A Framework for Policy Inconsistency Detection in Software-Defined Networks

Seungsoo Lee, Seungwon Woo, Jinwoo Kim, Jaehyun Nam, Vinod Yegneswaran, Phillip Porras, and Seungwon Shin, *Member, IEEE*

***Abstract*— Software-Defined Networking (SDN) has aggressively grown in data center networks, telecommunication providers, and enterprises by virtue of its programmable and extensible control plane. Also, there have been many kinds of research on the security of SDN components along with the growth of SDN. Some of them have inspected network policy inconsistency problems that can severely cause network reliability and security issues in SDN. However, they do not consider whether a single network policy itself is corrupted during processing inside and between SDN components. In this paper, we thus focus on the question of how to automatically identify cases in which the SDN stack fails to prevent policy inconsistencies from arising among those components. We then present AudiSDN, an automated fuzz-testing framework designed to formulate test cases in which policy inconsistencies can arise in *OpenFlow* networks, the most prevalent SDN protocol. To prove its feasibility, we applied AudiSDN to two widely used SDN controllers, Floodlight and ONOS, and uncovered three separate CVEs (Common Vulnerabilities and Exposures) that cause the network policy inconsistencies among SDN components. Furthermore, we investigate the design flaws that cause the inconsistencies in modern SDN components, suggesting specific validations to address such a serious but understudied pragmatic concern.**

***Index Terms*— SDN, software-defined networking, network policy inconsistency.**

I. Introduction

OVER the past years, Software-Defined Networking (SDN) has been one of the most promising networking technologies, which presents a paradigm that emphasizes the decoupling of the control plane from the data plane, with a logically centralized control plane operated using (high-performance) commodity hardware, and it has been widely adopted in real-world networking environments, such as data centers [1], network infrastructure providers, and enterprise networks [2], [3]. In addition, as SDN has been getting popular, its security has been getting more attention as well. Hence, we can easily find many works of SDN security, trying to secure SDN components such as SDN application, controller, switch. For example, since FortNox [4] has first explored the feasibility of SDN-specific attack scenarios, many researchers have introduced attack scenarios and proposed a range of defensive measures [5]–[15].

More specifically, researchers have mostly focused on the security of SDN control plane as it is the most critical part of SDN, controlling all other components of SDN [7], [9]–[16]. For instance, Flow Wars [7] pointed out several attack scenarios against an SDN controller, and Rosemary [9] also investigated possible critical attacks against existing SDN controllers, proposing a secure SDN controller. Besides them, there are many other attack and defense works related to SDN controllers [9], [10], [12], [16], [17]. Indeed, the majority of SDN security projects have proposed protections from the perspective of mitigating system-level concerns, such as software bugs and application misuse, or the countermeasure of malicious traffic.

On the other hand, the security community has paid attention to the examination of the *inconsistency* of network policies (flow rules) as well, which are transformed from a component to another through the SDN stack.[1] For example, pioneering work [18]–[22] aims to address inconsistent policies that can arise from controller-switch interactions due to a variety of SDN features, such as asynchronous southbound operations [19], [23], data-plane implementation heterogeneity [20], and reactive policy updates [18], [21], [22]. While they address the inconsistencies using network-wide invariant checking, *none of them investigates whether the integrity of a single flow rule is compromised for each processing step over the SDN stacks.* We argue that such network policy inconsistency issues raise significant reliability and security concerns in SDN, and a minor syntactic mistake that arises during the multi-step process of translating SDN application inputs to instantiated flow rules can even lead to significant

Manuscript received February 28, 2021; revised June 4, 2021 and October 17, 2021; accepted January 3, 2022; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor S. Schmid. Date of publication January 14, 2022; date of current version June 16, 2022. This work was supported by Incheon National University (International Cooperative) Research Grant in 2021. *(Corresponding author: Seungsoo Lee.)*

Seungsoo Lee is with Incheon National University, Incheon 22012, South Korea (e-mail: seungsoo@inu.ac.kr).

Seungwon Woo is with the Electronics and Telecommunications Research Institute (ETRI), Daejeon 34129, South Korea (e-mail: seungww@etri.re.kr).

Jinwoo Kim and Seungwon Shin are with the Korea Advanced Institute of Science and Technology (KAIST), Daejeon 34141, South Korea (e-mail: jinwoo.kim@kaist.ac.kr; claude@kaist.ac.kr).

Jaehyun Nam is with AccuKnox, Cupertino, CA 95014 USA (e-mail: jn@accuknox.com).

Vinod Yegneswaran and Phillip Porras are with SRI International, Menlo Park, CA 94025 USA (e-mail: vinod@csl.sri.com; porras@csl.sri.com).

Digital Object Identifier 10.1109/TNET.2022.3140824

[1]The paper refers to *network policies* and *flow rules* interchangeably, as network policies represent a semantic instantiation of what is implemented by flow rules.

instability in network operations (Examples are presented in Section II.).

As the scrutiny of those problems and the approaches to mitigate them have been poorly studied so far, this paper focuses on those network policy inconsistency issues, and proposes a new framework, called AudiSDN, that automatically analyzes the processes by which administrative SDN policies are communicated through SDN components and detects inconsistencies that arise among those components. In particular, we present a specialized network policy fuzz-testing module designed to stimulate opportunities for generating potential malformed SDN policies, and introduce a detection strategy for automatically recognizing those inconsistencies in real SDN controllers. Our detection approach is informed by a state-transition diagram conception of how SDN flow rules are relayed among components and across the layered SDN stack. This model informs the fuzz testing strategy and our understanding of how to achieve a consistent test coverage that is generalizable across OpenFlow [24], the most widely used SDN implementation.

To show its feasibility, we have implemented a prototype of AudiSDN for OpenFlow and evaluated it with two well-known open-source SDN controllers: Floodlight [25] and ONOS [26]. Using AudiSDN, we have demonstrated five critical network policy inconsistency cases that have not been studied before. In the case of ONOS, we reported our findings to the vendor and obtained three new CVEs (Common Vulnerabilities and Exposures) as well. Furthermore, we have categorized the discovered inconsistencies that were found among Floodlight and ONOS, and identified the underlying root-cause design inadequacies that were responsible. While our analysis focused on two SDN implementations, we believe that AudiSDN is applicable across the breadth of OpenFlow-based SDN stacks that are implemented and widely deployed today.

The scope of our project is a narrow but important and under-studied question: *how to mitigate concerns that programmatic and interface errors among SDN components do not produce unexpected network policy inconsistencies*. With this objective, this paper makes the following contributions:

- We present a fuzz-testing methodology to stimulate policy inconsistency occurrences against SDN stacks.
- We present an SDN policy translation state model and an inconsistency detection method for automatically identifying which of our test inputs produce potential policy inconsistencies.
- We present the design and implementation of a new testing framework, called AudiSDN, which is capable of automatically generating randomized OpenFlow rules and detecting flow-rule inconsistencies.
- We evaluate AudiSDN by performing analyses on two popular and widely deployed OpenFlow controllers: Floodlight and ONOS, illustrating the effectiveness of AudiSDN in identifying a range of consistency errors and design problems in these controllers, obtaining three new CVE reports based on our analysis.

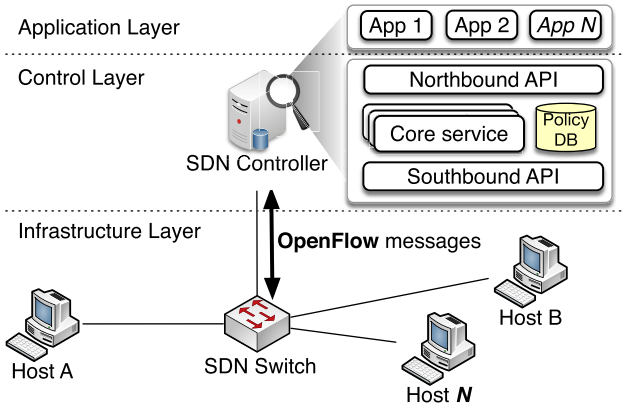


Fig. 1. SDN architecture overview.

II. Background and Motivation

A. SDN and OpenFlow

Figure 1 provides an overview of the SDN architecture that comprises three SDN components; (i) applications, (ii) controller, and (iii) switches. Unlike traditional networks, SDN decouples the control plane (composed of the application and control layers) that decides how network traffic is handled in the data plane (also known as, the infrastructure layer) which implements the packet forwarding policy given by the control layer. SDN introduces logically centralized network-policy control, enabling agile and flexible administrative control over the internal network topology. SDN controllers can provide an abstraction of the low-level management of flow rule implementation, providing a network operating system (NOS) that allows network programmers to implement intuitive network functions as an SDN application. The SDN controller manages network configurations and forwarding rules to the network forwarding devices (e.g., SDN switches) through the southbound APIs (e.g., OpenFlow [24]).

