Computer Networks 153 (2019) 49–62

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

MSAID: Automated detection of interference in multiple SDN applications

Yahui Li[a], Zhiliang Wang[b,c,*], Jiangyuan Yao[d], Xia Yin[a,c], Xingang Shi[b,c], Jianping Wu[a,c], Han Zhang[e]

[a] Department of Computer Science and Technology, Tsinghua University China
[b] Institute for Network Sciences and Cyberspace, Tsinghua University China
[c] Beijing National Research Center for Information Science and Technology, Beijing, China
[d] College of Information Science and Technology, Hainan University China
[e] School of Cyber Science and Technology, Beihang University China

ARTICLE INFO

Article history:
Received 12 July 2018
Revised 7 January 2019
Accepted 31 January 2019
Available online 5 February 2019

Keywords:
Multiple SDN applications
Interferences
Formal methods

ABSTRACT

Software-defined networking (SDN) enables control programs from diverse parties to generate network policies independently. However, multiple SDN applications can cause several unintentionally harmful interferences even though each individual application is properly developed. Unfortunately, the existing SDN network verification and testing frameworks do not take interferences between applications into account. This paper proposes a multiple SDN applications interference detector (MSAID). To bridge the gap between the applications' source code and the actual negative interferences, we leverage symbolic execution and constraint solving to evaluate how the event handler of the applications handles the input messages. Then, we analyze the complex interaction behavior of multiple applications, and we develop novel algorithms to identify the interferences of multiple applications. In addition, these algorithms help determine the traffic affected by the interferences, which can help users avoid unwanted behavior. We evaluate the correctness and performance of the detector, and we demonstrate its usefulness with a variety of SDN applications. Our evaluation shows that MSAID can detect both new and known interferences in a few minutes for popular SDN applications.

© 2019 Elsevier B.V. All rights reserved.

1. Introduction

With its decoupling of the data plane and the control plane, software-defined networking (SDN) promises to ease the deployment of new network functions. The network control logic in SDN is logically centralized in the controller, whereas in traditional networks, it is defined by the routing protocols running in devices that work with the information of network topology and device configurations. OpenFlow is one of most well-known southbound interfaces, which enables the controller to determine the path of packets across switches. By programming and composing SDN applications,[1] network operators can realize a series of critical policies. The availability, security, and performance of the network depend on the correct implementation of these policies.

With the recent success of open-source controllers, e.g., OpenDaylight [1] and ONOS [2], and the emergence of SDN application stores [3], increasingly more SDN applications are being developed and deployed on controllers. These applications are designed to be event-driven, and they are implemented with *event handlers*. After receiving input OpenFlow messages, these event handlers respond with the corresponding output OpenFlow messages, which will (un)install rules in switches to react to a number of network events, e.g., packet arrival (packet-in), link activation, switch-down or switch-up.

To implement devised network polices, multiple applications (e.g., firewall and load balancer) from various parties are typically deployed on a controller within an OpenFlow network [4]. These applications modify multiple aspects of the network by (un)installing rules. In fact, the diversity of applications indeed makes network control more complex and more error-prone. Multiple applications could intentionally or unintentionally lead to negative interferences where rules installed by different applications intersect but have conflicting actions, which could lead to serious consequences [5]. These applications may consist of non-secure applications (e.g., in the second example in Section 2, network traffic from the adversary-controlled external IP address suc-

* Corresponding author.
E-mail address: wzl@cernet.edu.cn (Z. Wang).

[1] We use 'SDN applications' and 'SDN program' interchangeably in this paper.

https://doi.org/10.1016/j.comnet.2019.01.042
1389-1286/© 2019 Elsevier B.V. All rights reserved.

ceeds in communicating with a host in the internal network because of the non-secure application) that are operating in a malicious manner. This problem is prevalent in SDN architectures. Some systems [6,7] aim to avoid these conflicts by deciding the priorities of the applications, and they always assume that the developers of the applications are aware of the others. Even with having clear guidelines on how the rules generated by one application take precedence over others, how to combine the outcomes of multiple applications is not tractable. In the absence of such explicit guidelines, identifying harmful interferences is the only way to avoid unwanted behavior.

Unfortunately, the existing verification works on SDN applications cannot check the interferences. *i*) Many verification and testing tools (e.g., VeriCon [8] and NICE [9]) are designed to check the correctness of each individual SDN application. However, they are insufficient for detecting interferences because they are unable to capture the states of the multiple SDN applications. *ii*) Data plane verification tools (e.g., HSA [10] and VMV [11]) verify invariant violations for the specific data plane, but they cannot find interference errors proactively before the applications are deployed [10–13]. *iii*) Correct-by-design control plane (e.g., Netcore [14] and Frenetic [15]) allows the composition of applications (via the northbound API of the controller) and attempts to provide explicit guidance to the controller on how to resolve potential conflicts. Although these efforts are worthwhile, they cannot reason about existing developed applications. *iv*) Technically, some tools (e.g., FLOVER [16]) have been proposed to check whether OpenFlow flow rules satisfy non-bypass security properties (including these interferences). However, they check after the applications install the conflicting flow rules. *v*) Moreover, some extensions (e.g., FortNOX [6] and SE-Floodlight [7]) for specific controllers (e.g., NOX and Floodlight [17]) have been proposed based on role-based authorization and security constraint enforcement. However, they rewrite or extend the kernel of the controllers with specific mechanisms, which are too inflexible and complex to be applied to other varieties of controllers. Therefore, what is critically missing today is a systematic and automated methodology to check the interferences of multiple SDN applications before the applications are deployed.

Detecting the interferences of multiple SDN applications is not an easy task. As the behavior of an SDN program depends on a large environment, including various network events, the state space of multiple SDN applications is very large, which makes verifying multiple applications with formal verification methods (e.g., model checking) ineffective. Fortunately, we observe that interferences are generally caused by the rules with intersecting match fields and conflicting action fields (i.e., an example in Fig. 1). With the information about which input OpenFlow messages cause which output OpenFlow messages in each individual application, we can reason about the interferences of multiple applications. To achieve this goal, an extension of the *symbolic execution* engine can help to reason about the corresponding output messages of the input messages in each application. Symbolic execution, a well-known program analysis technique, runs a program with symbolic variables as inputs and explores the logic of all paths in the program.

Toward a solution for detecting the interferences of multiple SDN applications, we develop a systematic framework called multiple SDN applications interference detector (MSAID). First, MSAID explores the source code of each individual application by the symbolic execution technique. Then, it identifies the interferences of multiple applications based on novel algorithms. In addition, it reasons about the traffic affected by the interferences, which can help users avoid harmful unwanted behavior. MSAID is a combination of a program analysis technique and domain knowledge, which can help to build a highly safe and reliable network. The contributions of this paper are as follows:

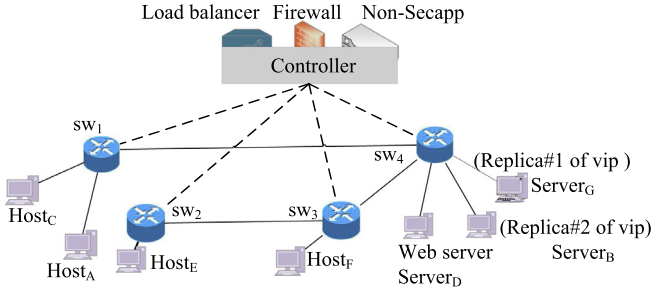
- Develop a framework for detecting the interferences of multiple SDN applications, which is in contrast to prior approaches that only check the correctness of individual applications.
- Explore the application's event handlers based on scalable symbolic execution. Present effective approaches to automatically analyze the complex interaction behavior of multiple applications and determine the (in)direct interferences.

The remainder of this paper is organized as follows. We state our motivation in Section 2. Section 4 provides an overview of our approach, and Section 5 contains the detailed description. We implement and evaluate our approach in Section 6 and Section 7. Finally, we present concluding remarks in Section 8.

2. Motivation

In this section, we present some motivating examples to illustrate how interference errors arise. We start with a toy example. Three applications, namely, a load balancer, firewall, and non-Secapp, developed by different developers are deployed on the same controller (from the examples in [5,7]). Note that the load balancer performs balancing among a set of replicas identified by a virtual IP (VIP) address. For packets destined to a VIP, the load balancer selects a particular host and installs forwarding rules along the path. The firewall installs rules to permit or block flows based on commercial or security considerations. As depicted in Fig. 1, there are two replicas of VIP ($Server_B$ and $Server_G$), a web server $Server_D$, and four hosts ($Host_A$, $Host_C$, $Host_E$, and $Host_F$) in the network. The firewall inserts rule 1 to block the flows from $Host_A$ to a local server $Server_B$ because of commercial considerations. Unfortunately, $Server_B$ blocked by the firewall becomes the target for new connection flows by the load balancer because server $Server_B$ is now loaded the least in the VIP pool. The load balancer installs rule 2 to forward $Host_A$'s requests to $Server_B$. Additionally, the priority and match fields of rule 2 unexpectedly equal those of rule 1. If the process priority of the *packet_in* event handler in the load balancer is higher, rule 2 will be installed on the switches first. Then, rule 1 cannot be installed because of the same priority and match fields, which renders the policy of the firewall *ineffective* and results in a direct interference.

Next, we consider another simple scenario (from the examples in [5]). The firewall installs rule 3 to block traffic from an external host $Host_C$ to an internal web server $Server_D$. At a certain time, the non-Secapp implements rule 4, rule 5, and rule 6. Rule 4 rewrites the source IP address of a packet to $Host_E$ through a *Set* operation if the packet is delivered from $Host_C$. Rule 5 rewrites the destination IP address of a packet to $Server_D$ if the packet is



Rule1: (add, sw_1, e_1(p.src = $Host_A$, p.dst = $Server_B$) → drop);
Rule2: (add, sw_1, e_2(p.src = $Host_A$, p.dst = $Server_B$) → fwd(3)) ;
Rule3: (add, sw_1, e_3(p.src = $Host_C$, p.dst = $Server_D$) → drop);
Rule4: (add, sw_1, e_4(p.src = $Host_C$, p.dst = *) → Set (p.src = $Host_E$), output(table));
Rule5: (add, sw_1, e_5(p.src = *, p.dst = $Host_F$) → Set (p.dst = $Server_D$),output(table));
Rule6: (add, sw_1, e_6(p.src = $Host_E$, p.dst = $Server_D$) → fwd(3)));

Fig. 1. An example of applications that have interferences.

destined to $Host_F$. Rule 6 simply allows packet forwarding from $Host_E$ to $Server_D$. Unfortunately, now packets from $Host_C$ destined to $Host_F$ can reach the web server $Server_D$, bypassing the firewall. A synthetic rule composed of the three rules in the non-Secapp, $(add, sw_1, e((\lambda.src = Host_C,\ \lambda.dst = Server_D) \rightarrow fwd(3)))$, conflicts with rule 3 in the firewall, rendering the firewall *ineffective*. The inherent flexibility of the *Set* action can introduce some indirect interferences.

Such interferences may occur frequently. Identifying potential bad interferences is an effective way to avoid unexpected behavior. Although necessary, checking the aforementioned interferences before these applications are pushed to the system is not an easy task. As discussed earlier, the existing verification tools [8–13] are insufficient because they are unable to capture the intersection state of multiple SDN applications. Fortunately, we observe that interference faults arise from two scenarios. First, *direct interference* occurs if multiple applications inform the same switch to install rules that have intersecting match fields and conflicting action fields. Second, the *Set* action of a rule in a switch can rewrite the header of a matching flow, leading to complex synthetic rules. *Indirect interference* occurs if the synthetic rules of one application have direct interferences with the rules of other applications.

