AudiSDN: Automated Detection of Network Policy Inconsistencies in Software-Defined Networks

Seungsoo Lee[†] Seungwon Woo[♯] Jinwoo Kim[†] Vinod Yegneswaran[‡] Phillip Porras[‡] Seungwon Shin[†]
[†]KAIST, Daejeon, Korea [♯]ETRI, Daejeon, Korea [‡]SRI International, CA, USA
{lss365, jinwoo.kim, claude}@kaist.ac.kr seungww@etri.re.kr {vinod, porras}@csl.sri.com

***Abstract*—At the foundation of every network security architecture lies the premise that formulated network flow policies are reliably deployed and enforced by the network infrastructure. However, software-defined networks (SDNs) add a particular challenge to satisfying this premise, as for SDNs the flow policy implementation spans multiple applications and abstraction layers across the SDN stack. In this paper, we focus on the question of how to automatically identify cases in which the SDN stack fails to prevent policy inconsistencies from arising among these components. This question is rather essential, as when such inconsistencies arise the implications to the security and reliability of the network are devastating. We 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 used today. We also present results from applying AudiSDN to two widely used SDN controllers, Floodlight and ONOS. In fact, our test results have led to the filing of 3 separate CVE reports. We believe that the approach presented in this paper is applicable to the breadth of OpenFlow platforms used today, and that its broader usage will help to address a serious but yet understudied pragmatic concern.**

***Index Terms*—SDN, Software-Defined Networking, Network Policy Inconsistency**

I. Introduction

Software-defined networking has emerged as one of the most influential technology directions in modern digital networks. SDNs are now being widely deployed in data computing centers, by network infrastructure providers, and in enterprise networks. With this greater adoption, so too has come increasing scrutiny on all aspects of SDN architectures and implementations. Since the work of FortNox has first explored the feasibility of SDN-specific attack scenarios [1], many researchers have introduced attack scenarios and proposed a range of defensive measures [2]–[12]. However, to date nearly all SDN security projects have proposed protections from the perspective of mitigating system-level concerns, such as software bugs and application misuse, or countering malicious traffic.

The security community has paid limited attention to the examination of the consistency of network policies (flow rules) as they are translated among the layers and components that compose the SDN. While there is substantial prior work to address the concerns of policy conflict prevention in SDNs [13]–[18], these studies have primarily focused on policy conflicts that arise within a single component (such as policy DB in an SDN controller). These previous projects are not directly applicable for detecting policy inconsistencies that may manifest between different components in an SDN stack (e.g., between an SDN controller and a switch).

We posit that network policy inconsistency issues raise significant reliability and security concerns for SDNs. Even a minor syntactic mistake that arises during the multi-step process of translating SDN application inputs to instantiated flow rules can lead to significant instability in network operations (examples are presented in Section II). While such issues can cause serious problems in SDN, scrutiny of these problems and approaches to mitigate these concerns have been poorly studied by SDN security researchers.

This paper proposes a new framework, AudiSDN, for automatically analyzing the processes by which administrative SDN policies are communicated through SDN components, and detecting inconsistencies that arise among those components. 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 detecting these inconsistencies in real SDN stacks. Our detection approach is informed by a state-transition diagram conception of how SDN flow rules are relayed among components and at each layer of the SDN. This model informs the fuzz testing strategy and our understanding of how to achieve a consistent test coverage that is generalizable across the most widely used SDN implementation: OpenFlow.

We implement a prototype system, AudiSDN for OpenFlow, and evaluate it with two well-known open-source SDN controllers: ONOS [19] and Floodlight [20]. Using AudiSDN, we have identified various network policy inconsistency cases, and among them, we demonstrated three case studies. In the case of ONOS, we reported our findings to the vendor as we discovered them, resulting in three new CVEs (Common Vulnerabilities and Exposures). In addition, the paper also categorizes the discovered inconsistencies that were found among ONOS and Floodlight. While our analysis focused on two OpenFlow SDN implementations, we believe that AudiSDN is applicable across the breadth of OpenFlow SDN stacks that are implemented and widely deployed today.

Our project is scoped to a narrow but important and understudied question: *how to mitigate concerns that programmatic and interface errors among SDN components do not produce*

*Seungsoo Lee and Seungwon Woo contributed equally to this work. Seungwon Shin (claude@kaist.ac.kr) is the corresponding author.

unexpected network policy inconsistencies. With this objective, this paper makes the following contributions:

- We present a methodology for fuzz testing SDN stacks for the purpose of stimulating policy inconsistencies.
- We present an SDN policy translation state model and 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. Our evaluation illustrates the effectiveness of AudiSDN in identifying a range of consistency errors and design problems in these controllers, resulting in three new CVE reports that were published based on our analysis.