OpenFlow is currently the de-facto SDN protocol, defining the interfaces that enable the controller to interact with the forwarding devices (OpenFlow-compatible network switches). OpenFlow-enabled switches maintain a number of flow tables, containing a set of flow entries in each table. According to the OpenFlow specification, each flow entry consists of three main parts; (i) match fields (criteria) that are compared to incoming packets, (ii) a set of actions (same as instructions) that define how to process the matched incoming packets, and (iii) packet/byte counters that calculate the total number of packets/bytes. When an incoming packet arrives in a switch and there is no matching flow rule entry, the switch sends a PACKET_IN message, including the partial information of the packet to the controller. The controller decides how to handle the flow (including the given packet), builds a relevant flow rule, and then sends the rule to the switch as a FLOW_MOD message. FLOW_MOD messages include the `priority`, `match fields`, `actions` per rule, enabling the switch to bind the rule to all subsequent packets that meet the same criteria. When two or more flow rules have identical match fields, the higher priority rule takes precedence.

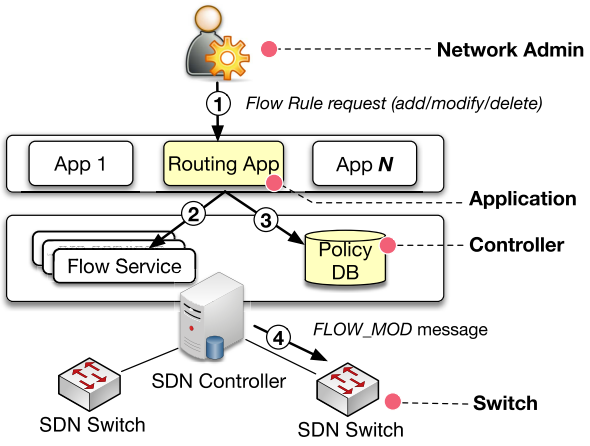


Fig. 2. The procedure of installing a flow rule to the switch and four different states of the flow rule.

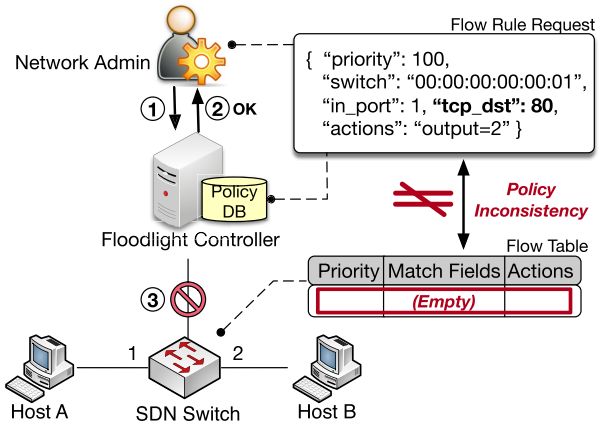


Fig. 3. This figure illustrates a Floodlight controller bug that an ill-formed flow rule can cause a CPU burst due to an infinite loop within the controller, which may result in a switch disconnect.

B. Network Policy Enforcement in SDN

In SDN, each FLOW_MOD message represents a network security policy decision, and flow rules in the message contain exactly what data will be allowed in and out of the SDN, which path the packet will traverse, and to which endpoint (host) the packet will target. A network administrator has two options in submitting network flow rules; (i) using an SDN application that computes flow rules dynamically, in response to PACKET_IN messages *reactively*, and (ii) *proactively* requesting the flow rule through the REST APIs[2] or command line interfaces provided by the SDN applications running in the controller.

The procedure for installing a flow rule into a network switch in a proactive way (the second option) is illustrated in Figure 2. (1) An administrator submits a flow rule to an SDN application through the external interfaces (e.g., the REST APIs). (2) The application builds a FLOW_MOD message based on the received request, and (3) it saves the flow rule in the internal database that the controller manages. (4) Then, the controller sends the FLOW_MOD message to the switch. (5) the switch finally installs the flow rule in the FLOW_MOD message into its flow table. Following this procedure, we observe that a flow rule is managed at four different processing points; administrator, application, controller, and switch. Of interest for this paper, this observation implies that a network policy can have different contexts (states) at each processing point. We refer to the concern that such a state difference may arise during the course of SDN operations as the **policy inconsistency problem**.

C. Motivating Example

Let us consider the example of policy inconsistency represented in Figure 3. The figure illustrates how a missed precondition made by a network administrator causes an error in the controller that can lead to a policy inconsistency between the control and data planes. In this example, we assume that there is a network topology consisting of one Floodlight SDN controller and one OpenFlow-enabled switch with two hosts. The administrator attempts to install a flow rule into the switch through REST APIs provided by the *StaticEntryPusher* application, enabling Host A to communicate with Host B over TCP port 80. The procedure of installing the flow rule into the switch is as follows. (1) The network administrator makes the request for a flow rule in a REST API form and sends it to the application. (2) Then, the application stores the flow rule in the policy database and sends the success of the reception back to the administrator. (3) However, when the application tries to build a FLOW_MOD message, the switch is disconnected from the controller due to a processing error. Finally, although there is no flow rule on the switch, the controller believes it was successfully installed, which causes the policy inconsistency between them.

The problem here is that according to the OpenFlow specification [24], one should specify the IP protocol when the TCP/UDP port numbers are used in match fields of a flow rule, but the network administrator did not follow this. Also, the application did not check for this precondition as well, and thus tried to build the FLOW_MOD message. This condition is known to cause a CPU burst due to an internal controller loop, which may result in the switch disconnecting from the controller. More seriously, the controller returns the result message (i.e., OK message), indicating that the requested flow rule was successfully installed while it was not, and the administrator would believe that the requested rule was properly installed. Moreover, the controller will preserve the flow rule in its policy database, which will now affect all SDN applications operating on the controller. The essential reason causing such inconsistencies is that the controller implementation did not properly check the protocol specification and conduct error handling.

III. System Design

This section provides an overview of the design considerations motivating AudiSDN and a detailed description of its system architecture.

[2]The REST (REpresentational State Transfer) API provides users with the interfaces to GET, PUT, POST and DELETE data through HTTP requests.

A. Design Considerations

The motivating example (from Section II) demonstrated how a simple omission can lead to *network policy (flow rule) inconsistency* between the SDN control and data planes. Manual detection of such issues is quite error-prone and complicated for network administrators. Hence, we propose an automated framework for testing and detecting policy inconsistencies in SDN stacks based on the following two design considerations.

Automatic Testing. The framework should be highly automated to minimize the administrator's intervention and time needed to generate flow rules for testing. Manually enumerating all possible network policies (flow rules), that may cause an inconsistency, is an impractical and arduous task. Thus, we adopt a black-box fuzzing technique that enables us to automatically generate various flow rule candidates and employ a flow-rule dependency tree to increase the probability of inducing inconsistencies.

Causality Detection. The framework should effectively and concretely pinpoint the root cause of the SDN flow rule inconsistency. We begin by designing a flow-rule state diagram that we use to track the state transition of flow rules from the network administrator's request to the installation in the switch. Using this state model, our approach seeks to identify the first point where a flow rule inconsistency arises during its path from formulation to deployment.

B. Problem Scope and Deployment Model

We primarily focus on the investigation of policy integrity whose corruption may affect the forwarding behavior of data-plane *permanently* [20]. Note that *temporary* inconsistency between control- and data-plane (e.g., the moment before a controller receives BARRIER_RESPONSE for flow rule installation) is out-of-scope in this paper as it is already discussed in many prior works [19], [23]. In addition, we do not focus on the rule conflict problem where multiple policies conflict with each other [27]–[29]. Regarding the deployment model, we envision that network administrators can use our framework to verify the correctness of protocol implementations (e.g., OpenFlow) in SDN controller. This task is crucial since controllers can interpret the protocol specification in a different way, producing inconsistency for the same policy between a network administrator and controllers.

C. System Architecture

This section presents the overall architecture of AudiSDN and explains its components. As illustrated in Figure 4, our framework consists of three main components; *flow rule generator, application agent, and inconsistency detector*.

Flow Rule Generator. The flow rule generator is composed of two main modules: the dependency analyzer and flow request fuzzer. The generator receives a seed file of the flow rule request, which is given by a network administrator in a `JSON` or `XML` format. The seed file is first forwarded to the dependency analyzer module. The dependency analyzer maintains an internal dependency tree (DT) of flow rules. It inspects the seed flow request based on the dependency tree, and then decides whether the seed flow request is malformed or not. It further determines which elements and values of the seed flow request will be randomized during test case generation.

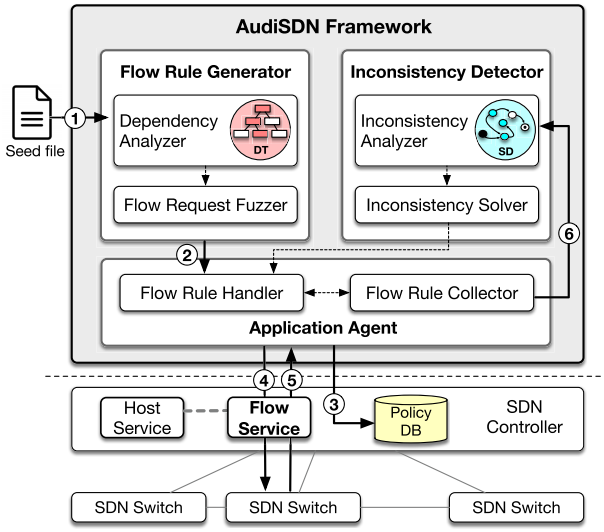


Fig. 4. Overall architecture and its workflow of AudiSDN with three key components: (i) flow rule generator, (ii) application agent, and (iii) inconsistency detector.