3. Related work

Control plane verification: The work closest to our work is verifying and testing SDN applications. VeriCon [8] verifies infinite-state SDN applications through theorem proofs. NICE [9] combines model checking and symbolic execution to detect bugs in SDN applications. SDNRacer [18] detects concurrency-induced errors, such as update consistency errors, based on a happens-before model. As discussed earlier, these approaches are designed to check the correctness of each individual application. MSAID is complementary to these efforts and can capture interferences of multiple SDN applications.

Data plane verification: There is rich literature on verifying the reachability behavior of the data plane (e.g., HSA [10], NOD [19] and VMV [11]). These tools check whether a network invariant (e.g., reachability) would be satisfied. The result of such verification, however, is valid only for the specific data plane. Unlike MSAID, they cannot find errors proactively before the applications are deployed. SOFT [20] tests the interoperability of OpenFlow switches via symbolic execution, which is different from our approach.

Conflict detection: Volpano et al. [21] detect and resolve control conflicts in the SDN control plane with a deterministic finite-state transducer. However, the models of the network functions are manually obtained with expert knowledge. Our work automatically checks the interferences of application coexistence, which is in contrast to their work. Statesman [22] composes uncoordinated application actions using distinct views of the network state. Unfortunately, these designs are not extensible because application developers are not always aware of others. FortNOX [16] leverages role-based authorization and security constraint enforcement to detect these conflicts. SE-Floodlight [7] extends Floodlight in a similar spirit. However, they both need to specifically change the kernel of the controller, which is inflexible. Furthermore, they cannot answer the question of whether interference occurs among applications before applications are pushed to the system.

Clean-slate controller design: Efforts, e.g., Frenetic [15], Netcore [14] and Pyretic [23], aim to build a correct-by-design control plane. The compiler in these controllers can compose these rules and detect conflicts, but they work for a specific northbound interface language. Although worthwhile in the long term, they cannot reason about existing applications. Our work automatically checks the interferences of applications, which differs from their work.

Detection of conflicting firewall policies: The same problem arises in the space of the firewall. Some researchers focus on the conflict classification and analysis of distributed firewall policies [24]. Others leverage the architecture of SDN for security policy enforcement [25]. However, those works require human modeling of policies, which is inflexible. MSAID is an automatic detection approach with the source code of applications. Furthermore, our framework can check more than conflicting firewall polices.

Symbolic execution: The symbolic execution used in our work is inspired by its success in network verification. PIC [26] detects non-interoperabilities in protocol implementations based on symbolic execution. Dobrescu and Argyraki [27] propose writing the router software data plane in a specific style and verify correct properties based on symbolic execution. BUZZ [28] builds the FSM model from NF's code with domain knowledge and generates test cases via symbolic execution. NFactor [29] leverages symbolic execution to extract models for middlebox. We argue that modeling or rewiring programs such as theirs are not extensible or maintainable. MSAID aims to detect interferences of applications, which is in contrast to these works.

An early version of this work was presented in [30]. We presented the description of interferences in multiple applications, the profile of the detection framework and the preliminary detection results. In this paper, we have added more content, such as examples of interferences in multiple applications, a formal description of interference (including direct and indirect interference), details of the symbolic execution of applications, details of the interference detection approach, details of the implementation, and a more complete evaluation of its correctness and performance.

4. Overview

We provide the intuitions behind the choices of our approach in this section. In the verification field, researchers typically leverage formal methods (e.g., theorem proofs and model checking) to verify the correctness of each individual SDN application [8,9], but attempting to verify multiple applications with formal verification tools is ineffective because of the extremely large state space of SDN applications.

Fortunately, based on the aforementioned observations in Section 2, for each individual application, after reasoning about which input OpenFlow messages cause which output OpenFlow messages, we can leverage these results to identify the latent interferences of multiple applications. However, the task is fundamentally challenging in following aspects:

i) *The state space of possible input OpenFlow messages is extremely large.* As discussed earlier, applications are data plane driven. After receiving input messages, the application can correspondingly react to a vast number of possible packets (with different source and destination MAC addresses, source and destination IP addresses, and port numbers). Searching every possible message in the state space is neither feasible nor scalable.

ii) *As the event handlers in the applications are various and complicated, deciding input messages that can systematically exercise all code branches is complex.* To reason about the corresponding output message, we need to track a variety of event handlers in each application. A naive approach is to execute the program with different input messages one by one. Clearly, it cannot cover the different branches of the program systematically.

iii) *With the input messages and corresponding output messages for each individual application, identifying interference is complex because of OpenFlow's introduction of the Set action.* With this action, applications are able to instruct a switch to change the header attributes of a matching flow. The changed flow might continue matching with other flow entries on the switch, which can result in indirect interferences.

Table 1
Part of the symbolic execution results of non-Secapp and firewall (in Section 2). λ represents the symbolic packet.

Application	Path condition		Output message	
non− Secapp	c_{11}	$\lambda.src = Host_C \wedge \lambda.dst \neq Host_F$	r_{11}	$o_1^{11}: (add, sw_1, e_4((\lambda.src = Host_C, \lambda.dst = *) \rightarrow Set(\lambda.src = Host_E),\ output(table)))$ $o_2^{11}: (add, sw_1, e_6((\lambda.src = Host_E, \lambda.dst = Server_D) \rightarrow fwd(3)))$
	c_{12}	$\lambda.src \neq Host_C \wedge \lambda.dst = Host_F$	r_{12}	$o_1^{12}: (add, sw_1, e_5((\lambda.src = *, \lambda.dst = Host_F) \rightarrow Set(\lambda.dst = Server_D)\ output(table)))$ $o_2^{12}: (add, sw_1, e_6((\lambda.src = Host_E, \lambda.dst = Server_D) \rightarrow fwd(3)))$
	c_{13}	$\lambda.src = Host_C \wedge \lambda.dst = Host_F$	r_{13}	$o_1^{13}: (add, sw_1, e_4((\lambda.src = Host_C, \lambda.dst = *) \rightarrow Set(\lambda.src = Host_E)\ output(table)))$ $o_2^{13}: (add, sw_1, e_5((\lambda.src = *, \lambda.dst = Host_F) \rightarrow Set(\lambda.dst = Server_D)\ output(table)))$ $o_3^{13}: (add, sw_1, e_6((\lambda.src = Host_E, \lambda.dst = Server_D) \rightarrow fwd(3)))$
firewall	c_{21}	$\lambda.src = Host_C \wedge \lambda.dst = Server_D$	r_{21}	$o_1^{21}: (add, sw_1, e_3((\lambda.src = Host_C, \lambda.dst = Server_D) \rightarrow drop))$
	c_{22}	$\lambda.src \neq Host_C \vee \lambda.dst \neq Server_D$	r_{22}	$o_1^{22}: (add, sw_1, e_6((\lambda.src = Host_E, \lambda.dst = Server_D) \rightarrow fwd(3)))$

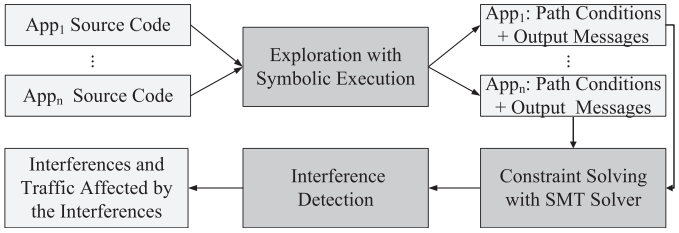


Fig. 2. The framework summarizes a high-level image of MSAID's operation.

To address these challenges, we explore a solution, MSAID, to detect interferences. MSAID operates directly on the source code of the *unmodified* OpenFlow applications to be verified. MSAID does not need to extend the kernel or structure of the controller. After verification, it automatically outputs all possible interferences among these applications. Here, we present the high-level view of our approach.

i) *To address the first search scalability challenge, we use symbolic execution, which is a well-known technique to address state-space explosion in program analysis*. It identifies which inputs would exercise different code paths through the program rather than explore all possible inputs.

ii) *With an extension of the symbolic execution engine, we can also address the second coverage challenge*. It runs a program with symbolic inputs and explores all feasible paths in the program. With an extension of the engine, we can reason about the corresponding output messages of the input messages.

iii) *To address the third challenge, we propose novel algorithms to generate synthetic rules for these output messages.* Then, we can identify the (in)direct interferences among all applications based on the standards of the OpenFlow protocol.

Fig. 2 provides an illustration of the MSAID framework. First, MSAID leverages scalable *symbolic execution* to systematically identify code paths in each individual application's source code. In addition to collecting code paths, it records the corresponding output results of each code path. Thus, it can identify the inputs and the corresponding output OpenFlow messages for each individual application. Table 1 provides the execution results of the non-Secapp and firewall from Section 2. For example, if the input satisfies the condition c_{21}: $\lambda.src = Host_C \wedge \lambda.dst = Server_D$, then the output message r_{21} is $(add, sw_1, e((\lambda.src = Host_C,\ \lambda.dst = Server_D) \rightarrow drop))$. For the input packets whose source is $Host_C$ and destination is $Server_D$, the firewall will add a flow entry on switch sw_1 to drop these packets.

Next, for multiple applications, MSAID solves the path conditions with the *constraint solver*. Note that the *Set* action in some output messages can lead to indirect interference via complex synthetic rules. Hence, it generates synthetic rules for these output messages according to the OpenFlow specification. Then, it cross-checks all distinct output messages of multiple applications to identify interferences and the traffic affected by the interferences. For the aforementioned example, MSAID generates synthetic rules for the output messages in Table 1 and computes the affected traffic: packets (src = $Host_C$, dst = $Host_F$) and packets (src = $Host_C$, dst = $Server_D$). The traffic triggers output messages r_{13} and r_{21}, which instruct switch sw_1 to install rules with intersecting match and conflicting action fields. In the example, the traffic from $Host_C$ to $Host_F$ can finally reach the web server $Server_D$, which indicates that the policy of the firewall is ineffective due to the interference of the non-Secapp.

Note that because interference detection in the framework depends on the code paths of SDN applications, MSAID does have an inherent limitation in handling applications that do not perform flow decisions embedded on source code. In addition, we assume that users would ideally have access to all parts of the source code of application implementations. Clearly, for commercial concerns, there are some proprietary and closed-source SDN applications. To handle these applications, users of the detection tool need to ask for help from the developers and providers of the applications. Specifically, the interference detection (the second phase) depends on the output of the symbolic execution (the first phase). In the first phase, the symbolic execution runs locally on each application's source code. When the source code of an application is unavailable, users can provide the detection tool to the developer and provider of the application and ask them to provide the outputs of the symbolic execution.

5. Interference detection

5.1. Problem definition

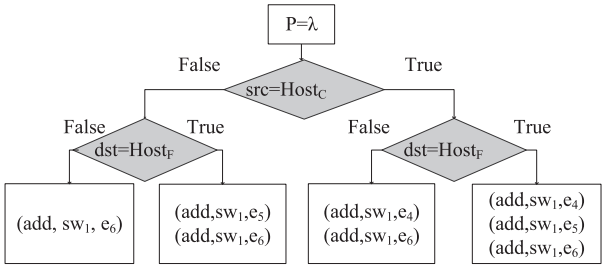
Based on the OpenFlow specification, we first define a formal model of the operations performed by the SDN applications and the switches in the network. Then, we use this formalization to specify the interference of multiple applications.

5.1.1. Forwarding behavior of data plane in SDN network

Packet. We first define a packet p by a set of header fields, $p = [f^1, f^2, \ldots, f^j, \ldots, f^d]$, where f^j is the value of field j ($j = 1, 2, 3, \ldots, d$).[2] In particular, header space is a set of all possible packet headers, denoted as $\{0, 1\}^l$ space, where l is the length of the header [31]. The switch processes the packet arriving at its port in accordance with the *flow table*. A flow table is a sequence of flow entries that determine the behavior of the switch.