II. Background and Motivation

A. SDN and OpenFlow

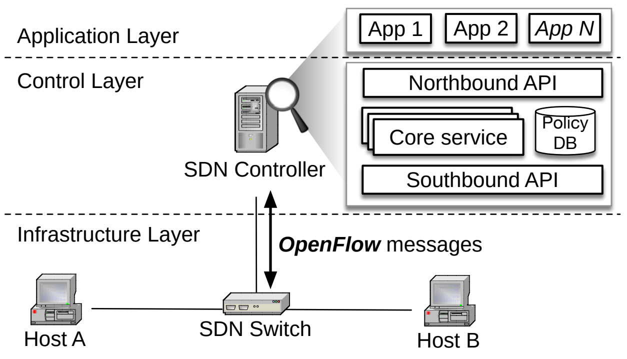


Fig. 1: SDN architecture overview.

Figure 1 provides an overview of the SDN architecture. Unlike traditional networks, SDNs decouple the control plane (i.e., application and control layers) that decides how network traffic is handled from the data plane (i.e., infrastructure layer) which implements the pack forwarding policy defined 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 also provide an abstraction of the low-level management of flow rule implementation, providing a network operating system that enables 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-enabled switches) through the southbound APIs (e.g., OpenFlow [21]).

OpenFlow is currently the de-facto SDN protocol, defining commands and behaviors that enable the controller to interact with the forwarding devices (OpenFlow-compatible network switches). Every OpenFlow-enabled switch maintains a number of flow tables, which manage a set of flow entries. 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 add up the total number of packets/bytes. When an incoming packet arrives in a switch and has 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 packet, builds a relevant flow rule, and then sends the rule to the switch through a FLOW_MOD message. FLOW_MODs 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.

B. Network Policy Enforcement in SDN

For SDNs, each FLOW_MOD message represents a network security policy decision in the truest sense. Flow rules define 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*) use an SDN application that computes flow rules dynamically, in response to PACKET_IN messages *reactively* and (*ii*) *proactively* request the flow rule through the REST APIs[1] or command line interfaces provided by the existing SDN application running inside the controller.

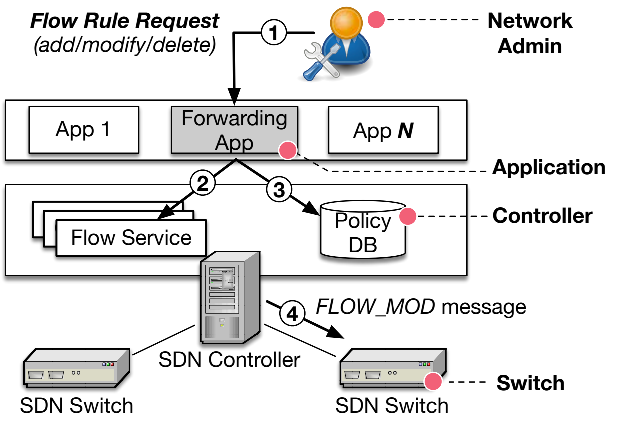


Fig. 2: The procedure of installing a flow rule to the switch and four different states of the flow rule.

The procedure for installing a flow rule into a network switch in a proactive way (the second option) is illustrated in Figure 2. First, an administrator submits a flow rule addition to an SDN application through the external interface (e.g., the REST APIs) (1). Next, the application builds a FLOW_MOD

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

message based on the received request (2) and saves the flow rule in the internal database that the controller manages (3). Then, the controller sends the FLOW_MOD message to the switch (4). Finally, the switch installs the flow rule within 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 (potentially) have different views (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

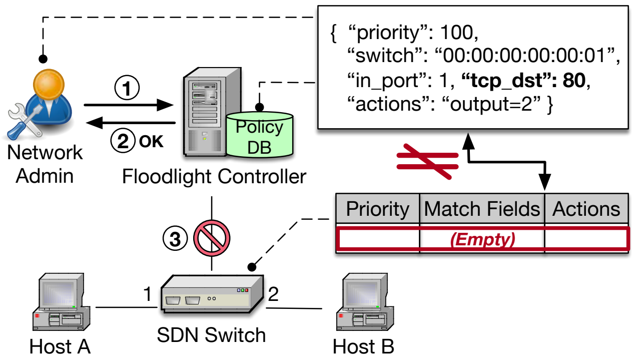


Fig. 3: This figure illustrates a Floodlight controller bug that an ill-formed flow rule can cause a CPU burst due to an infinite loop within the controller, which may result in a switch disconnect.

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 state inconsistency between the control and data plane. In this example, we assume that there is a network topology consisting of a Floodlight SDN controller, one OpenFlow-enabled switch, and 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. First, the network administrator makes the request for flow rule addition in the REST API form and sends it to the application (step 1). Then, the application stores the flow rule in the policy database and sends the success of the reception back to the administrator (step 2).