Next, the generator module hands decisions over to the flow request fuzzer module together with the initial flow rule request. The flow request fuzzer module generates one or numerous mutated flow rule requests according to the decisions received from the analyzer module (i.e., value and semantics randomization). After randomizing the flow rule request, the fuzzer module sends all the requests (i.e., original and mutated ones) to the flow-rule handler module in the application agent. The detailed fuzzing technique is described in Section IV.

Application Agent. The application agent runs on the target SDN controller, so it is dependent on the implementation of each controller. There are two main modules in this agent; the flow rule handler and flow rule collector. The flow rule handler module takes the role of managing flow rules in the switch through the flow services provided by the controller. Thus, the handler receives all flow rule requests (including the original seed request from the flow rule generator). It then creates flow rules based on each request and stores it in the controller database. Finally, the handler builds FLOW_MOD messages, including the created flow rule, which it sends to the switch together with the BARRIER_REQUEST message to confirm the flow rule installation by the switch.

The flow rule collector module gathers flow rule states and relevant message information, which will be used for detecting flow rule inconsistencies. Specifically, the collector tracks the process whereby one initial flow request received from the generator ultimately becomes a flow rule in the switch. If the flow rule is dropped in the middle, it records where the rule processing was stopped. Lastly, all records corresponding to each flow rule are compiled into a *rule history*, which includes all the flow rule information from the initial request by the seed file to the actually installed on the switch, and it is sent to the inconsistency detector.

Inconsistency Detector. The inconsistency detector has two main modules: the inconsistency analyzer and the inconsistency solver. The inconsistency analyzer module investigates the occurrence of flow-rule inconsistencies by exploring each rule history based on a flow rule state diagram (SD), which represents the state transition of a single flow rule from the network administrator's request to the installation in the switch.

The inconsistency solver is a runtime module. Its purpose is to deal with flow rule inconsistencies detected by the analyzer module. Since the module considers the database in the controller as the first priority by default, the module deletes the inconsistent flow rules in the switch through the application agent. It attempts to resynchronize the flow rules between the controller and the switch. However, if the flow rule cannot be installed because the rule is malformed or the switch cannot accommodate it, the solver module deletes the flow rule in the controller as well. In addition, the module leaves the logs for ex-post-facto analysis. The detection method used in this component is explained in Section V.

D. System Workflow

Our framework supports two different operation modes. First, the *testing mode* aims to detect network policy inconsistency problems in the SDN stack by using fuzzing techniques as mentioned in the previous section. Second, the *runtime mode* enables us to detect inconsistencies through real-time monitoring.

Figure 4 illustrates how an SDN policy inconsistency is detected by AudiSDN. First, in the testing mode, (1) the user inputs a seed file with flow rule requests to the flow rule generator. The generator analyzes the seed rule based on the dependency tree and then randomizes the semantics and values of the seed rule (will be explained in the next section). The generator sends the mutated flow requests with the original one to the application agent. (2) The agent processes the flow rule requests received from the generator in turn. (3) The agent stores the flow rule in the policy database and sends a FLOW_MOD message to the switch through the flow service provided by the controller. (4) The application agent fetches the installed flow rule in the switch using FLOW_STATS messages, and (5) then packs all the information about one flow rule into a rule history and sends it to the inconsistency detector. (6) The inconsistency detector finally inspects the rule history based on the flow rule state diagram, and renders a verdict on the existence of flow rule inconsistencies. On the other hand, in the runtime mode, instead of steps (1) and (2), the application agent processes the flow rule request directly from the network administrator, so the flow rule randomization is not conducted.

IV. Network Policy Fuzzing

The likelihood of network policy inconsistencies by malformed flow rules is arguably higher than from well-formed rules. Hence, we adopt a fuzzing technique to randomly generate malformed flow rules. Our technique uses flow rule dependency trees to efficiently create such malformed flow rules that allow us to inspect if the target SDNs are prone to unexpected inconsistency issues.

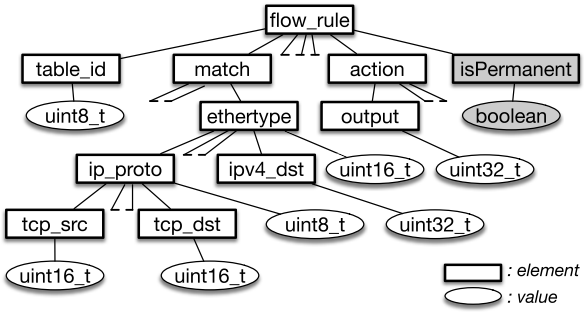


Fig. 5. The partial dependency tree example: white elements are derived from the OpenFlow specification and grey elements are extended according to ONOS and Floodlight specifications.

A. Flow Rule Dependency Tree

The first step in randomizing flow rules is that of determining the set of input parameters that must be subject to input randomization. In our framework, all elements (e.g., match fields) comprising the flow rule are potential targets. For the efficient randomization of flow-rule elements, we employ a flow-rule dependency tree (as opposed to selecting elements for randomization in an ad-hoc manner).

Figure 5 illustrates the partial dependency tree of a flow rule. The rectangular nodes and circular nodes stand for the element and its value type respectively. For example, the `table_id` element is of the unsigned byte type, so its value should be from `0` to `255`. In addition, if a child element has a parent element, there is a *dependency* between them since defining the child element semantically requires the parent element based on the OpenFlow protocol specification. Hence, if a flow rule wants to filter packets based on the TCP source port (`tcp_src`), it should specify the IP protocol (`ip_proto`) to 6 in its match fields. The dependency tree is derived from the OpenFlow specification, but it can be extended for other SDN controllers according to their implementations (as shown in the grey nodes in Figure 5).

The dependency-tree-based flow rule fuzzing framework is divided into the following two subcomponents; *(i) value randomization and (ii) semantics randomization*.

Value Randomization. The values of each element in the flow rule can be simply randomized. For example, based on the dependency tree, the `priority` element should be a numeric value and its range is from `0` to `65535`. We can randomize this value without consideration of ranges to mislead applications that try to cause the overflow or underflow for the numeric type (e.g., 65536 or -1). Also, we can manipulate the numeric type of the element (e.g., type casting) to the string type by adding double quotation marks ("`65535`"), or the Boolean type by changing the value to `false`. This method makes an application fail to cast the input value into a suitable type, generating unexpected values (e.g., `null`).

Semantics Randomization. In addition to randomizing the values, we can manipulate the semantics of flow rules by deleting or adding the inter-dependencies between the elements. For example, as shown in Figure 6, the `ipv4_dst`

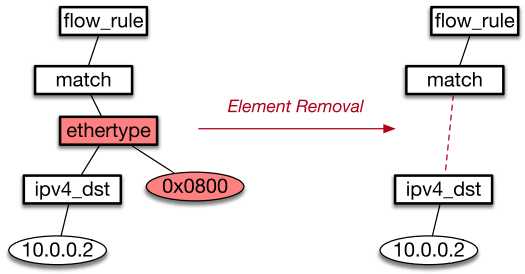


Fig. 6. Semantic randomization example in the dependency tree by removing the element.

element has a dependency on the `ethertype` element, which means that if we want to filter IP packets, we should specify the Ethernet protocol type (e.g., 0×0800) in the match fields as well. However, we can eliminate the ethertype element to see if the controller can properly handle it. Also, it is possible to append an arbitrary element to the dependency tree in order to corrupt the processing of building the flow rule.

Algorithm 1 Policy Fuzzing Algorithm

Input:
A dependency tree $T_d = (V_d, E_d)$
A seed flow rule r_{seed}
A target value type $type$
Output:
A set of mutated trees T_m

1: **procedure** DEPENDENCYTREEFUZZING(T_d, r_{seed})
2: $T_m \leftarrow \emptyset$
3: $T_{seed} \leftarrow$ BUILDSEEDTREE(r_{seed}, T_d), where $T_{seed} = (V_{seed}, E_{seed})$
4: **for** $v \in V_{seed}$ where $v.type = type$ **do**
5: $T \leftarrow T_{seed}$
6: $v_m \leftarrow$ RANDOMIZEVALUE($v, v.type$)
7: $T.modify(v, v_m)$
8: $T_m.append(T)$
▷ Randomizing Values
9: **for** $e \in E_{seed}$ **do**
10: $T \leftarrow T_{seed}$
11: $v_{parent} \leftarrow$ GETNODE(e, $Type.element$)
12: $v_{child} \leftarrow$ GETNODE(e, $Type.value$)
13: **if** $v_{parent}, v_{child} \in V_{seed}$ **then**
14: $T.removeDependency(v_{parent}, v_{child})$
15: $T_m.append(T)$
16: **else**
17: $v \leftarrow$ GETRANDOMNODE($T_{seed}, Type.element$)
18: $T.addDependency(v_{parent}, v)$
19: $T.addDependency(v, v_{child})$
20: $T_m.append(T)$
▷ Randomizing Semantics
21: **return** T_m

B. Policy Fuzzing Algorithm

To automatically randomize the values and semantics of flow rules, we present a fuzzing algorithm using tree traversal and graph matching concepts [30] as shown in Algorithm 1. This algorithm requires two inputs. The first input is the dependency tree $T_d = (V_d, E_d)$, where V_d is a set of the nodes whose types are *element* and *value*, and E_d is a set of dependency relations among V_d. The second input is a seed flow rule r_{seed} and the last is a target (mutated) value type $type$. The output is a set of the mutated flow rule trees T_m.

