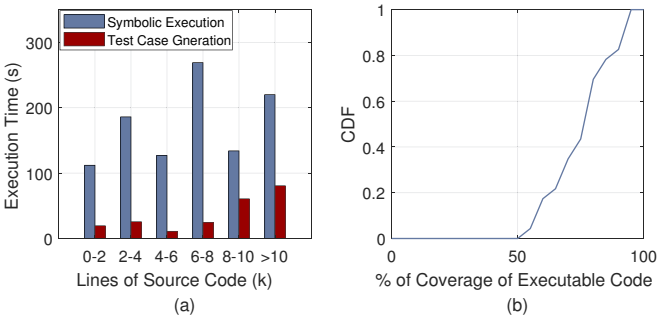

Fig. 3. Execution with varying applications. In (a), abscissa is the total lines of code of multiple applications in one application coexistence scenario.

ACKNOWLEDGMENT

This work is supported by the National High Technology Research and Development Program of China (863 Program) No. 2015AA016105.

REFERENCES

[1] R. Durairajan, J. Sommers, and P. Barford, "Off: Bugspray for openflow," in *ACM HOTSDN 2014*.

[2] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the software-defined network control layer," in *NDSS 2015*.

[3] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A nice way to test openflow applications," in *USENIX NSDI 2012*.

[4] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, and etc., "Frenetic: a network programming language," in *ACM ICFP 2011*.