Flow entry. Flow entry e can be represented as an ordered triplet (e^P, e^M, e^A), where e^P, e^M and e^A refer to priority field, match field and action field, respectively. A flow entry contains t fields (e.g., in OpenFlow V1.3, $t = 13$). The match field e^M refers to a

[2] The set of fields consists of IP source, IP destination, and so forth. We denote the source IP field as *p.src* (e.g., in Table 1). *p.dst* is defined similarly.




```
1 If(p.src == Host_C){
2    (add, sw_1, e_4((p.src = Host_C, p.dst = *) → Set (p.src =  Host_E), output(table)));
3 }
4 If(p.dst == Host_F){
5    (add, sw_1, e_5((p.src = *, p.dst = Host_F) → Set (p.dst = Server_D), output(table)));
6 }
7 (add, sw_1, e_6((p.src = Host_E, p.dst = Server_D) → fwd(3)));
```

Fig. 3. Symbolic execution of the *packet_in* handler in the non-Secapp application in Section 2.

list of fields in a flow entry (e.g., MAC source, IP source, Ether type). We omit fields that do not affect forwarding for simplicity. e^P specifies the preference of entries for the case where the packet matches more than one flow entry. Fig. 3 presents some examples for entries. For example, line 7 provides an entry e_6, where e^P is the default value, which is omitted in the pseudo code; e^M refers to $src = Host_E$; $dst = Server_D$; and e^A refers to action $fwd(3)$.

When a packet is matched against e^M, the action $e^A \in A$ can be performed, where A can contain one or more of the following actions. *i*) Drop the packet corresponding to this match (denoted as *drop*). *ii*) Forward the packet to one or more output ports *pt* (denoted as $fwd(pt)$). Specifically, output to a table (denoted as $output(table)$) instructing that the flow needs to be rematched by the flow table. *iii*) Rewrite the header fields of a matching flow (denoted as *Set*). *iv*) Generate and send OpenFlow messages to the controller (e.g., the $packet_in(sw, pt, p)$ message carries the information that *p* has arrived at port *pt* of switch *sw*).

5.1.2. Interference of multiple SDN applications

SDN application. SDN enforces applications' policies through transforming them into flow entries. Applications are constructed as a set of event handlers (e.g., *packet_in* handler) that react to the messages reported by the switch (e.g., *packet_in* message). We can define an application as functions that react to an input OpenFlow message *im* with a set of output messages *os*.

Output OpenFlow message. Suppose that there are a set of applications to be checked, which are denoted as $A_1, A_2, A_3, \ldots, A_i, \ldots, A_n$, where n is the number of applications. Let os_i denote the output messages corresponding to an input message *im* in application A_i. $os_i = \{o^i_1, o^i_2, o^i_3, \ldots, o^i_j, \ldots, o^i_r\}$, $(j = 1, 2, 3, \ldots, r)$, where r is the number of output messages. If there are no such messages, the os_i is empty ϕ. An output message o^i_j is represented as a triplet $(o^i_j.op, o^i_j.sw, o^i_j.e)$, where $o^i_j.op \in \{add, mod, del\}$. $o^i_j.op$, $o^i_j.sw$, and $o^i_j.e$ respectively represent the operation, the switch, and the related entry. If $o^i_j.op$ is an *add* operation, it aims to add a new flow entry $o^i_j.e$ to the switch $o^i_j.sw$. In other words, it makes switch $o^i_j.sw$ apply $o^i_j.e^A$ to all packets that match the fields in $o^i_j.e^M$. If $o^i_j.op$ is a *mod* operation, it attempts to modify the entry $o^i_j.e$ in switch $o^i_j.sw$. If $o^i_j.op$ is a *del* operation, it attempts to delete the entries that match $o^i_j.e^M$ in switch $o^i_j.sw$. Table 1 provides some examples for output messages. For instance, for output message o^{11}_1, the $o^{11}_1.op$ refers to the *add* operation, the $o^{11}_1.sw$ refers to sw_1, and $o^{11}_1.e$ refers to entry e_4.

Synthetic rule. The *Set* action introduced in the OpenFlow protocol can introduce complex synthetic rules. Let os_{i_b} denote the synthetic rules of the output messages os_i. In particular, $os_{i_b} = \{o^{i_b}_1, o^{i_b}_2, o^{i_b}_3, \ldots, o^{i_b}_j, \ldots, o^{i_b}_{r_b}\}$, $(j = 1, 2, 3, \ldots, r_b)$, where r_b is the number of synthetic rules.[3] In addition, the synthetic rule $o^{i_b}_j$ has the same structure as the output message o^i_j, represented as a triplet $(o^{i_b}_j.op, o^{i_b}_j.sw, o^{i_b}_j.e)$. Fig. 4 shows an example of a synthetic rule.

Interference. Interference occurs between application A_m and application A_n if os_m and os_n instruct a switch to perform conflicting actions on the same flow. Formally, for the same input OpenFlow message *im*, a direct interference arises if o^m_i and o^n_j, where $\forall o^m_i \in os_m$ and $\forall o^n_j \in os_n$, satisfy the following conditions: *i*) $o^m_i.op \neq del$, $o^n_j.op \neq del$. It dictates that o^m_i and o^n_j add or modify flow entries on the switch. *ii*) $o^m_i.sw = o^n_j.sw$, which dictates that o^m_i and o^n_j update the flow entry on the same switch. *iii*) $o^m_i.e^M \cap o^n_j.e^M \neq \phi$, which indicates that packet headers exist that match the matching field of $o^m_i.e$ and that of $o^n_j.e$. *iv*) $o^m_i.e^A \neq o^n_j.e^A$. It dictates that $o^m_i.e$ and $o^n_j.e$ instruct the same communication flow with conflicting forwarding actions. In addition, indirect interference arises if o^m_i, o^n_j, $o^{m_b}_i$ and $o^{n_b}_j$, where $\forall o^m_i \in os_m$, $\forall o^n_j \in os_n$, $\forall o^{m_b}_i \in os_{m_b}$ and $\forall o^{n_b}_j \in os_{n_b}$, satisfy the following condition: one of the three pairs ($o^{m_b}_i$ and $o^{n_b}_j$, $o^{m_b}_i$ and o^n_j, and o^m_i and $o^{n_b}_j$) satisfies all the conditions of direct interference.

Note that the "$\cap$" (in condition *iii* above) is performed on the values of all fields in e^M. Let r^k_i refer to the range of values that are permissible for the *kth* field in $o^m_i.e^M$. If $\forall k, r^k_i \cap r^k_j \neq \phi$, then $o^m_i.e^M$ and $o^n_j.e^M$ are intersecting.

5.2. Symbolic execution of SDN applications

Symbolic execution [32] executes a program with symbolic variables as inputs. During the execution, code is executed normally until it reaches a branch instruction. The branch instruction can be denoted as a conditional expression *expr*. If the conditional expression *expr* depends on a symbolic value of the inputs, then program execution is logically forked into two paths at this point. One path is constrained to make the conditional expression *expr* false and the conditional expression *expr* in another path true. Internally, the symbolic execution engine invokes a constraint solver to verify the feasibility of each path. Then, it continues down all feasible paths and records the constraints of the symbolic variables on each path. These constraints are also called the path conditions.

5.2.1. Formulation of the execution

When symbolically executing an SDN application, we begin by constructing the input OpenFlow message.[4] For the sake of presentation, we show how to structure a symbolic *packet_in* message, whose main part is the incoming packet. We can customize symbolic fields in a packet $\lambda = [\lambda^1, \lambda^2, \ldots, \lambda^j, \ldots, \lambda^d]$ (e.g., the *j*th field of the packet is defined by a symbolic value λ^j). After executing the application symbolically, the execution engine automatically partitions the input space of symbolic packets into several subspaces. Furthermore, we modify the engine to log the corresponding output messages of the path conditions.

Formally, for application A_i, we denote the execution results as $SE_{A_i} = \{< c_{i1}, r_{i1} >, < c_{i2}, r_{i2} >, \ldots, < c_{ik}, r_{ik} >, \ldots, < c_{im}, r_{im} >\}$

[3] The details on generating synthetic rules are provided in Section 5.3.

[4] The input OpenFlow messages can be PACKET_IN, FLOW_REMOVED, PORT_MOD, BARRIER_REPLY and so on. The output messages include PACKET_OUT, BARRIER_REQUEST, and FLOW_MOD.

$(i = 1, 2, 3, \ldots, n)$, $(k = 1, 2, 3, \ldots, m)$. c_{ik} is the path condition logged, and r_{ik} is a set of the output messages produced when executing the kth code path of the ith application. m is the count of the code paths of the ith application. Specifically, c_{ik} is the disjunction or conjunction of all constraints of the symbolic input. We denote it as $c_{ik} = S_{ik}(\lambda^1) \diamond S_{ik}(\lambda^2) \diamond \ldots \diamond S_{ik}(\lambda^j) \ldots \diamond S_{ik}(\lambda^d)$, where $\diamond$ is the conjunctive ($\wedge$) or disjunctive ($\vee$) operation. $S_{ik}(\lambda^j)$ are the constraints of the symbolic variables λ^j. Let r_{ik} be the normalized output result when executing the path represented by c_{ik}. Because an SDN application can issue multiple messages in response to one network event, r_{ik} is a set of output messages, denoted as $\{o_1^{ik}, o_2^{ik}, \ldots, o_j^{ik}, \ldots, o_q^{ik}\}$, where q is the number of output messages in r_{ik}. Because of the symbolic inputs, after the execution of the symbolic engine, some fields in the messages of o_j^{ik} may be constructed as a symbolic value of the symbolic input. Given l fields matching the field of an flow entry, we represent the matching field of $o_j^{ik}.e$ as $o_j^{ik}.e^M = (x_{j1}, \ldots, x_{ji}, \ldots, x_{jl})$, where x_{ji} is a concrete value or an expression of the symbolic field in λ.

Fig. 3 illustrates the symbolic execution of the source code of an example application. Consider the non-Secapp application in Fig. 1. Lines 1–3 indicate that if the source address equals the address of $Host_C$, then it will install entry e_4 (which rewrites the source address of the matched packet to the address of $Host_E$ via the *Set* operation). Lines 4–7 are implemented similarly. We symbolically execute the implementation and log the output messages of the corresponding code path. The initial execution path is split into four execution paths. For example, if the *src* of the input equals the address of $Host_C$ and the *dst* equals the address of $Host_F$, then it adds flow entries e_4, e_5, and e_6 on switch sw_1. The corresponding execution results are shown in Table 1: If the input satisfies the path condition c_{13}, then the application will output message r_{13}.

Note that each r_{ik} (resulting from symbolic execution) encodes the output messages. To track all output messages in a program, we extend the symbolic execution engine to collect the corresponding messages. Based on the standards of the OpenFlow protocol, we extract the key fields of these messages to conduct the translation to abstract-level results. The code development is a one-time offline task.[5]

5.2.2. Optimization with domain knowledge

As is well known, code paths in symbolic execution increase exponentially with the size of the inputs (called the path explosion problem). To address this challenge, we optimize the symbolic inputs as follows.

Structuring inputs to leverage lazy initialization: For the sake of presentation, we show how to structure a symbolic *Packet_In* message, whose main part is the incoming packet. Creating a symbolic input with the entire packet structure is inefficient because that will cause the symbolic execution engine to find values for every field of the packet. The payload of the packet is a sequence of bits. Occasionally, it does not need to match packets against flow entries. Therefore, in this case, we can omit the payload of the packets, which can reduce the number of symbolic variables. Moreover, maintaining each header field as an individual symbolic variable can allow us to leverage the lazy initialization technology [34], which can greatly reduce the overhead for the constraint solver.