The algorithm initializes the output T_m as an empty set, and translates the seed flow rule into a seed tree T_{seed} (lines 2 to 3). To conduct the value randomization, the algorithm iteratively visits all the nodes v of the seed tree, and mutates each value given a type $type$. It then modifies the original node v to the mutated one v_m in the seed tree, and appends it to the output (lines 4 to 8). In the case of the semantics randomization, the algorithm mutates the seed tree by appending new nodes or removing existing nodes. To do this, it visits all edges e of the dependency tree T_{seed}, and gets an element-value node pair (lines 9 to 12). If they are included in the seed tree, the algorithm removes the dependency by removing that node pair v_{parent}, v_{child} from the seed tree, and appends it to the output set (lines 13 to 15). Otherwise, it randomly selects an element node v from the seed tree, and adds new dependency edges by adding the node between v_{parent} and v_{child} (lines 16 to 20).

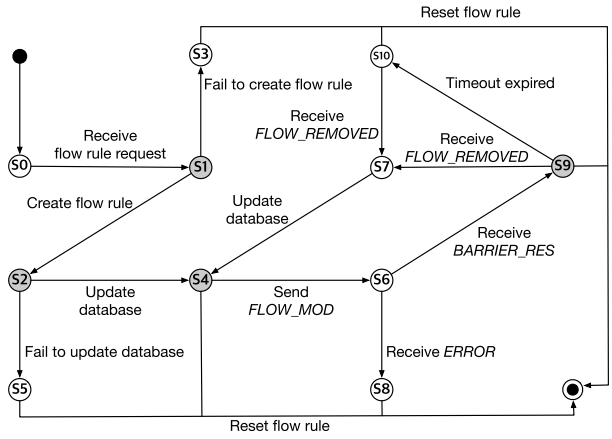


Fig. 7. The flow rule state diagram tracks the trajectory of a single flow rule from a network administrator to switch. The grey color denotes the critical states where a different rule instance is created for each SDN stack (i.e., network admin, application, controller, and switch), which are used to detect the inconsistencies of those instances.

V. DETECTION OF NETWORK POLICY INCONSISTENCY

This section describes the detection techniques to find any policy inconsistencies within the SDN components. First, we derive a flow rule state diagram that represents the state changes across SDN layers when a flow rule is enforced, and we analyze the root causes of policy inconsistencies. Lastly, we introduce a detection algorithm based on the state diagram and the analysis.

A. Flow Rule State Diagram

In the state diagram as shown in Figure 7, we define 11 different states from `S0` to `S10`, and each edge designates the specific behavior of the application agent that is responsible for handling the flow rule. For example, when the application

agent in state S1 succeeds in creating a flow rule on behalf of an administrator request, the state of the flow rule transitions to state S2; otherwise, it moves to state S3. If the application correctly receives the request in state S2, it updates the policy database by parsing the request into a data entry and transitions to state S4. The controller creates a FLOW_MOD message based on the entry, sends it to a switch, which transitions to state S6. If a switch rejects the message (S8), the controller terminates the enforcement procedure. In the case that the controller receives a BARRIER_RESPONSE message, which denotes the end of rule installation on the switch, the state transitions to S9. The rule can be removed by the controller instruction (S7) or time expiration (S10). If the controller receives a FLOW_REMOVED message, which means that the rule is removed on the switch, the controller updates the database and transitions to S4.

Critical States. We define the following four *critical states* that represent locations where a flow rule is committed to a unique SDN component in the installation procedure: S1 (admin) state corresponding to an initial request from the administrator, S2 (application) state following the store request from the application, S4 (policy database) state meaning that the flow rule has been stored in the policy database, and S9 state implying the successful flow rule installation in the switch. While flow rules in state S1 and S2 are transient, the others in state S4 and S9 are maintained in the controller and the switch respectively.

Inconsistency Relations. Based on those four critical states of the flow rule, we specify three types of *inconsistency relations* between them; the administrator (S1) and the application (S2) as IR1, the application (S2) and the controller (S4) as IR2, and the controller (S4) and the switch (S9) as IR3.

B. Root Cause Analysis

With the definition of the critical states and the inconsistency relations, we analyze the root causes that cause the flow rule inconsistencies during the flow rule installation from the network administrator to the switch. Figure 8 illustrates the steps involved in installing the flow rule from the administrator to the switch, and we mark the diverse causes that can raise such flow rule inconsistencies with red circles in this figure.

Administrator (S1) to Application (S2). A network administrator can create a flow rule using the external interfaces (e.g., REST APIs or CLIs) exposed by SDN applications. In the case of the REST APIs, the administrator composes the flow rule request in accordance with the format desired by the application, and then sends it to the application through the HTTP protocol. Then, the application creates the flow rule based on the request received from the administrator. At this time, three causes can lead to inconsistency between the administrator and the application.

(IR1-1) Poor Input Validation 1. After the application receives the flow rule request as the input through the REST APIs, it should examine if the flow rule request complies with its own implementation (and OpenFlow protocol) specification. For example, it can check if an element value (e.g., priority) included in the flow rule is within its boundary. However, if the application has a poor input validation, it could allow malformed flow rules to be created, leading to flow rule inconsistencies.

(IR1-2) Incorrect Feedback. The application should provide the administrator with a correct and detailed feedback service to inform whether the request has been accepted or not. If the feedback is simply "failed" when the requested rule is denied, the administrator cannot exactly figure out the reason for the fault. Moreover, when the application successfully creates the flow rule, it should even return the result to the user, including the created flow rule, for sanity checking.

(IR1-3) Malicious Application. There could be a malicious SDN application in the controller, which can manipulate the flow rule request received from the administrator on purpose. It can also access the database in the controller and then tamper with the installed flow rules as well. While there have been some studies that detect and prevent faulty logic in malicious SDN applications in advance [16], [31], such approaches are out of scope for our work.

Application (S2) to Controller (S4). SDN controllers maintain internal databases to manage various network assets. For example, in the case of flow rules, an SDN application stores a created flow rule in the controller's policy database before sending the flow rule to the switch with a FLOW_MOD message. Hence, other SDN applications can obtain the flow rule information from the database to serve their network functions. However, this may cause inconsistency between the application and the controller due to the following three reasons.

(IR2-1) Poor Input Validation 2. Similar to the poor input validation 1 (IR1-1), the poor input validation 2 could allow the database to store the malformed flow rule created by an SDN application. As an SDN application can reactively create a flow rule for a given PACKET_IN message from the switch, the input validation should be done before storing flow rules in the databases. However, in many cases, the flow rule created by the SDN application is directly stored in the database without thorough inspection because of the critical assumption, which most controllers have, that SDN applications are benign and comply with the OpenFlow specification.

(IR2-2) False Identification. For managing a number of flow rules in the database correctly, SDN controllers have their own identification strategy. According to the OpenFlow specification, a unique flow rule is identified by *priority* and *match fields*. With these two identification items, the controllers should track which application created flow rules and which switch they were installed to. However, if the controllers internally use a non-standard flow ID for managing the flow rules, there could be an identification gap between the database and a switch, causing abnormal flow rule operations.

(IR2-3) Improper Overwriting. The inconsistent usage of flow IDs between a controller and a switch can lead to unintended flow rule operations. For example, if a flow rule conflicts with an existing rule in the database (due to the same priority and match fields), the controller needs to properly replace the old flow rule with the new one. However, if the controller uses a weak flow ID that does not follow the standard, it can unfortunately remove other flow rules or keep

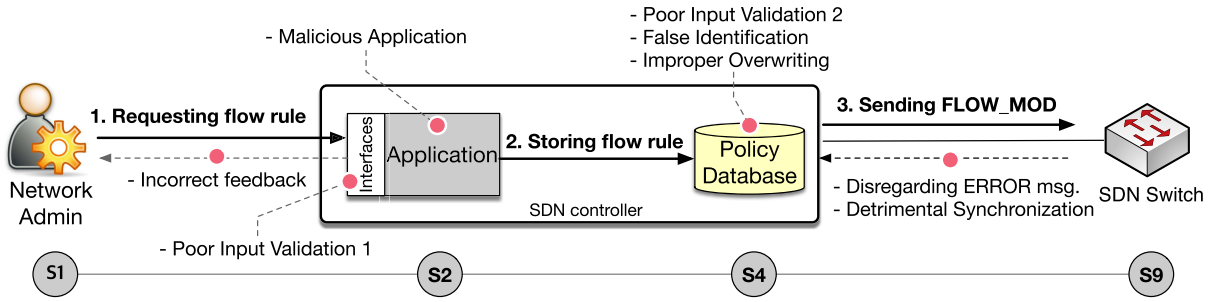


Fig. 8. The analysis of the flow rule inconsistency in the SDN stacks: four critical states of the flow rule (`S1`, `S2`, `S4` and `S9`) and the root causes of the inconsistency shown in the red points.

the conflicted flow rules together, resulting in an inconsistent rule state.