Minimizing number or scope of symbolic variables: We identify the subset of packet header fields and only make these symbolic values. The remaining fields are constructed as concrete values. For example, when checking the firewall, we make the HTTP-relevant fields concrete by assigning them a concrete value. In this scenario, we are not concerned with the value of the HTTP. Furthermore, users can constrain the possible values of header fields with their domain knowledge. For example, the value of 'eth_type' is limited. It is ineffective to make the value be totally symbolic. We can add constraints to limit the value of the variable. The MAC and IP addresses of the switches can be specified by the network topology if operators know the target scenarios to be deployed. Undeniably, there are some applications that take actions based on the content of the payload (e.g., deep packet inspection security-related solutions). For these applications, the symbolic variables include not only the header but also the payload of packets to make the analysis more complete.

[5] In addition, in symbolic execution, the presence of loops can cause a large number of paths to be explored, whereas this paper does not emphasize studying this problem. We leverage recent advances in symbolic execution [33].

5.3. Interference detection

In this subsection, we explore the solution to identify interferences based on the aforementioned results of symbolic execution. The solution consists of three steps: constraint solving, synthetic rule generation and interference identification.

Algorithm 1 shows how to detect interferences for multiple applications. This algorithm first solves the path conditions with a satisfiability solver, and it concretes the symbolic fields of the output OpenFlow messages using the value returned by the solver. Then, it generates synthetic rules for the output messages with the *Set* action. Finally, it identifies the interferences of the output messages from different applications according to the definition of interferences.

Algorithm 1: Interference detection for multiple applications.

Input: $SES := \{SE_{A_1}, SE_{A_2}, \ldots, SE_{A_i}, \ldots, SE_{A_n}\}$
Output: Traffic affected by the interferences: TSI

```
1  init TSI := φ;
2  foreach SE_{A_i} in SES do
3      foreach < c_ip, r_ip > in SE_{A_i} do
4          cv_ip := Solve (c_ip);
5          if cv_ip = φ then
6              Remove < c_ip, r_ip >;
7          foreach o_i^ip.e^M in r_ip do
8              Concrete (o_i^ip.e^M, cv_ip);
9      if one of o_j^ip.e^A in r_ip includes Set action then
10         r_ip := Update (r_ip);
11 SEC := Combine (SES);
12 foreach {SE_{A_i}, SE_{A_j}} in SEC do
13     foreach r_ip in SE_{A_i} do
14         foreach r_jq in SE_{A_j} do
15             if Interference (r_ip, r_jq) ≠ φ then
16                 TSI+ = Solve (cv_ip, cv_jq);
17 return TSI;
```

5.3.1. Constraint solving

SES is a collection of symbolic execution results for each application. For each SE_{A_i} in *SES*, $< c_{ip}, r_{ip} >$ denotes each pair of constraints and the corresponding output messages. As previously defined, c_{ip} is the disjunction or conjunction of all constraints of the symbolic input. To obtain a concrete packet that can satisfy formula c_{ip} (line 4), we need to compute the assignments that make a conjunctive normal form (CNF) or disjunctive normal form (DNF)

true. Note that solving the formula c_{ip} is NP-hard. Fortunately, with the recent advances in satisfiability solvers, this computation can be efficiently performed in polynomial time [35]. We solve the constraint c_{ip} with the solver (e.g., Choco [36]). If the solver can obtain a concrete value cv_{ip} that satisfies the constraint c_{ip}, at least one packet exists that can be processed by the application. Some messages in r_{ip} may be constructed as symbolic values. We then concrete the symbolic fields of the output message with the concrete value cv_{ip} (lines 5–8).

5.3.2. Synthetic rule generation

For each pair of constraints and output messages, $< c_{ip}, r_{ip} >$, in SE_{A_i}, one message $o^{ip}_m.e^A$ in r_{ip} includes the *Set* action (lines 10–11 in Algorithm 1). The inherent flexibility of the *Set* action may lead to indirect interference via complex synthetic rules. To address this problem, we should update the output messages r_{ip} by adding synthetic rules. Algorithm 2 illustrates the steps for updating the output messages.

Algorithm 2: Output message updating.

Input: Output messages r_{ip} of application A_i
Output: Updated messages r^{ip}_{nos} of application A_i

1 (r^{ip}_s, r^{ip}_{nos}) := *Split* (r_{ip});
2 **foreach** o^{ip}_j *in* r^{ip}_s **do**
3 **foreach** o^{ip}_k *in* r^{ip}_{nos} **do**
4 **if** $o^{ip}_k.e^M \subseteq o^{ip}_j.e^S$ **then**
5 **if** $o^{ip}_j.op \neq del$ && $o^{ip}_k.op \neq del$ && $o^{ip}_j.sw = o^{ip}_k.sw$ **then**
6 **if** $o^{ip}_j.e^A.Set(dst)$ **then**
7 e^M_{tr} := $(o^{ip}_k.e^M.src,\ o^{ip}_j.e^M.dst)$;
8 **if** $o^{ip}_j.e^A.Set(src)$ **then**
9 e^M_{tr} := $(o^{ip}_j.e^M.src,\ o^{ip}_k.e^M.dst)$;
10 $o^{ip}_{tr}.e$:=$(min(o^{ip}_j.e^P, o^{ip}_k.e^P),\ e^M_{tr},\ o^{ip}_k.e^A)$;
11 o^{ip}_{tr} := $(o^{ip}_k.op,\ o^{ip}_k.sw,\ o^{ip}_{tr}.e)$;
12 $r^{ip}_{nos} += o^{jq}_{tr}$;
13 Delete o^{ip}_j from r^{ip}_s;
14 **return** r^{ip}_{nos};

Before discussing the details of the algorithm, we introduce the e^S field of entry e. The e^S field captures the alterations to the flow attributes performed by the *Set* action. The structure of e^S is the same as that of the matching field e^M, and the value of e^S is set via the value of e^M such that the entry excludes the *Set* action. Table 1 provides an example. Considering entry e_4 (in output message o^{11}_1), the e^M field of entry e_4 refers to the match field $(src = Host_C, dst = *)$, and the e^S field of entry e_4 refers to the match field $(src = Host_E, dst = *)$.

To compute synthetic rules for the output messages r_{ip} (line 11 in Algorithm 1), we first split the output messages into two output message sets: r^{ip}_s and r^{ip}_{nos}. For each output message in r_{ip}, if it includes *Set* and *output*(*table*) actions, it will be added to the set r^{ip}_s. Otherwise, the output message will be added to the set r^{ip}_{nos} (line 1 in Algorithm 2). We then check whether different output messages could lead to a synthetic rule according to the OpenFlow specification. For output messages o^{ip}_j and o^{ip}_k, where $\forall o^{ip}_j \in r^{ip}_s$ and $\forall o^{ip}_k \in r^{ip}_{nos}$, if the $o^{ip}_k.e^M$ field matches the $o^{ip}_j.e^S$ field, message o^{ip}_k can be chained to message o^{ip}_j, resulting in the synthetic rule o^{ip}_{tr} (lines 2–5 in Algorithm 2). Note that the operation (e.g., "$\subseteq$" at line 4 in Algorithm 2) can be converted to the "$\cap$" operation. Table 1 provides an example of this case. For the output messages r_{13}, we can split the output messages into two sets: $\{o^{13}_3\}$ and $\{o^{13}_1, o^{13}_2\}$. Because the message pairs (o^{13}_3, o^{13}_1) and (o^{13}_3, o^{13}_2) satisfy the conditions of synthetic rule generation, we should then synthesize rules for the output messages (o^{13}_3, o^{13}_1) and the output messages (o^{13}_3, o^{13}_2), respectively (Fig. 4).

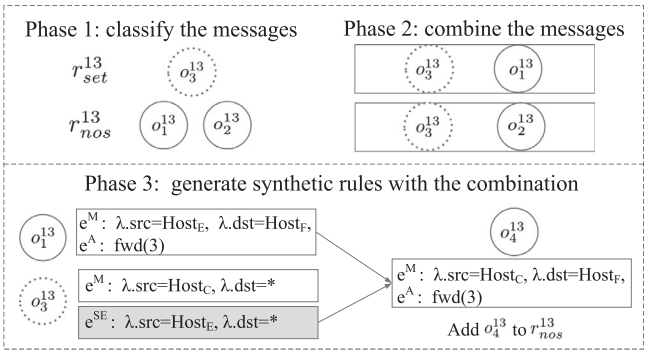


Fig. 4. Synthetic rules of r_{13}: $(add, sw_1, e(\lambda.src = Host_C, \lambda.dst = Server_D) \rightarrow fwd(3))$; $(add, sw_1, e(\lambda.src = Host_E, \lambda.dst = Host_F) \rightarrow fwd(3))$; $(add, sw_1, e(\lambda.src = Host_C, \lambda.dst = Host_F) \rightarrow fwd(3))$.

The next step is to generate synthetic rule o^{ip}_{tr} for the output messages o^{ip}_j and o^{ip}_k (lines 6–11 in Algorithm 2). For this purpose, the synthetic rule can be constructed as follows: (*i*) The priority field of entry $o^{ip}_{tr}.e$ is set with the lower one in the priority of entry $o^{ip}_j.e$ and the priority of entry $o^{ip}_k.e$. Additionally, the action of entry $o^{ip}_{tr}.e$ is set to the action of entry $o^{ip}_k.e$. (*ii*) If the *Set* action in space $o^{ip}_j.e^A$ operates on the *dst* field, then the *src* field in the match $o^{ip}_{tr}.e^M$ field is set by the *src* field in the $o^{ip}_j.e^M$ field, and the *dst* field is set by the *dst* field in the $o^{ip}_k.e^M$ field. If the *Set* action in the $o^{ip}_j.e^A$ field operates on the *src* field, then the fields in synthetic rule o^{ip}_{tr} are processed similarly. Then, we should mark output message o^{ip}_j for deletion because the logic of the output message is now incorporated in the synthetic rule. Finally, the synthetic rule should be added to the set r^{ip}_{nos}. Again considering the above example in Fig. 4, we show the generation of the synthetic rules of the output messages (o^{13}_3, o^{13}_1). The e^S field in output message o^{13}_3 matches the e^M field in output message o^{13}_1. Consequently, the e^M field in output message o^{13}_4 and the e^A field match in output message o^{13}_4. Then, the synthetic rule o^{13}_4 can be added to the set r^{13}_{nos}. By repeating the above steps until no new synthetic rules are generated, we can finally obtain three synthetic rules of r_{13} (i.e., the e^M field of o^{13}_5 refers to $(src = Host_C, dst = Host_D)$, and the e^A field of o^{13}_5 refers to $fwd(3)$).

5.3.3. Interference identification

After updating the output messages of these applications, we should check interference by considering any combinations $\{r_{1p}, r_{2q}, \ldots, r_{im}, \ldots, r_{nk}\}$ of output messages, where r_{1p} is from the output messages of the first application, r_{2q} is from the output messages of the second application, and r_{im}, r_{nk} are similarly defined. Then, for each combination, we check whether any one element (e.g., r_{ip}) conflicts with every other possible output message. However, the computation in this process is very expensive because we have to check all combinations one by one. Checking all combinations for multiple applications essentially equals to repeated checking of any two messages (e.g., $\{r_{ip}, r_{jq}\}$) from differ-

ent applications. We now prove that, for n applications, if there is no interference in any combination of two applications, then there is no interference among n applications. The converse negative proposition of above proposition is as followings: for n applications, if there is an interference among n applications, there exist an interference in some combination of two applications. The proposition is logically equivalent to its converse negative proposition. Proof sketch: This can be done by reductio ad absurdum. IH: There exists an interference among n applications which is not in the interferences in any combination of two applications. Consider an arbitrary interference among n applications. According to the definition of interference, there exists some output messages in $\{r_{1p}, r_{2q}, \ldots, r_{im}, \ldots, r_{nk}\}$ install rules with intersecting match field and conflicting action field. Let $\{r_{1p}, r_{2q}, \ldots, r_{im}\}$ denote these messages which install rules with intersecting match field and conflicting action field. Hence, any two messages in $\{r_{1p}, r_{2q}, \ldots, r_{im}\}$ install rules with intersecting match field and conflicting action field. Consider arbitrary two messages $\{r_{1p}, r_{2q}\}$ in $\{r_{1p}, r_{2q}, \ldots, r_{im}\}$. r_{1p} and r_{2q} install conflicting rules. According to the definition of interference, there is an interference among the two corresponding applications SE_1 and SE_2. Therefore, there must exist an interference in some combination of two applications. This leads to a contradiction with the IH fact. Therefore, we can conclude that if there is an interference among n applications, there exists an interference in some combination of two applications.