Controller (S4) to Switch (S9). Once an SDN switch receives a FLOW_MOD message from an SDN controller, it first examines if the message complies with the OpenFlow specification. Then, it checks if the flow rule can be accommodated in its flow table. If the flow rule fails to be installed in the switch during any inspection steps, the switch sends an ERROR message with a reason for the failure to the controller. If the flow rule is removed (e.g., timeout expired), the switch sends a FLOW_REMOVED message to the controller for updating the policy database. Also, the controller can periodically exchange the FLOW_STATS messages with the switch to sync up the flow rules between them. In this step, we found that the following reasons can cause the flow rule inconsistency between the controller and the switch.

(IR3-1) Disregarding ERROR Message. An SDN controller should keep track of the ERROR messages received from the switch and properly handle them. For instance, when the switch could not install the flow rule received from the controller due to some reasons (e.g., malformed rule), they send an ERROR message to the controller. However, if the controller disregards the ERROR message, the inconsistency between the controller and the switch can occur as the database in the controller maintains the malformed flow rule.

(IR3-2) Detrimental Synchronization. As another case, if the controller finds out a different flow rule between the switch and itself during exchanging FLOW_STATS messages with the switch, it first removes the flow rule in the switch. Then, the controller tries to reinstall the flow rule to the switch. However, if the flow rule is not subsequently installed in the switch, the re-installation process will be repeated again and again. If the controller does not address this situation, it can affect the performance degradation in the infrastructure layer.

C. Inconsistency Detection Algorithm

To automatically discover the flow rule inconsistencies, AudiSDN uses a graph-based heuristic. The goal is to locate any inconsistency between the abstract states of the state diagram and the concrete values in the flow rule created by SDN components. The key idea of the algorithm is to build a rule snapshot that is a currently installed policy on each SDN component when transitioning to a critical state (e.g., `S1`, `S2`, `S4`, and `S9`) and to compare the state with the last snapshot created in the previous critical state to find out if there are any inconsistent fields between them.

For those points, we model the flow rule state diagram by a Mealy machine $M = (Q, \Sigma, \delta, q_0, F)$ where:

- A finite set of states Q
- A finite set of input symbols called the alphabet Σ
- A transition function $\delta : Q \times \Sigma \rightarrow Q$
- An initial state $q_0 \in Q$
- A set of accept states $F \subseteq S$

Algorithm 2 illustrates the state traversal algorithm, which takes the state diagram (Mealy machine) M, the set of critical states $Q_{critical}$, and the input flow rule r_{input}. It is triggered when the input flow rule is delivered to the application agent from an administrator (see Figure 8). The algorithm begins by transitioning to the state S_1 and initializes the variable r', which is used for storing the last rule snapshot (lines 2 to 3). If the current state q is the one of critical states (line 5), it creates a new rule snapshot r based on the current state q and the input rule r_{input}, i.e., CREATERULESNAPSHOT. If the last rule r' snapshot exists, the algorithm compares r and r' if they are different (line 7). If any different fields are found, the algorithm computes inconsistent fields f and returns them as an output, i.e., GETINCONSISTENTFIELDS (lines 8 to 9). Otherwise, the algorithm iterates the state diagram by visiting the next state (derived based on the current state and the next controller action), until an inconsistent case is found, i.e., DONEXTACTION (lines 10 to 12). The procedure is repeated until it reaches an accept state (line 13).

VI. IMPLEMENTATION

We have implemented a prototype of AudiSDN using a combination of Java and Python to verify its feasibility and effectiveness. AudiSDN currently includes application agents for two well-known open source controllers (i.e., Floodlight [25] and ONOS [32]), enabling it to conduct the functions handling the flow rules. To randomly generate various flow requests in the flow rule generator, we implemented our fuzzing module with dependency trees and leveraged FuzzDB [33] for various malformed contents. The dependency trees are derived from the OpenFlow 1.3 specification [24] and extended according to the controller implementation, as shown in Table I. In the case of the OpenFlow, we extracted the elements of flow rule specified as `required` in it.

Algorithm 2 Inconsistency Detection Algorithm

Input:
A flow rule state diagram $M = (Q, \Sigma, \delta, q_0, F)$
A set of critical states $Q_{critical}$
An input flow rule r_{input}
Output:
An inconsistent rule field f

procedure RULEINCONSISTENCYDETECTION(M, r_{input})
  $q \leftarrow S_1$ ▷ Initialize the current state q as S_1
  $r' \leftarrow \emptyset$ ▷ The last rule snapshot
  do
    if $q \in Q_{critical}$ **then**
      $r \leftarrow$ CREATERULESNAPSHOT(q, r_{input})
      if $r' \neq \emptyset \bigwedge r \neq r'$ **then**
        $f \leftarrow$ GETINCONSISTENTFIELDS(r, r')
        return f
    $r' \leftarrow r$
    $a \leftarrow$ DONEXTACTION(q, r)
    $q \leftarrow \delta(q, a)$
  while $q \notin F$

TABLE I
THE NUMBER OF ELEMENTS AND DEPENDENCIES IN THE DEPENDENCY TREES FOR OPENFLOW, FLOODLIGHT, AND ONOS

	OpenFlow	Floodlight	ONOS
Element	66	82	106
Dependency	44	77	80

In summary, to support the design features described in Section III, we implemented two types of application agents, a flow rule generator, and an inconsistency detector, in approximately 5K lines of code.

VII. EVALUATION

We conducted a wide range of experiments and performance evaluations involving the AudiSDN framework. For the testbed, we used a Mininet [34] as the infrastructure layer in the SDN. In the case of the SDN controller, the latest versions of Floodlight [25] and ONOS [26] controllers were tested.

A. Network Policy Inconsistency Case Studies

This section demonstrates a few case studies of flow rule inconsistencies that we have identified using AudiSDN. In each case, we detail how AudiSDN detected the inconsistency and its results.

1) Flow Rule Priority OverFlow (CVE-2019-1010249): An OpenFlow switch uses the `priority` value to determine which flow rule should be applied first when matching an incoming packet stream. In this instance, we demonstrate how very large `priority` values can be translated to an abnormal one in the switch due to numeric overflows. This leads to a network policy inconsistency among the switch, the controller, and the application that had intended to install the highest priority flow rule. We identified such an inconsistency risk in the case of the ONOS controller.

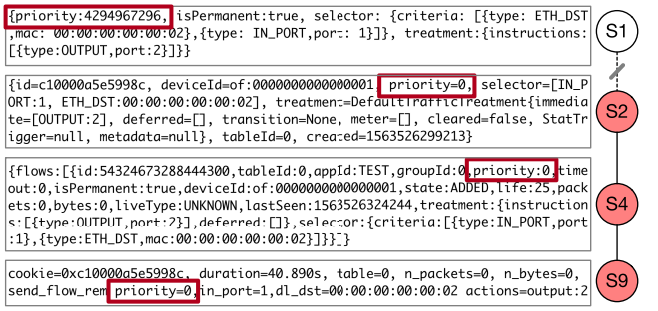


Fig. 9. Four different states of the flow rule based on the state diagram shown in Figure 7 of the ONOS controller.

Detection and Results. After the inconsistency detector fetches the rule histories from the application agent, it filters out the abnormal cases first. In this instance, one rule history was filtered out as the flow rule arrived at state `S9`. Unfortunately, its arrival produced no error message feedback, even though the priority of the flow request in state `S1` has an invalid value (i.e., out of range). However, the detector compares each flow rule in different states (i.e., S1, S2, S4, and S9) extracted from the rule history shown in Figure 9. Thus, it was able to infer that there was a flow-rule inconsistency originating between states S1 and S2. The root cause is that the invalid priority value in state S1 became 0 in state S2 (i.e., *poor input validation (IR1-1)*). Contrary to the intent of the initial flow rule request, the detector discovers that the final flow rule installed in the switch has the lowest priority value.

2) Improper Flow Rule Association and Overwriting: For identifying many flow rules correctly, various information from each flow rule could be used as a flow ID in an SDN controller such as the priority, match fields, switch ID, and flow name. However, we discovered that improper flow association can overwrite other irrelevant flow rules using AudiSDN. In this example, the test environments consist of two switches (switch A and B) and one Floodlight controller.

Detection and Results. In this case, the flow rule generator created two normal flow rule requests, but in state S9, there was only one flow rule in the switch causing the rule history to be filtered out first by the inconsistency detector. Then, an inspection of the rule history revealed the presence of flow rule inconsistencies between the application and the controller, as shown in Figure 10. The application agent creates flow rule A for switch A and installs it in the switch successfully. But, when the next flow rule (rule B) for switch B is created and stored in the controller (1), rule A is overwritten by rule B (2). Thus, the controller sends FLOW_MOD messages to delete rule A in switch A (3) and then installs rule B into switch B (4).