In Algorithm 1, first, we compute the set of all 2-combinations of the set of application *SES*, denoted as *SEC*. The number of combinations is C_n^2, where n is the number of applications in the set *SES* (line 11). Then, for each individual combination, we check whether an interference occurs between output messages r_{ip} and output messages r_{jq}, where r_{ip} and r_{jq} are from SE_{A_i} and SE_{A_j} respectively, SE_{A_i} and SE_{A_j} are a combination in *SEC* (lines 12–15). Specifically, as shown in Algorithm 3, we check whether output message o_m^{ip} and output message o_n^{jq}, $\forall o_m^{ip} \in r_{ip}$ and $\forall o_n^{jq} \in r_{jq}$, cause an interference according to the conditions in the definition of direct interference in Section 5.1 .

Algorithm 3: Interference identification.

Input: Output messages r_{ip} and r_{jq} of application A_i and application A_j
Output: Interference *IFS*

1 init $IFS := \phi$;
2 **foreach** o_m^{ip} *in* r_{ip} **do**
3 **foreach** o_n^{jq} *in* r_{jq} **do**
4 **if** $o_m^{ip}.op \neq del$ && $o_n^{jq}.op \neq del$ && $o_m^{ip}.sw = o_n^{jq}.sw$ **then**
5 **if** $o_m^{ip}.e^M \cap o_n^{jq}.e^M \neq \phi$ **then**
6 **if** $o_m^{ip}.e^A \neq o_n^{jq}.e^A$ **then**
7 **if** $o_m^{ip}.e^P \neq o_n^{ip}.e^P$ **then**
8 OverEntry (o_m^{ip}, o_n^{jq});
9 **else**
10 MissEntry (o_m^{ip}, o_n^{jq})
11 $IFS+ = \{o_m^{ip}, o_n^{jq}\}$
12 **return** *IFS*;

Note that to check whether output message o_m^{ip} and output message o_n^{jq} create an interference, we should check whether the match fields $o_m^{ip}.e^M$ and $o_n^{jq}.e^M$ have an intersection (line 5 in Algorithm 3). The result has five cases, as follows.[6]

i) $o_m^{ip}.e^M = o_n^{jq}.e^M$: The match fields of two output messages o_m^{ip} and o_n^{jq} are identical (including the wildcards), denoted as $o_m^{ip}.e^M = o_n^{jq}.e^M$. As illustrated in Fig. 5(a), the match fields in output messages in way enforcement and the firewall are the same ($src = 10.0.1.3, dst = 10.0.1.4$).

ii) $o_m^{ip}.e^M \subset o_n^{jq}.e^M$: The match fields $o_m^{ip}.e^M$ are not identical to the match fields $o_n^{jq}.e^M$, but some fields in $o_m^{ip}.e^M$ are contained in $o_n^{jq}.e^M$ due to more specified prefixes, denoted as $o_m^{ip}.e^M \subset o_n^{jq}.e^M$. As shown in Fig. 5(b), the match field ($src = 10.0.1.*$) of the way enforcement is included in the match field ($src = 10.0.*.*$) of the shortest path routing.

iii) $o_m^{ip}.e^M \supset o_n^{jq}.e^M$: As depicted in Fig. 5(b), the match field ($src = 10.0.*.*$) of the shortest path routing includes the match field ($src = 10.0.1.*$) of the way enforcement.

iv) $o_m^{ip}.e^M \cap o_n^{jq}.e^M \neq \{\phi, o_m^{ip}.e^M, o_n^{jq}.e^M\}$: As Fig. 5(c) shows, the match field ($src = 10.0.1.*$) of the way enforcement and match field ($tcp_port = 80$) of the traffic monitoring are intersecting. The result of $o_m^{ip}.e^M \cap o_n^{jq}.e^M$ is ($src = 10.0.1.* \ tcp\ port = 80$).

v) $o_m^{ip}.e^M \cap o_n^{jq}.e^M = \phi$: It means that the match field of message o_m^{ip} is not exactly intersecting with that of message o_n^{jq}. For instance, in Fig. 5(d), the match ($src = 10.0.1.*, dst = 10.0.2.*$) of the way enforcement does not intersect with the match ($src = 10.0.2.*, dst = 10.0.4.*$) of the firewall.

Except for the last case $o_m^{ip}.e^M \cap o_n^{jq}.e^M = \phi$, the match fields $o_m^{ip}.e^M$ and $o_n^{jq}.e^M$ have an intersection. In addition, if the action fields $o_m^{ip}.e^A$ and $o_n^{jq}.e^A$ are conflicting, an interference arises between output message o_m^{ip} and output message o_n^{jq} (line 6 in Algorithm 3). Then, we add concrete inputs, which respectively satisfy path conditions c_{ip} and c_{jq}, to the affected traffic (line 16 in Algorithm 1). Consider the example in Table 1. A synthetic rule o_5^{13} conflicts with message o_1^{21}, rendering the policy of the firewall ineffective. ($src = Host_C, dst = Host_D$) indicates the traffic (whose source IP is the address of $Host_C$ and whose destination IP is the address of $Host_D$) that is impacted by the interference.

Consider that an interference occurs to output message o_m^{ip} and output message o_n^{jq}. If the priority field of o_m^{ip} is higher than that

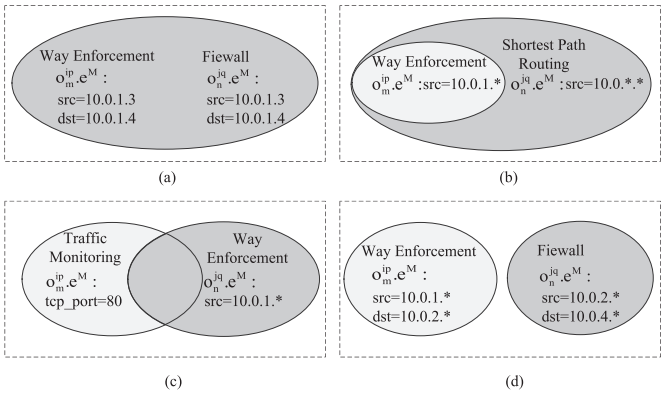


Fig. 5. Example cases to illustrate the relationship of $o_m^{ip}.e^M$ and $o_n^{jq}.e^M$. The oval represents the flow space of each match field.

[6] Fig. 5 shows examples of $o_m^{ip}.e^M \cap o_n^{jq}.e^M$ using a single field in the match space. Multiple fields in e^M are handled similarly. For example, $o_m^{ip}.e^M$ refers to $src = 10.10.1.*, dst = 10.10.2.*$, $o_n^{jq}.e^M$ refers to $dst = 10.10.2.*, dst_port = 80$, and $o_m^{ip}.e^M \cap o_n^{jq}.e^M$ is computed by intersecting each corresponding field in e^M: $src = 10.10.1.* \cap src = *.*.*.*$, $dst = 10.10.2.* \cap dst = 10.10.2.*$, $dst_port = * \cap dst_port = 80$.

of o_n^{jq} (denoted as $o_m^{ip}.e^P > o_n^{jq}.e^P$) for all packets p that can be matched by $o_m^{ip}.e^M \cap o_n^{jq}.e^M$, the switch will process packet p following the action $o_m^{ip}.e^A$, rendering o_n^{jq} ineffective (lines 7–8 in Algorithm 3). If the priority field of o_m^{ip} is the same as that of o_n^{jq} (denoted as $o_m^{ip}.e^P = o_n^{jq}.e^P$), only one entry can be installed according to the OpenFlow specification. If the entry $o_m^{ip}.e$ is missing on the switch, then o_m^{ip} is ineffective. For all packets p that can be matched by $o_m^{ip}.e^M$, the switch will process packet p without following the action $o_m^{ip}.e^A$ (lines 9–10 in Algorithm 3).

MSAID relies on the notion of 'interferences' where two rules installed by two different applications intersect but have conflicting actions. Some systems rely on priorities, assuming that the application developers are aware of other applications. In such systems, two intersecting rules with seemingly contradicting actions might actually be intentionally installed to achieve a certain behavior. To further confirm whether the interferences are unwanted behavior, users can observe the final forwarding behavior of the affected traffic (the outputs of Algorithm 1) in the network using the postcard method by NetSight [37].

Complexity: In Algorithm 1, first, constraint solving requires $\mathcal{O}(nmv^k)$ time complexity, where n is the number of SE_{A_i} in the set *SES*, m is the number of path conditions in each SE_{A_i}, and constraint solving for each path condition is polynomial time v^k (v is the number of variables, and k is a constant.). Then, synthetic rule generation only occurs when the output message r_{ip} includes the *Set* action. In the degenerate case, in each round, it requires $\mathcal{O}(u^2)$ comparisons, where u is the number of messages in each r_{ip}. The number of rounds is upper bounded by $\mathcal{O}(nm)$. The complexity of rule synthesis is thus $\mathcal{O}(nmu^2)$. In the final step, interference identification depends on the comparisons of $\{r_{ip}, r_{jq}\}$, where r_{ip} and r_{jq} can from any two applications. The total number of comparisons is $u^2n^2m^2$. Combining the above parts, the complexity yields $\mathcal{O}(nmu^2 + u^2n^2m^2)$.

Design decisions and limitations: As discussed earlier, existing verification work about SDN applications are designed to check the correctness of each individual SDN application, whereas MSAID is designed to detect the interferences of multiple SDN applications. *i*) Our approach does not produce false positives: an interference is reported if the analyzed applications instruct the switches to perform conflicting actions on the same flow. The approach might have false negatives because symbolic execution does not explore the entire state space of the program. As discussed in Section 4, even for a single SDN application, exploring the entire state space is neither feasible nor scalable. SDN application verification tools, such as NICE [38], which aim to check the correctness property of each individual SDN application, share this limitation because of the huge state space of applications. *ii*) The framework only considers interferences when flow rules are applied to the same switch. In fact, indirect interferences can occur when flow rules are applied to multiple switches. More powerful interference detection algorithms based on the output of symbolic execution are required. A strawman approach is to compute a set of nonoverlapping flow spaces based on the output OpenFlow messages and track each flow on the downstream switches, followed by checking the interference based on the rules' match, action and other fields. Obviously, this approach must be designed carefully, and in the future, we hope that we can provide a sound way to handle interference cases from multiple switches. *iii*) Several SDN applications are stateful. One symbolic message is insufficient to cover all feasible code paths. If the program's state depends on the former input symbolic message, the subsequent message can improve the coverage. Other tools based on symbolic execution, such as BUZZ [28], which check the correctness property of a stateful data plane, share a similar limitation. For future work, we can combine symbolic execution and other verification techniques (e.g., model checking) to address this problem.

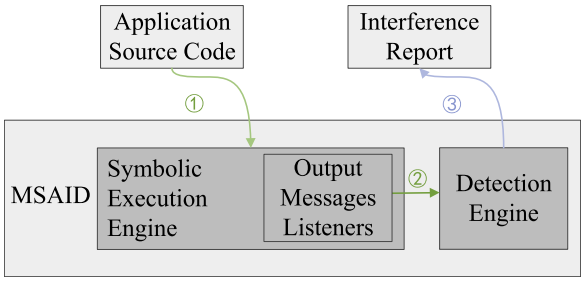


Fig. 6. MSAID prototype architecture and workflow.

6. Implementation

Fig. 6 shows the architecture and workflow of the MSAID prototype. The inputs of the prototype are the source code of the analyzed applications, and the outputs are the corresponding interference report. The prototype consists of a symbolic execution engine and detection engine. We implement the prototype in Java with approximately 4000 lines of code. The source code is available at [39].

The symbolic execution engine explores each individual application. In our prototype, we use Symbolic Java PathFinder (SPF) [34] as the symbolic execution engine to extract path conditions for each individual application. SPF is one of the most popular symbolic execution engines for Java programs, and it can be viewed as a custom-made Java virtual machine. The Model Java Interface (MJI) is one of the key components of SPF for library abstraction. Its basic functionality consists of a mechanism to intercept method invocations and delegate them by means of Java reflection calls to dedicated classes for performance concerns [34]. The analyzed SDN applications (from some controller platforms, e.g., Floodlight) running on SPF may invoke some abstractions of the classes in JDK in MJI. MJI does not consist of the abstraction of some methods in classes of JDK (e.g., class java.nio.ByteBuffer, java.lang.String.format and so on). We rewrite the corresponding codes to provide these methods based on the simplified implementation in jdk. These developments aim to make the symbolic execution engine work. These developments are a one-time offline task, and operators can use the tool directly. Then, to make the symbolic execution engine receive the input OpenFlow message and corresponding output OpenFlow messages, we leverage constraint solvers to solve the constraints in path conditions. In our prototype, constraint solving is completed by Z3bitsolver. Users can also select other constraint solvers (e.g., Choco [36] and IASolver [40]). We also develop listeners for reading, writing and manipulating the output OpenFlow messages of each individual application.

The detection engine implements the interference analysis task described in Section 5.3. The inputs of the detection engine are the input OpenFlow messages and corresponding output OpenFlow messages from the symbolic execution engine. First, the detection engine is implemented, according to Algorithm 2, to obtain the synthetic rules for these applications. By implementing an interference detection algorithm, the detection engine then crosschecks the output messages, and then it outputs an interference report if interferences exist among these multiple applications.

To start the detection process, users only need to enter some configurations, e.g., setting the symbolic variables. The configurations are the essential part of the symbolic execution engine. Specifically, these configurations enable operators to enter detailed policies, including specifying multiple methods to be

Table 2
Examples of both known and new interferences of applications coexisting.

No.	Description
I	In [7]: The internal asset protected by Secapp was unexpectedly accessed by an external entity because of the rules in non-Secapp.
II	In [43]: DNS fails to block quarantined IPs due to the prioritized QoS rule.
III	In [7]: The preferred server in the load balancer is unexpectedly quarantined by the firewall, leading to a performance degradation in the load balancer.
IV	In [5]: Rules in non-Secapp make the flow from the external host reach the internal web server, rendering the firewall ineffective.
V	New Interference in Floodlight: Clients cannot connect with the server pool in the load balancer due to unreasonable rules in the virtual network filter.
VI	New Interference in Floodlight: Traffic does not go through the firewall due to the conflicting rules in the load balancer.
VII	New Interference in Floodlight: Rules in the virtual network filter unexpectedly divide A and B into two virtual networks, rendering the firewall's policy (A to B forwarding) ineffective.
VIII	New Interference in ONOS: Traffic from Host A to Host B cannot pass SDNIP due to unreasonable rules in the virtual private LAN service (VPLS).
IX	New Interference in ONOS: Rules in ACL unexpectedly block traffic from divide A to B, which is originally permitted by Abstraction and Control of TE networks (ACTN).
X	New Interference in ONOS: Host A to Host B is unexpectedly quarantined by VPLS, rendering ACTN's policy ineffective.

executed symbolically, limiting the search depth, specifying the search strategy, selecting which constraint solver to choose, specifying min/max values for symbolic variables, setting the default values for variables you don't care about and so on. For example, to minimize the number or scope of symbolic values, users only need to enter parameters in the corresponding configuration snippets. As this configuration format follows the specification of the configuration of SPF, it is very easy to understand.

Note that MSAID uses SPF as the symbolic execution engine. Hence, it supports a representative set of SDN applications and controllers written in Java, e.g., Floodlight [17], OpenDaylight [1], and ONOS [2]. In the MSAID framework, the interference detection phase (in Section 5.3) is separated from symbolic execution (Section 5.2). The detection phase does not require access to the source code and the symbolic execution tools. For applications running on other controllers, e.g., RYU [41], users can replace SPF with other symbolic execution engines that support the *Python* language. With the advances in execution engines, any application developed in popular languages (e.g., Python and C) can be verified.

7. Evaluation

All experiments are performed on a machine with 16 GB of RAM and an Intel Core CPU running at 3.40 GHz. As introduced earlier, MSAID is an offline detection tool and it is easy to install on a computer or a server. It doesn't need to change or rewrite the kernel of the controller running the applications. The only inputs of MSAID are the source code of the tested applications. As is well known, Floodlight [17] and ONOS [2] are among the most popular controllers in the SDN community. We attempt to collect open-source applications from official controller distributions and third parties as much as possible. Eventually, we collect 59 real applications, including 38 applications on Floodlight and 21 applications on ONOS. These applications consist of applications developed by third-parties (e.g., RuleFlow, FoxNox, FlowScale, SNAC, Straros, and VTC) [7,42,43]. We run MSAID against a representative set of SDN applications from Floodlight version master [17] and ONOS version Falcon [2]. Using MSAID, we uncover several interferences proposed in recent papers, a few of which we present in Table 2.

7.1. Experiences with SDN applications

7.1.1. Known interferences of coexisting applications

For brevity, here, we introduce several examples (e.g., I–IV, where applications run on Floodlight) in Table 2.

No. I Secure application ineffectiveness due to non-secure application [7]: Secapp and non-Secapp run together. The results in MSAID show that communication from an external entity $Host_A$ to an internal asset $Host_B$ are prevented in Secapp; packets (src = *, dst = $Host_C$) are rewritten as packets (src = *, dst = $Host_B$) by non-Secapp. MSAID finds that the synthetic rules of non-Secapp and a rule in Secapp have an intersection match (src = $Host_A$, dst = $Host_B$) and conflicting actions. In validation, packets from $Host_A$ to $Host_C$ are diverted to an internal asset $Host_B$. The policy of Secapp is overwritten by non-Secapp, leading to an unsafe network.

No. II DNS ineffectiveness due to conflicting rules in QoS [43]: The QoS application and DNS protector run together on the same network. The second- and fourth-highest priorities are given to QoS, while the first- and third-highest priorities are given to the DNS protector in one snapshot. QoS's rule (the second-highest priority) that marks QoS bits of VoIP packets and forwards them to output ports is assigned a higher priority than one of the DNS rules (the third-highest priority) that blocks quarantined IPs. MSAID detects that rules in DNS and QoS have an intersecting match (src = $Host_A$, dst = $Host_B$) and conflicting actions, where $Host_B$ is one of the quarantined IPs. In validation, DNS fails to block traffic to $Host_B$ due to the QoS rules.

No. III Performance degradation in load balancer due to direct conflicting rules in firewall [7]: The load balancer and firewall run together on a controller. The path conditions and outputs in MSAID indicate that packets destined to vip_A can be handled by hosts $Host_B$, $Host_C$ and $Server_D$, the members of the VIP in the load balancer; packets from $Host_E$ to $Server_D$ are blocked by the firewall. In validation, the traffic from $Host_E$ destined to vip_A does go through the load balancer. However, the traffic is only handled by the members $Host_B$ and $Host_C$, leading to a performance degradation of the load balancer.

No. IV Firewall ineffectiveness due to conflicting rules in non-Secapp [5]: The firewall application and non-Secapp application run together on the same network. The path conditions and outputs in MSAID indicate that the non-Secapp application installs three rules. The first rule rewrites the source IP of packets whose source IP is $Host_A$ to $Host_C$. The second rule rewrites the destination IP address of packets whose destination IP is $Server_D$ to $Server_B$. The final rule simply allows packet forwarding from $Host_C$ to $Server_D$; the firewall installs a rule to block traffic from an external host $Host_A$ to an internal web server $Server_B$. MSAID detects that the synthetic rules in the non-Secapp and rules in the firewall have an intersecting match ($Host_A$ to $Server_B$) and conflicting actions. Traffic to $Server_B$ bypasses the firewall due to the rules in the non-Secapp application.

7.1.2. New interferences of coexisting applications

With these applications, we perform a large number of application coexistence scenarios (e.g., two applications, three applications or more applications in each coexistence scenario. In addition, for each coexistence scenario, we ensure that the coexistence of ap-

Table 3
Statistics of the experimental results with MSAID.

Applications	Scenarios	Interferences
59	226	51

plications is reasonable according to the specifications of applications.). For applications on Floodlight and ONOS, we can, at most, respectively obtain $C_{38}^2(703)$ and $C_{21}^2(210)$ possibilities of coexistence scenarios (two applications in each scenario). We randomly select hundreds of scenarios and apply MSAID on these real applications. Table 3 provides the experimental results. After verification in MSAID, we obtain 51 interferences of coexisting applications. Applications in the following examples are included as part of the official controller.[7]

No. V Load balancer ineffectiveness due to the unreasonable virtual network: The load balancer and virtual network filter (VNF) run together on Floodlight. The results in MSAID indicate that for packets destined to a VIP vip_A, the load balancer selects a member of the VIP and installs flow rules along the path. The VNF creates logic networks, where host $Host_B$ is in a logic network, and host $Host_C$ and host $Host_D$ are in another logic network. MSAID finds that rules in the load balancer and VNF have intersecting matches (src = $Host_B$, dst = $Host_C$) and (src = $Host_B$, dst = $Host_D$) and conflicting actions, where $Host_C$ and $Host_D$ are the members of the VIP. In validation, the conflicting rules of the VNF take precedence over that of the load balancer, resulting in an ineffective load balancer.

No. VI Firewall ineffectiveness due to load balancer: The firewall and load balancer run together on Floodlight. The path conditions and outputs in MSAID indicate that the firewall installs rules along the forwarding path to block packets from $Host_A$ destined to $Server_C$; for packets destined to a VIP vip_A, the load balancer selects a member of the VIP ($Server_B$ or $Server_C$) and installs flow rules along the path. MSAID finds that the rules in the load balancer and firewall have an intersecting match (src = $Host_A$, dst = $Host_C$) and conflicting actions. The conflicting rules of the load balancer take precedence over that of the firewall, resulting in ineffective rules in the firewall.

No. VII Firewall ineffectiveness due to the unreasonable virtual network: The VNF and firewall run together on Floodlight. The results in MSAID indicate that for packets from $Host_A$ destined to $Host_B$, the firewall installs forwarding rules along the path. VNF creates a logic network where host $Host_A$ and host $Host_B$ are divided into two logic networks. MSAID finds that rules in the firewall and VNF have an intersecting match (src = $Host_A$, dst = $Host_B$) and conflicting actions. In validation, the conflicting rules of VNF take precedence over that of the firewall. The policy in the firewall is ineffective.

No. VIII SDNIP ineffectiveness due to virtual private LAN service (VPLS): SDNIP and VPLS run together on the same controller. MSAID shows that SDNIP installs route L, which permits traffic from $Host_A$ to $Host_B$ in the external network; VPLS blocks traffic from $Host_C$ to $Host_D$ (which is the essential node of route L) by a different VLANID. MSAID detects that rules in SDNIP and VPLS have an intersecting match (src = $Host_C$, dst = $Host_D$) and conflicting actions. In validation, the conflicting rules of VPLS take precedence over the rules of SDNIP. Traffic from $Host_A$ destined to $Host_B$ does not go through route L with nodes $Host_C$ and $Host_D$.