The reason is that the Floodlight controller distinguishes flow rules by flow's unique *name* in the database, while the switches use `match` and `priority` fields in flow table entries according to OpenFlow protocol specification (i.e., *false identification (IR2-2)*). In this case, the controller sets both flow rules' name property to `default`. Thus, although the two flow rules (A and B) have distinct switch device IDs, match fields, and actions, the inconsistency

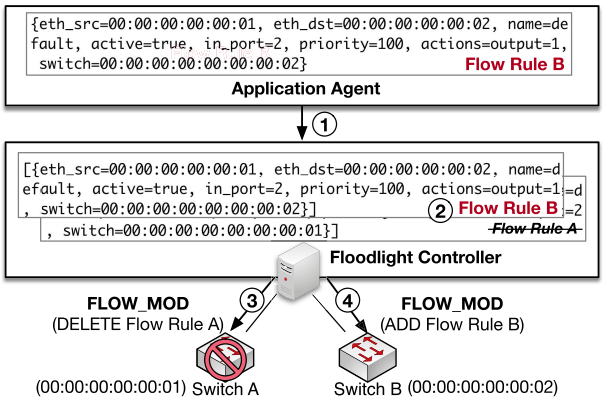


Fig. 10. Illustration of an improper flow rule association discovered in the Floodlight controller. Flow rule B overwrites flow rule A, although they correspond to different switch IDs. This example was uncovered by AudiSDN and illustrates an implementation issue (or error) that exists in the Floodlight controller.

```
511: if (oldFlowMod.getMatch().equals(newFlowMod.getMatch())
512:                 && oldFlowMod.getCookie().equals(newFlowMod.getCookie())
513:                 && oldFlowMod.getPriority() == newFlowMod.getPriority()
514:                 && dpidOldFlowMod.equalsIgnoreCase(dpid)) {   (A)
515:         log.debug("ModifyStrict SFP Flow");
516:         entriesFromStorage.get(dpid).put(entry, newFlowMod);
517:         entry2dpid.put(entry, dpid);
518:         newFlowMod = FlowModUtils.toFlowModifyStrict(newFlowMod);
519:         outQueue.add(newFlowMod);
520:         /* DELETE_STRICT and then ADD b/c the match is now different */
521: } else {
522:         log.debug("DeleteStrict and Add SFP Flow");
523:         oldFlowMod = FlowModUtils.toFlowDeleteStrict(oldFlowMod);   (B)
524:         OFFlowAdd addTmp = FlowModUtils.toFlowAdd(newFlowMod);
```

Fig. 11. A code snippet of the Floodlight controller causing the flow rule overwriting.

between the application and the controller occurs, causing network corruption at the switches.

Figure 11 provides a code snippet from the Floodlight controller that causes such improper flow rule overwriting. As shown in Figure 11 (A), when the flow name conflicts, the application checks whether the switch ID is the same along with the match fields, priority, and cookie. If all the fields are the same, it just modifies the flow rule with the new one. However, the problem arises that if at least one of them is different, it removes the old flow rule (i.e., (B) in Figure 11) and then installs a new one although the switch ID (DPID) is different. We argue that flow rules for switch ID should be maintained independently to avoid such issues.

3) Infinite Synchronization by Broken Precondition (CVE-2019-1010252): The OpenFlow protocol has evolved with support for more diverse message types and features across versions. Thus, when adding a flow rule in the switch, we should carefully consider which OpenFlow version is used. For example, the `group` action, which was added from OpenFlow version 1.1, is responsible for directing a flow to a predefined group ID used in the switch. Here, we identified an infinite flow rule synchronization problem between the ONOS controller and the switch when an incompatible action is used due to misunderstanding of protocol versions.

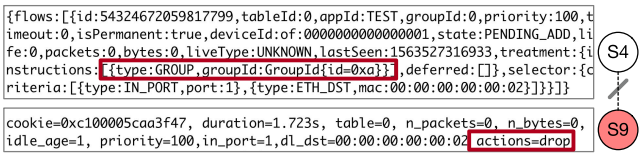


Fig. 12. Different flow rule states between the ONOS controller (S4) and the switch (S9).

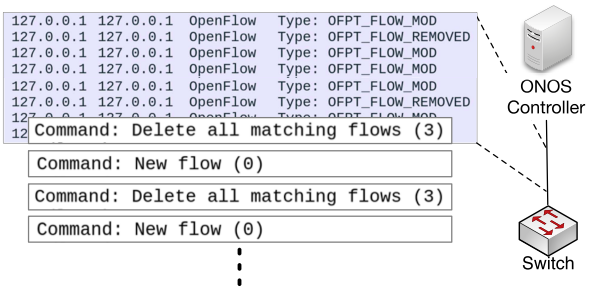


Fig. 13. Illustration of infinite synchronization in the ONOS controller.

Detection and Results. In this instance, the ONOS controller handshakes with the switch using OpenFlow version 1.0 and hence the switch cannot interpret the `group` action. However, the flow rule generator creates an abnormal flow request that has the group action using its semantics randomizer. And, the final flow rule derived from the flow request arrived in state S9, which is one of the filtering cases for the inconsistency analyzer. Finally, after inspecting the rule history received from the application agent, the inconsistency analyzer detects the inconsistency between the controller and the switch as shown in Figure 12. The flow rule built from the application agent is successfully installed in the controller. However, the switch misinterprets the FLOW_MOD message, so it installs the drop action. This example illustrates how a flow inconsistency between the controller and the switch occurs, which can affect other decisions by applications.

Furthermore, when reproducing this example, we discovered that messages are continuously exchanged between the controller and the switch [35], (as shown in Figure 13). In the case of the ONOS controller, to handle the flow rule inconsistency with the switch, all matching flow rules in the switch are removed and reinstalled in the switch through FLOW_MOD messages. This reinstallation process runs every 5 seconds by default, but administrators can modify the interval time. The problem is that the flow rule in the controller cannot be installed in the switch because the switch handshake was for OpenFlow version 1.0. This repeated reinstallation process can affect overall network performance.

4) Errors Due to Undefined Elements (CVE-2019-1010250): When a network administrator manually crafts a flow rule addition request, there is a potential for "undefined elements" due to typos in the element name. If there are such undefined typos in the flow rule, the SDN controller should not process the rule and should return an error message to the administrator. However, AudiSDN formulated a test case illustrating that a simple typo caused by the network administrator can raise inconsistency issues in the ONOS controller.

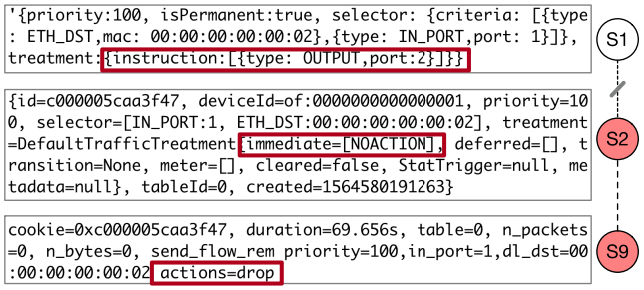


Fig. 14. Different flow rule states (S1, S2 and S9) from the rule history caused by the undefined element `instruction`.

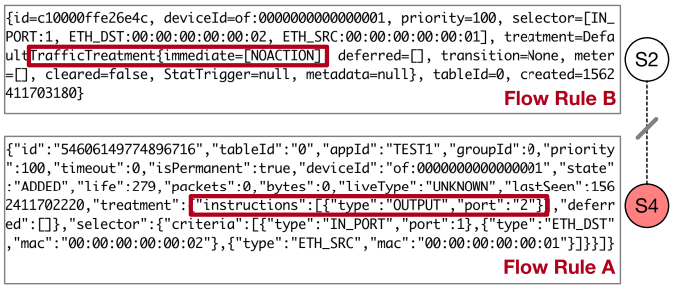


Fig. 15. Different flow rule states between the application (S2) and the ONOS controller (S4) caused by the improper flow rule association.

Detection and Results. First, the inconsistency detector discovered that a malformed flow rule in state S1 was ultimately installed in the switch at state S9 without an error state transition from the rule history. Figure 14 shows the different states of the flow rule extracted from the rule history. In state S1, the flow rule request contains the undefined element `instruction`, so it is malformed because the one defined in the application running on the ONOS controller is the `instructions` element. Since the application cannot understand the element, it builds the FLOW_MOD message by leaving the instructions as an empty field (S2 in Figure 14). However, the FLOW_MOD message that has an empty `instructions` field is recognized as a `drop` rule by the OpenFlow specification. Finally, the switch will drop all the packets received from the physical port number 1. Such undefined elements can be caused by a human error in the real world. As our results demonstrate, such problems are not limited to the ONOS controller but also applicable to the Floodlight controller.

The reason for such flow rule inconsistency between an administrator and a switch is that an action element in a request form is undefined in the REST API implemented in the StaticEntryPusher application [36]. In fact, the user should have written the request with `instructions` element (i.e., the plural form, not `instruction`) according to the specification. But, the problem is that there is no way to verify this flow request form. More seriously, the controller returned a result message indicating that the requested flow rule was successfully installed, so the administrator believes that the requested rule (i.e., forwarding action) was properly installed. This example is not only the problem of Floodlight but also applicable to ONOS.

5) Inappropriate Flow Rule Identification: Similar to the previous improper flow rule association case (Section VII-A.2), we discovered that improper flow association case in one ONOS controller by using AudiSDN. The controller should overwrite an old flow rule with a new one when they have the same identification (i.e., same priority and match fields). Here, we identified no overwriting problem between the application and the ONOS controller due to the improper flow rule association.

Detection and Results. In this example, we assume there is already flow rule A in the controller (S4 in Figure 15) that allows the packets from port 1, and it is actually installed in the switch as well. Next, the flow rule generator creates flow rule B, which has the same identification of flow rule A but has the drop action, and tries to install it through the application agent. However, in the case of the ONOS controller, there is no longer state transition from state S2 in Figure 15. Thus, the inconsistency detector was able to discover such flow rule inconsistency.

TABLE II

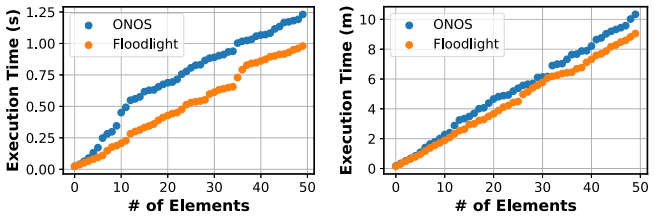
FLOW RULE INCONSISTENCY CASES IN ONOS AND FLOODLIGHT CONTROLLERS FOR MAJOR ELEMENTS: NO (NUMERIC OVERFLOW), IT (INVALID TYPE), AND BP (BROKEN PRECONDITION)

	ONOS			Floodlight		
	NO	IT	BP	NO	IT	BP
Cases	11	9	12	9	2	6

The reason for such no flow rule overwriting is that the storage manager in the ONOS controller differentiates flow rules by the flow ID only. In fact, flow rule A and B have the same priority value and match fields, so the latest flow rule B should have overwritten flow rule A. But, the controller ignored it. Consequentially, if other applications provide the network functions based on those erroneous flow rules in the controller, it can mislead the network to be unintended.

In summary, by leveraging the dependency tree, the flow rule generator in AudiSDN can create four abnormal cases of flow rule handling as follows; (i) numeric overflow, (ii) invalid type, (iii) broken precondition, and (iv) undefined element. The former two cases are instantiated by value randomization, while the others are created by semantic randomization. Besides abnormal flow requests, the flow rule generator can also design normal flow requests that can lead to the unintended network state, as stated in the aforementioned "false association" case study. We comprehensively evaluated flow rule handling by 18 major elements using AudiSDN with respect to ONOS and Floodlight controllers, and the results are summarized in Table II.

For each element, out of the 18 elements, if there exists at least one flow inconsistency between the SDN stacks, we examined it using three types of abnormal flow rule requests (numeric overflow, invalid type, broken precondition). For example, in the case of the ONOS controller, an inconsistency due to a broken precondition affects 12 out of the 18 elements, but the Floodlight controller has only 6 cases. Overall, the Floodlight controller has better validation of abnormal flow requests than the ONOS controller (as shown by the total count in Table II). Finally, in the case of undefined



(a) Semantic randomization (b) Value randomization

Fig. 16. The measured time to execute one fuzz test cycle as a function of the number of elements.

elements, we discovered that both of the controllers do not verify it because they process elements using a whitelist.

B. Performance Measurement

For finding the network policy inconsistency cases, AudiSDN serially randomizes a seed rule in the flow rule generator and builds a flow rule in the application agent. Upon completion of each fuzz test cycle, the inconsistency detector checks if there occurs an inconsistency between the policy states. Figure 16 shows the amount of time taken to complete one fuzz test cycle according to the number of selected elements from a seed rule in randomizing the values and the semantics respectively, and in this case, this seed rule consists of 50 elements in total. Overall, we observed that ONOS takes a longer time than Floodlight. This is because ONOS has more dependencies than Floodlight; thus, the tree depth significantly increases for ONOS cases.

The execution time of randomizing the values and semantics increases in the number of the selected elements. However, the value randomization takes a longer execution time than the semantic operations because it tries to explore all the possible inputs from the selected elements in the seed rule while the semantic randomization simply cuts off the random dependency between the elements. However, despite its large number of inputs in the value randomization, we found that most of the test cases can be finished in $<$11 minutes.

VIII. Limitations and Future Work

While we demonstrated that the prototype of AudiSDN can detect policy inconsistencies among SDN layers, it has several limitations that should be extended in future work:

Automatic Controller-specific Dependency Analysis. The current version of the analysis module in AudiSDN primarily aims to analyze the standard OpenFlow specification to generate a dependency tree for the policy fuzzing (Section V). However, as shown in Table I, most SDN controllers often have their own elements and dependencies beyond the OpenFlow specification to facilitate their management. Putting those elements into the dependency tree requires manual analysis of a target controller by network administrators, which is time-consuming and error-prone. In the future work, we plan to extend AudiSDN with a program analysis module that automatically finds and extracts such unique elements from controller implementations for building a rich dependency tree. Moreover, it would also be possible to provide some common functions of an application agent as libraries for its scalability.

Compatibility with OpenFlow Versions and P4 The OpenFlow specification has been constantly complicated when a major release is updated. For example, OpenFlow v1.3 has extended various functional elements and even changed existing protocol fields, compared to the OpenFlow v1.0: new protocol messages such as `multipart`, `group`, and `meters` were added and existing fields such as `actions` was changed to `instructions`. This requires AudiSDN to periodically track major changes of the specification in future.

Especially, regarding P4 [37], although the overall workflow can be similar to AudiSDN, there should be a specific module that interprets the customized protocol written in P4 format. And, instead of using OpenFlow-related APIs, the P4 Runtime APIs [38] can be employed to build a flow entry instance and retrieve the installed one on the switch.

IX. Related Work

Limited prior work on data plane security [10], [39], [40] largely relied on ad hoc empirical methods to document security flaws from diverse perspectives. Moreover, WedgeTail [41] proposed a methodology that enables detecting data-plane threats autonomously, which uses an unsupervised trajectory-based sampling mechanism. In contrast, AudiSDN systematically and synthetically tests and detects network policy inconsistency through the SDN stack.

SDN security testing such as DELTA [11] and BEADS [42] use fuzzing techniques to find bugs and vulnerabilities in SDNs by manipulating OpenFlow messages. Most similar to our work, RE-CHECKER [43] and AIM-SDN [12] randomize the REST-API inputs to check for faulty management logic in the data store of the SDN controller. However, their work primarily reports flooding attacks against SDN controllers and does not reason about the potential policy inconsistency issues of the SDN stack. Unlike these previous studies, AudiSDN provides a comprehensive strategy to assess potential network policy inconsistency problems between network administrator, application, controller (policy DB), and the data plane.

The problem of issuing consistent updates to the data plane has been well studied [44], [45]. For example, Veriflow [27], Header Space Analysis [46], and NetKat [47] provide methods for analyzing SDN flow rules to detect possible conflicts. In addition, Ravana [48] and Covisor [49] introduce methods to update SDN flow rules without conflict. VeriDP [50] proposes a way of checking the flow rule integrity between the control- and data-plane. Likewise, there have been some studies to verify the correctness of SDN applications, flow rule installation, and desired network policies with formal methods [18]–[22]. However, they do not consider the problem of software bugs leading to policy inconsistencies by the corruption of flow rule context itself between components in the SDN network.

Recently, Shukla *et al.* proposed a fuzzing methodology to verify the consistency of the control-data plane in OpenFlow-based SDNs [51] and P4-based SDNs [52]. However, while they mainly focused on the reachability between the source

and destination via the crafted traffic, AudiSDN randomizes a flow rule itself by leveraging the flow rule structure analysis and then checks whether the flow rule maintains its original meaning from the initial request to the switch installation in a top-down approach.

X. Conclusion

In this work, we have presented AudiSDN, an automated software framework for identifying deficiencies in real-world SDN implementations with respect to preventing runtime network flow policy inconsistencies. Several examples of how and why such policy inconsistencies may arise are presented, including an evaluation that utilizes our framework to uncover some implementation weaknesses in two widely used OpenFlow controllers. This paper is the first work to present a fuzz-testing methodology for automatically recognizing when and where such inter-component policy inconsistencies can arise in an SDN stack, and it highlights a fundamental security and reliability concern that has to date been largely understudied. AudiSDN offers a novel reference implementation that can be applied for testing across the breadth of OpenFlow implementations used today, and could be extended to analyze other SDN architectures beyond OpenFlow through the extension of its flow-state transition model.

References

[1] B. Heller *et al.*, "ElasticTree: Saving energy in data center networks," in *Proc. USENIX Symp. Networked Syst. Design Implement.*, 2010, pp. 249–264.

[2] C.-Y. Hong *et al.*, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM Conf. SIGCOMM*, Aug. 2013, pp. 15–26.

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

[4] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for OpenFlow networks," in *Proc. 1st Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2012, pp. 121–126.

[5] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "SDN security: A survey," in *Proc. IEEE SDN for Future Netw. Services (SDN4FNS)*, Nov. 2013, pp. 1–7.

[6] D. Kreutz, F. M. V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

[7] C. Yoon *et al.*, "Flow wars: Systemizing the attack surface and defenses in software-defined networks," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3514–3530, Dec. 2017.

[8] S. Shin and G. Gu, "Attacking software-defined networks: A first feasibility study," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2013, pp. 156–166.

[9] S. Shin *et al.*, "Rosemary: A robust, secure, and high-performance network operating system," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 78–89.

[10] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: Detecting security attacks in software-defined networks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 8–11.

[11] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras, "DELTA: A security assessment framework for software-defined networks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.

[12] V. H. Dixit, A. Doupé, Y. Shoshitaishvili, Z. Zhao, and G.-J. Ahn, "AIM-SDN: Attacking Information Mismanagement in SDN-datastores," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 664–676.

[13] J. M. Dover, "A switch table vulnerability in the open floodlight SDN controller," *Relatório Técnico*, to be published.