No. IX Abstraction and control of TE networks (ACTN) ineffectiveness due to conflicting rule in ACL: ACTN runs on ONOS. The path conditions in MSAID indicate that ACTN permits $Host_A$ to communicate with $Host_B$; the rules in ACL written by another developer unexpectedly block traffic from $Host_A$ to host $Host_B$. MSAID detects that the rules in ACTN and ACL have an intersecting match ($Host_A$ *to* $Host_B$) and conflicting actions. In validation, traffic from $Host_A$ cannot reach $Host_B$. The policy of ACTN is ineffective because the rules of ACL take precedence earlier.

No. X ACTN ineffectiveness due to VPLS: ACTN and VPLS run together on the same network. The path conditions and outputs in MSAID show that the VPLS blocks traffic from $Host_A$ to $Host_B$ by different VLANID, and ACTN installs rules to permit $Host_A$ to communicate with $Host_B$. MSAID detects that rules in VPLS and VPLS have an intersecting match ($Host_A$ *to* $Host_B$) and conflicting actions. The conflicting rules of VPLS take precedence over the rules of ACTN, resulting in the policy of ACTN being ineffective.

Symbolically executing all feasible program paths does not scale to large programs. The number of feasible paths in a program grows exponentially with an increase in program size and can even be infinite in the case of programs with unbounded loop iterations [44]. It is impossible to explore all feasible code paths or cover the entire state space due to time and resource limitations. It is a common feature of all bug-finding tools in SDN application verification and testing. While, in evaluation, like other bug-finding tool about SDN applications, it is almost infeasible to define a baseline of bugs for the tested applications. Similarly, it is almost infeasible to define a baseline of interferences that were to be found for new co-existing applications. Up to now, there is no baseline about bugs or interferences of SDN application. While, in our evaluation, we provide the coverage of the codes of the tested applications. The details are shown in Section 7.2.

Discussion: We provide insights into issues that were uncovered by MSAID along two dimensions. (*i*) What is the main cause of the problem? Multiple SDN applications implement complex flow management logic rather than simply halting or forwarding a flow. Application developers may not be aware of other applications, which unintentionally leads to rules with intersecting match and conflicting action fields. Worse, most SDN controllers do not provide the key security features and policy mediation logic necessary to deploy multiple applications. (*ii*) What are the possible ways to solve this problem? It is possible that a secure control layer can help manage the interactions between the application and the data plane, or the northbound API of the controller allows the composition of applications to provide explicit guidance to resolve potential conflicts. Irrespective of which approach we attempt, with MSAID, it is easy to know how each individual application handles the input OpenFlow messages. Specifically, in the absence of explicit guidelines, identifying interferences is the only way to avoid unwanted behavior. Although attempts [14,15,22] have made some progress, mediation of applications is a long-term solution to avoid interferences. Now, MSAID allows checking of unmodified applications to identify interferences.

7.2. Scalability and efficiency

As far as we know, MSAID is the first interference detection tool before applications are pushed to systems. The proposed approach is a mix of different techniques, such as symbolic execution and use of a constraint solver, followed by identifying interferences. In this subsection, we evaluate the overall performance of the proposed framework by critical indicators (e.g., time cost and coverage of applications).

As the analyzed applications are mostly collected from official controller distributions, the code size of these applications is representative. The histogram in Fig. 7 lists the number of the analyzed applications by lines of source code (LSCs). While relatively small, these applications are not toys (almost all of the

[7] Users can download these applications from the official controller distribution for free. For brevity, we omit the introduction of the applications. Users can find all information about these applications on the official website [2,17].

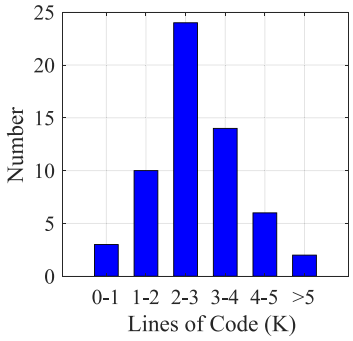


Fig. 7. Scale of the analyzed SDN applications.

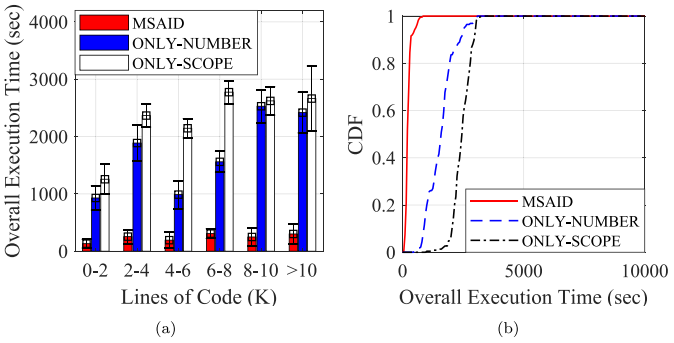


Fig. 8. Overall execution time for the analyzed SDN applications.

Table 4
Lines of code in the analyzed SDN applications.

Index	Minimum	25th Percentile
Lines	899	2152
Median	75th Percentile	Maximum
2968	3901.75	5826

open-source applications on Floodlight and ONOS from official controller distributions are collected.) – over half (39) have between 2K and 4K (59 applications in total). In addition, Table 4 shows the value of the average, minimum, maximum, and 25th and 75th percentiles among all LSCs in these applications.

We now evaluate how MSAID scales with the different applications. For all coexistence scenarios, Fig. 8(a) shows the overall execution time (including total symbolic execution time of each application (Section 5.2) and interference detection (Section 5.3) of the applications in one scenario). It is an error bar figure, where the bar shows the average value and the standard deviation. For example, for applications with 6–8K lines, the average execution time is less than 400 s, and the standard deviation is less than 100 s. Note that the x-axis refers to the total lines of code of the applications in one application coexistence scenario. Fig. 8(b) shows the cumulative distribution of the execution time. For all application coexistence scenarios, more than 92.8% of the overall execution time is less than 500 s.

As introduced earlier, the overall execution time consists of the symbolic execution time (Section 5.2) and interference detection (Section 5.3) time. Note that for all applications in one coexistence scenario, the symbolic execution of each single application is separated from others. The y-axis in Fig. 9(a) refers to the maximum of all execution times of the applications in one application coexistence scenario. Fig. 9(a) shows the time for symbolic execution, which is the most time-consuming phase of the framework. It is also an error bar figure. As shown in this figure, the time is a few minutes even for large applications with thousands of lines of code. As far as we know, its efficiency is similar to that of the

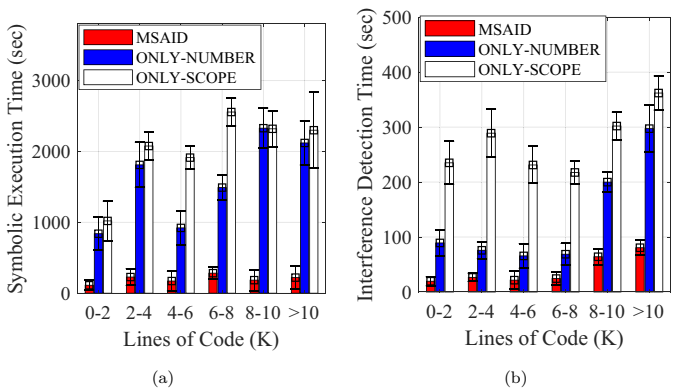


Fig. 9. Execution time in each phase for the analyzed SDN applications.

state-of-art symbolic execution technology [45]. As illustrated in Fig. 9(b), for all coexistence scenarios, the average interference detection time is less than 100 s even for applications with ten thousand lines.

Fig. 9a and Fig. 9b illustrate the contributions of the optimizations in reducing the execution time. We compare three optimization strategies, including ONLY-NUMBER (only minimizing the number of symbolic input variables), ONLY-SCOPE (only minimizing the scope of symbolic input variables), and MSAID (minimizing both the number and the scope of symbolic input variables), with the same experiment instances. Note that in the symbolic inputs of the analyzed applications, symbolic *Packet_In* message is most used. Its main part is the incoming packet. In source code, the packet is constructed as many variables (e.g, source IP, destination IP, source MAC, destination MAC, EthType, Vlan ID, TTL, variable, source port, destination port, payload and so on.). In ONLY-NUMBER, we minimize the symbolic input variables. For example, some variables (e.g., TTL and payload) that are not used to match packets against flow entries are constructed as concrete values. In ONLY-SCOPE, we minimize the scope of the symbolic input variables. For example, the variable source IP and the variable destination IP are limited from 0 (0.0.0.0) to 4,294,967,295 (255.255.255.255). The variable EthType is limited from 0 (0×0000) to 65,535 ($0\times$FFFF). The evaluation shows that, in the symbolic execution phase of the framework, for applications with 4K–6K code lines, the average time for MSAID is approximately 127 s, the time for ONLY-NUMBER is approximately 900 s, and the time for ONLY-SCOPE is approximately 1920 s. We compute ρ_1, a metric of time reduction between ONLY-NUMBER and MSAID, as $\frac{T(ONLY-NUMBER)-T(MSAID)}{T(ONLY-NUMBER)}$, and ρ_2, a metric of average time reduction between ONLY-SCOPE and MSAID, as $\frac{T(ONLY-SCOPE)-T(MSAID)}{T(ONLY-SCOPE)}$. We observe the following: in the symbolic execution phase, compared with ONLY-NUMBER and ONLY-SCOPE, MSAID significantly reduces the execution time ($\rho_1 = 79.7\%$ and $\rho_2 = 88.4\%$). In the interference detection phase, MSAID also offers a significant reduction with ($\rho_1 = 67.9\%$ and $\rho_2 = 83.9\%$).

We observe that the execution time depends on the constraints in the path conditions of the applications to some extent (Fig. 10). The execution time will increase if the constraints of the code increase. For instance, for 5K constraints, the average execution time is approximately 100 s, and for 15K constraints, the average execution time is approximately 200 s. This result is expected because symbolic execution depends on the branches of the applications, and the complexity of interference detection is determined by the path conditions. We further provide the coverage data of code of the analyzed SDN applications. Table 5 shows the statistics of the total number of code paths in the evaluated SDN application (maximum, minimum, average, median, and 75 and 25 percentiles). In addition, Fig. 11 shows the code coverage of the an-

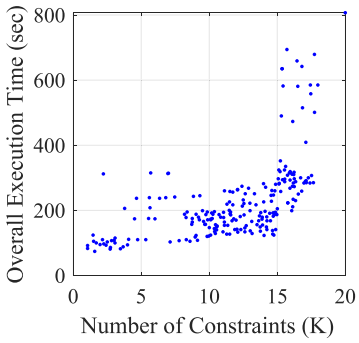


Fig. 10. Overall execution time with different constraints.

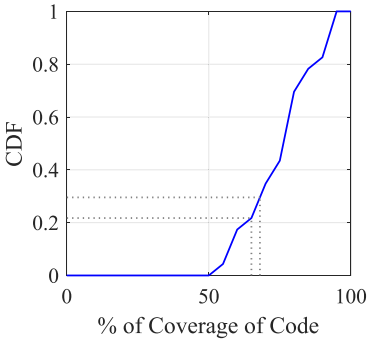


Fig. 11. Coverage of lines of code in different applications.

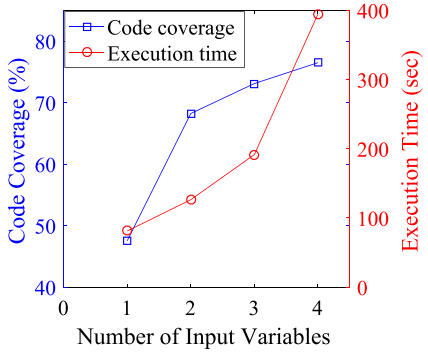


Fig. 12. Average code coverage and execution time of stateful SDN applications.

alyzed applications. For the publicly available application, according to the specification of the application on the website, we can check whether it is a stateless application. If yes, a stateless application (like stateless firewall) is executed with one symbolic input variable. For most of stateful applications, we use three symbolic input variables based on the tradeoff of coverage and execution time. In our evaluation, if the number of inputs increase, the execution time doubles but the coverage jumps less than five percentage points. And then we will not increase the inputs anymore. Fig. 12 shows the average code coverage and average symbolic execution time of stateful SDN applications with different number of input variables. The average code coverage of one symbolic input variable and that of three symbolic input variables are 47.53% and 73.09%, respectively. As shown in Fig. 11, the code coverage is more than 65% for 88.26% of the analyzed applications and more than 70% for 72.63% of the analyzed applications, which is at the right level in symbolic execution techniques [12].

These results suggest that MSAID strikes a good balance between reducing the overhead and capturing system coverage at the right level while allowing checking of unmodified SDN applications.

Table 5
Code paths of the analyzed SDN applications.

Index	Minimum	25th Percentile
Paths	256	1127.6
Median	75th Percentile	Maximum
1957.7	2506.6	8192

8. Conclusion

In this paper, we present a systematic technique that automatically uncovers the interferences of multiple SDN applications. While existing works focus on the correctness of single SDN applications, this paper takes the first step toward handling multiple applications. MSAID allows checking of unmodified applications to identify interferences before these applications are deployed in SDN systems. It explores each individual application via symbolic execution, and it analyzes the complex logic of coexisting applications effectively. We further demonstrate its effectiveness by using it to identify both new and known interferences in real SDN applications. For future work, we will extend MSAID to handle interferences caused by flow rules on multiple switches.

Acknowledgements

This work is supported by the Hainan Provincial Natural Science Foundation of China (618QN219), National Key Research and Development Program of China under Grant No. 2017YFB0803004, National Natural Science Foundation of China (61802092) and Startup Foundation of Hainan University (KYQD (ZR) 1837).

References

[1] Opendaylight., https://www.opendaylight.org/.
[2] ONOS, http://onosproject.org/.
[3] Hp's sdn, app store, http://www8.hp.com/us/en/hp-news/press-release.html?id=1798074#.W84HKdVKiM8.
[4] T. Feng, J. Bi, K. Wang, Joint allocation and scheduling of network resource for multiple control applications in SDN, in: Network Operations and Management Symposium, 2014, pp. 1–7.
[5] R. Durairajan, J. Sommers, P. Barford, Controller-agnostic SDN debugging, in: ACM International on Conference on Emerging NETWORKING Experiments and Technologies, 2014, pp. 227–234.
[6] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, G. Gu, A security enforcement kernel for openflow networks, 2012, 121–126.
[7] P. Porras, S. Cheung, M. Fong, K. Skinner, et al., Securing the Software-defined Network Control Layer, ISOC NDSS, 2015.
[8] T. Ball, B.N. Rner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, et al., Vericon: Towards Verifying Controller Programs in Software-defined Networks, in: ACM PLDI, 2014, pp. 282–293.
[9] M. Canini, D. Venzano, P. Perešíni, D. Kostić, et al., A Nice Way to Test Openflow Applications, in: USENIX NSDI, 2012, pp. 127–140.
[10] P. Kazemian, G. Varghese, N. Mckeown, Header Space Analysis: Static Checking for Networks, USENIX NSDI, 2012. 9–9.
[11] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, S. Shenker, Verifying Reachability in Networks with Mutable Datapaths, in: USENIX NSDI, USENIX Association, Boston, MA, 2017, pp. 699–718.
[12] R. Stoenescu, M. Popovici, L. Negreanu, C. Raiciu, Symnet: Scalable Symbolic Execution for Modern Networks, in: ACM SIGCOMM, 2016, pp. 314–327.
[13] G.D. Plotkin, N. Bjørner, N.P. Lopes, A. Rybalchenko, G. Varghese, Scaling Network Verification Using Symmetry and Surgery, in: ACM PLDI, 2016, pp. 69–83.
[14] A. Guha, M. Reitblatt, N. Foster, Machine-verified Network Controllers, in: ACM PLDI, 2013, pp. 483–494.
[15] N. Foster, R. Harrison, M.J. Freedman, C. Monsanto, J. Rexford, A. Story, D. Walker, Frenetic:a Network Programming Language, in: ACM ICFP, 2011, pp. 279–291.
[16] S. Son, S. Shin, V. Yegneswaran, P. Porras, Model Checking Invariant Security Properties in Openflow, in: IEEE International Conference on Communications, 2013, pp. 1974–1979.
[17] Floodlight, open sdn controller, http://projectfloodlight.org/floodlight.
[18] A. El-Hassany, J. Miserez, P. Bielik, L. Vanbever, M. Vechev, Sdnracer: Concurrency Analysis for Software-defined Networks, in: Proceedings of the 37Th ACM SIGPLAN Conference on Programming Language Design and Implementation, in: PLDI '16, ACM, New York, NY, USA, 2016, pp. 402–415.
[19] N.P. Lopes, P. Godefroid, K. Jayaraman, G. Varghese, Checking beliefs in dynamic networks, in: Usenix Conference on Networked Systems Design and Implementation, 2015, pp. 499–512.

[20] M. Kuzniar, P. Peresini, M. Canini, et al., A Soft Way for Openflow Switch Interoperability Testing, in: ACM CoNext, 2012, pp. 265–276.
[21] D.M. Volpano, et al., Towards Systematic Detection and Resolution of Network Control Conflicts, in: ACM HotSDN, 2014, pp. 67–72.
[22] P. Sun, R. Mahajan, J. Rexford, L.e.a. Yuan, A Network-state Management Service, in: ACM SIGCOMM, 2015, pp. 563–574.
[23] J. Reich, C. Monsanto, N. Foster, J. Rexford, D. Walker, Modular SDN programming with pyretic,
[24] E. Al-Shaer, H. Hamed, R. Boutaba, M. Hasan, Conflict classification and analysis of distributed firewall policies, IEEE J. Sel. Areas Commun. 23 (10) (2005) 2069–2084.
[25] J. Liu, Y. Li, H. Wang, D. Jin, L. Su, L. Zeng, T. Vasilakos, Leveraging software-defined networking for security policy enforcement, Inf. Sci. 327 (2016) 288–299.
[26] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, T. Millstein, Analyzing Protocol Implementations for Interoperability, in: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), USENIX Association, Oakland, CA, 2015, pp. 485–498.
[27] M. Dobrescu, K. Argyraki, Software Dataplane Verification, in: USENIX NSDI, 2014, pp. 101–114.
[28] S.K. Fayaz, T. Yu, et al., Buzz: Testing Context-dependent Policies in Stateful Networks, in: USENIX NSDI, 2016, pp. 275–289.
[29] W. Wu, Y. Zhang, S. Banerjee, Automatic Synthesis of Nf Models by Program Analysis, in: ACM Workshop on Hot Topics in Networks, 2016, pp. 29–35.
[30] Y. Li, Z. Wang, J. Yao, X. Yin, X. Shi, J. Wu, Msaid: automated interference detection for multiple SDN applications, in: IEEE International Conference on Network Protocols, 2017, pp. 1–2.
[31] P. Kazemian, G. Varghese, N. McKeown, Header space analysis: Static checking for networks, in: Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), USENIX, San Jose, CA, 2012, pp. 113–126.
[32] J.C. King, A new approach to program testing, ACM Sigplan Notices 10 (6) (1975) 228–233.
[33] J. Yang, C. Sar, P. Twohey, C. Cadar, D. Engler, Automatically generating malicious disks using symbolic execution, in: Proceedings of the IEEE Symposium on Security & Privacy, 2006, pp. 243–257.
[34] C.S. Pasareanu, W. Visser, D. Bushnell, et al., Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis, ASE 20 (3) (2013) 391–425.
[35] B. Selman, H. Levesque, D. Mitchell, A New Method for Solving Hard Satisfiability Problems, in: Tenth National Conference on Artificial Intelligence, 1992, pp. 440–446.
[36] Choco, http://www.choco-solver.org/.
[37] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, N. McKeown, I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks, in: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), USENIX Association, Seattle, WA, 2014, pp. 71–85.
[38] M. Canini, D. Venzano, P. Perešíni, D. Kostić, J. Rexford, A Nice Way to Test Openflow Applications, in: USENIX NSDI, USENIX, San Jose, CA, 2012, pp. 127–140.
[39] MSAID, https://github.com/lyh-jlu-thu/MSAID.
[40] Iasolver, http://www.cs.brandeis.edu/~tim/Applets/IAsolver.
[41] Ryu, http://osrg.github.io/ryu/.
[42] B. Chandrasekaran, B. Tschaen, et al., Isolating and Tolerating Sdn Application Failures with Legosdn, SOSR, 2016. 7.
[43] C. Prakash, J. Lee, Y. Turner, J.M. Kang, A. Akella, S. Banerjee, et al., Pga: Using Graphs to Express and Automatically Reconcile Network Policies, in: ACM SIGCOMM, 2015, pp. 29–42.
[44] S. Anand, P. Godefroid, N. Tillmann, Demand-driven Compositional Symbolic Execution, in: Theory and Practice of Software, International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems, 2008, pp. 367–381.
[45] S. Person, G. Yang, N. Rungta, S. Khurshid, Directed Incremental Symbolic Execution, in: ACM Sigplan Conference on Programming Language Design and Implementation, 2011, pp. 504–515.

Yahui Li received her B.S. degree from the College of Software from Jilin University, China in 2015. She is now pursuing her Ph.D. at Tsinghua University. Her research concerns network verification, network testing and formal methods.

Zhiliang Wang received his B.E., M.E. and Ph.D. degrees in computer science from Tsinghua University, China in 2001, 2003 and 2006, respectively. Currently, he is an Associate Professor at the Institute for Network Sciences and Cyberspace at Tsinghua University. His research interests include formal methods and protocol testing, next-generation Internet, and network measurement.

Jiangyuan Yao received his B.E. degree in engineering mechanics from Shandong University of Science and Technology, his M.E degree in network information and security from Beijing Jiaotong University and his Ph.D. degree in computer science from Tsinghua University. He is an Associate Professor at the College of Information Science and Technology at Hainan University. His research concerns formal methods, protocol testing and cyberspace security.

Xia Yin received her B.E., M.E. and Ph.D. degrees in computer science from Tsinghua University in 1995, 1997 and 2000, respectively. She is a Full Professor at the Department of Computer Science and Technology at Tsinghua University. Her research interests include future Internet architectures, formal methods, protocol testing and largescale Internet routing.

Xingang Shi received his B.S. degree from Tsinghua University and his Ph.D. degree from The Chinese University of Hong Kong. He is now working at the Institute for Network Sciences and Cyberspace at Tsinghua University. His research interests include network measurement and routing protocols.

Jianping Wu received his B.S., M.S., and Ph.D. from Tsinghua University. He is a Full Professor and Director of the Network Research Center and Ph.D. Supervisor at the Department of Computer Science, Tsinghua University. Since 1994, he has been in charge of the China Education and Research Network (CERNET). He is a member of the Information Advisory Committee, Office of National Information Infrastructure, Secretariat of State Council of China and is also a Vice President of the Internet Society of China (ISC). His research interests include next-generation Internet, IPv6 deployment and technologies, and Internet protocol design and engineering.

Han Zhang received the B.S. degree in Computer Science and Technology from JiLin University and Ph.D. in Tsinghua University. He is now working in the School of Cyber Science and Technology, Beihang University. His research concerns computer networks, network security and AI. He is a member of IEEE.