[14] C. Yoon *et al.*, "A security-mode for carrier-grade SDN controllers," in *Proc. 33rd Annu. Comput. Secur. Appl. Conf.*, Dec. 2017, pp. 461–473.

[15] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, "Attacking the brain: Races in the SDN control plane," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 451–468.

[16] C. Lee, C. Yoon, S. Shin, and S. K. Cha, "INDAGO: A new framework for detecting malicious SDN applications," in *Proc. IEEE 26th Int. Conf. Netw. Protocols (ICNP)*, Sep. 2018, pp. 220–230.

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

[18] J. McClurg, H. Hojjat, N. Foster, and P. Černý, "Event-driven network programming," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 369–385, 2016.

[19] V. Klimis, G. Parisis, and B. Reus, "Towards model checking real-world software-defined networks," in *Proc. Int. Conf. Comput. Aided Verification*. Springer, 2020, pp. 126–148.

[20] E. H. Campbell *et al.*, "Avenir: Managing data plane diversity with control plane synthesis," in *Proc. NSDI*, 2021, pp. 133–153.

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

[22] O. Padon, N. Immerman, A. Karbyshev, O. Lahav, M. Sagiv, and S. Shoham, "Decentralizing SDN policies," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 663–676, May 2015.

[23] R. Majumdar, S. D. Tetali, and Z. Wang, "Kuai: A model checker for software-defined networks," in *Proc. Formal Methods Comput.-Aided Design (FMCAD)*, Oct. 2014, pp. 163–170.

[24] (2012). *OpenFlow Switch Specification: Version 1.3.0*. [Online]. Available: https://www.open networking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf

[25] Floodlight. (2016). *1.2, Project Floodlight*. [Online]. Available: http://www.projectfloodlight.org/floodlight

[26] ONF. (2019). *ONOS Project, 2.1.0*. [Online]. Available: https://onosproject.org/

[27] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," in *Proc. 1st Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2012, pp. 15–27.

[28] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the software defined network control layer," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.

[29] Y. Yuan, R. Alur, and B. T. Loo, "NetEgg: Programming network policies by examples," in *Proc. 13th ACM Workshop Hot Topics Netw.*, Oct. 2014, pp. 1–7.

[30] D. B. West *et al.*, *Introduction to Graph Theory*, vol. 2. Upper Saddle River, NJ, USA: Prentice-Hall, 1996.

[31] B. Chandrasekaran, B. Tschaen, and T. Benson, "Isolating and tolerating SDN application failures with LegoSDN," in *Proc. Symp. SDN Res.*, Mar. 2016, pp. 1–12.

[32] P. Berde *et al.*, "ONOS: Towards an open, distributed SDN OS," in *Proc. Workshop Hot Topics Softw. Defined Netw.*, 2014, pp. 1–6.

[33] FuzzDB. *Dictionary of Attack Patterns and Primitives for Black-Box Application Fault Injection and Resource Discovery*. [Online]. Available: https://github.com/fuzzdb-project/fuzzdb

[34] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proc. 9th ACM SIGCOMM Workshop Hot Topics Netw. (Hotnets)*, 2010, pp. 1–6.

[35] A. Orebaugh, G. Ramirez, and J. Beale, *Wireshark & Ethereal Network Protocol Analyzer Toolkit*. Amsterdam, The Netherlands: Elsevier, 2006.

[36] *Floodlight REST API Documentation*. [Online]. Available: https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343539/Floodlight+REST+API

[37] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.

[38] P. A. W. Group. *Initial Draft Specification for P4 Runtime*. [Online]. Available: https://p4.org/p4-runtime/

[39] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 8–11.

[40] R. Skowyra *et al.*, "Effective topology tampering attacks and defenses in software-defined networks," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2018, pp. 374–385.

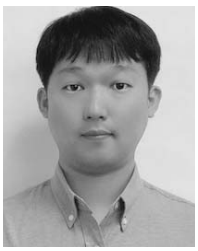
[41] A. Shaghaghi, M. A. Kaafar, and S. Jha, "WedgeTail: An intrusion prevention system for the data plane of software defined networks," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Apr. 2017, pp. 849–861.

[42] S. Jero, X. Bu, C. Nita-Rotaru, H. Okhravi, R. Skowyra, and S. Fahmy, "BEADS: Automated attack discovery in OpenFlow-based SDN systems," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*. Springer, 2017, pp. 311–333.
[43] S. Woo, S. Lee, J. Kim, and S. Shin, "RE-CHECKER: Towards secure RESTful service in software-defined networking," in *Proc. IEEE Conf. Netw. Function Virtualization Softw. Defined Netw. (NFV-SDN)*, Nov. 2018, pp. 1–5.
[44] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: Change you can believe in," in *Proc. 10th ACM Workshop Hot Topics Netw.*, 2011, pp. 1–6.
[45] T. D. Nguyen, M. Chiesa, and M. Canini, "Decentralized consistent updates in SDN," in *Proc. Symp. SDN Res.*, Apr. 2017, pp. 21–33.
[46] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. USENIX Conf. Networked Syst. Design Implement.*, 2012, pp. 113–126.
[47] C. J. Anderson *et al.*, "NetKAT: Semantic foundations for networks," in *Proc. 41st ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, Jan. 2014, pp. 113–126.
[48] N. Katta, H. Zhang, M. Freedman, and J. Rexford, "Ravana: Controller fault-tolerance in software-defined networking," in *Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res.*, Jun. 2015, pp. 1–12.
[49] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A compositional hypervisor for software-defined networks," in *Proc. USENIX Symp. Netw. Syst. Design Implement.*, 2015, pp. 87–101.
[50] P. Zhang *et al.*, "Mind the gap: Monitoring the control-data plane consistency in software defined networks," in *Proc. 12th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2016, pp. 19–33.
[51] A. Shukla, S. J. Saidi, S. Schmid, M. Canini, T. Zinner, and A. Feldmann, "Toward consistent SDNs: A case for network state fuzzing," *IEEE Trans. Netw. Service Manage.*, vol. 17, no. 2, pp. 668–681, Jun. 2020.
[52] A. Shukla, S. Fathalli, T. Zinner, A. Hecker, and S. Schmid, "P4Consist: Toward consistent p4 SDNs," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 7, pp. 1293–1307, Jul. 2020.

Seungsoo Lee received the B.S. degree in computer science from Soongsil University, South Korea, and the M.S. and Ph.D. degrees in information security from KAIST. He is an Assistant Professor with the Department of Computer Science and Engineering, Incheon National University. His research interests focus on cloud computing and network systems security. He is especially focusing his attention on software-defined networking (SDN), network function virtualization (NFV), and containers, and their security issues.

Seungwon Woo received the B.S. degree in computer science and engineering from Chungnam National University and the M.S. degree from the Graduate School of Information Security, KAIST. He is a Researcher at the Electronics and Telecommunications Research Institute (ETRI). He is interested in SDN security and blockchain area.

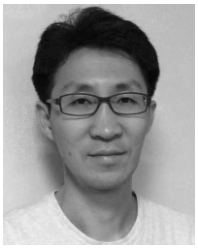
Jinwoo Kim received the B.S. degree in computer science and engineering from Chungnam National University and the M.S. degree from the Graduate School of Information Security, Korea Advanced Institute of Science and Technology (KAIST), where he is currently pursuing the Ph.D. degree with the School of Electrical Engineering. His research topics mainly focuses on software defined networking (SDN) security, designing a network security systems, and an applied networks theory.



Jaehyun Nam received the B.S. degree in computer science and engineering from Sogang University, South Korea, and the M.S. and Ph.D. degrees from the School of Computing, KAIST. He is a Technical Advisor at AccuKnox. His research interests focus on networked and distributed computing systems. He is especially interested in performance and security issues in cloud computing environments.

Vinod Yegneswaran received the A.B. degree from the University of California at Berkeley, Berkeley, CA, USA, in 2000, and the Ph.D. degree from the University of Wisconsin-Madison, Madison, WI, USA, in 2006, both in computer science. He is a Senior Computer Scientist with SRI International, Menlo Park, CA, USA, pursuing advanced research in networks and systems security. His current research interests include SDN security, malware analysis, and anti-censorship technologies. He has served on several NSF panels and program committees of security and networking conferences, including the IEEE Security and Privacy Symposium.

Phillip Porras received the M.S. degree in computer science from the University of California at Santa Barbara, Santa Barbara, CA, USA, in 1992. He is an SRI Fellow and the Program Director of the Internet Security Group, SRI's Computer Science Laboratory, Menlo Park, CA, USA. He has participated on numerous program committees and editorial boards, and participates on multiple commercial company technical advisory boards. He continues to publish and conduct technology development on numerous topics including intrusion detection and alarm correlation, privacy, malware analytics, active and software defined networks, and wireless security.



Seungwon Shin (Member, IEEE) received the B.S. and M.S. degrees from the Korea Advanced Institute of Science and Technology (KAIST), both in electrical and computer engineering, and the Ph.D. degree in computer engineering from the Electrical and Computer Engineering Department, Texas A&M University. He is an Associate Professor with the School of Electrical Engineering, KAIST. He is currently a Research Associate of Open Networking Foundation (ONF), and a member of the Security Working Group, ONF. His research interests span the areas of software defined networking (SDN) security, the IoT security, and botnet analysis/detection.