However, when the application tries to build a FLOW_MOD message, the switch is disconnected from the controller due to a processing error (step 3). Specifically, in Figure 3, a missed precondition results in a malformed FLOW_MOD request. According to the OpenFlow specification [21], one should specify the IP protocol when the TCP/UDP port numbers are used in match fields of a flow rule. The problem here is that the application did not check for this precondition, and thus tries 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, indicating that the requested flow rule was successfully installed when it was not. The outcome of this scenario is that the administrator believes that the requested rule was properly installed. The controller will also preserve the flow rule in its policy database, which will now affect all of the SDN applications operating through the controller.

III. System Design

This section provides an overview of the design considerations motivating AudiSDN and a detailed description of its system architecture. Succinctly, AudiSDN is designed to identify whether the SDN under evaluation provides adequate protections for preventing flow policy inconsistencies that may manifest between the SDN components.

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. Hence, we propose an automated framework for testing and detecting policy inconsistencies in SDN stacks based on the following two design considerations.

1. Automatic Testing. First, it should be highly automated to minimize the human 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.

2. Causality Detection. Second, it should effectively and concretely pinpoint the root cause of the SDN flow rule inconsistency. We begin by designing a flow-rule state diagram which 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. Network Policy (Flow Rule) 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.

Network Policy (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.

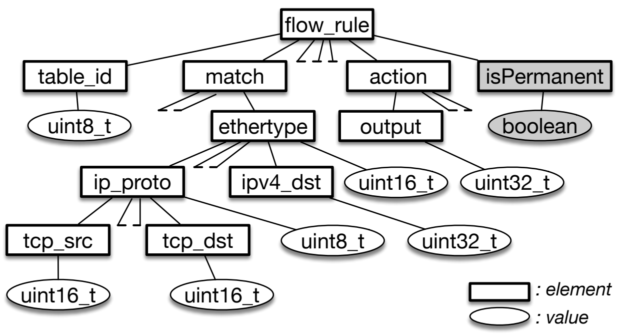


Fig. 4: The partial dependency tree example: white elements are derived from the OpenFlow specification and grey elements are extended according to ONOS and Floodlight specifications.

In our framework, all elements comprising the flow rule are potential targets. For 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 4 illustrates the partial dependency tree of a flow rule. The rectangular node 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 an element has a parent element, there is a dependency between them. 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 4).

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

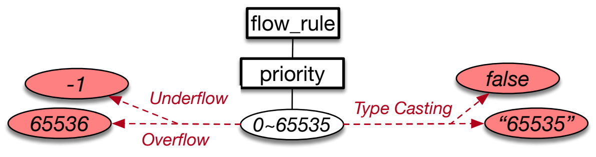


Fig. 5: Value and type randomization example in the dependency tree.

Value Randomization. The values of each element in the flow rule can be simply randomized. For example, based on the dependency tree (shown in Figure 5), the `priority` element should be 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 create the flow rule (e.g., 65536 or -1) to cause the overflow or underflow for the numeric types. Also, we can manipulate the numeric type of the element (i.e., type casting) to the string type by adding double quotation marks ("`65535`"), or the boolean type by changing the value to `false` so that it can cause the application to mistakenly set the value of the element to be empty.

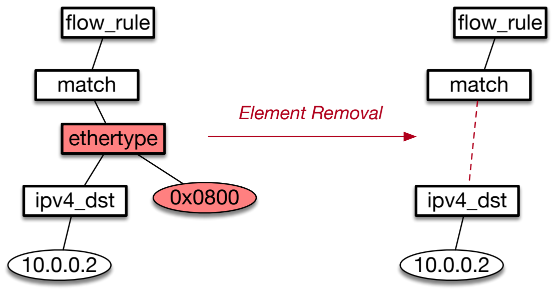


Fig. 6: Semantic randomization example in the dependency tree by removing the element.

Semantics Randomization. In addition to randomizing the values, we can manipulate the semantics of flow rules by pruning the dependency between the elements. For example, as shown in Figure 6, the `ipv4_dst` 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., 0x0800) in the match fields as well. However, we can eliminate the ethertype element to see if the controller can properly handle it. And, it is also possible to append an arbitrary element to the dependency tree as well to corrupt the processing of building the flow rule.

Algorithm 1 Fuzzing Algorithm with Dependency Tree

```
Input: a dependency tree $T_{dt} = (V_{dt}, E_{dt})$, a seed flow rule $flow_{seed}$
Output: a set of mutated flow rule trees $M$
 1: procedure DEPENDENCYTREEFUZZING($T_{dt}$, $flow_{seed}$)
 2:     $M \leftarrow$ empty set
 3:     $T_{seed} \leftarrow$ BUILDSEEDTREE($flow_{seed}, T_{dt}$)
 4:     for $v \in V_{seed}$ where $v.type = value$ do
 5:         $T \leftarrow T_{seed}$
 6:         $v_{mutated} \leftarrow$ RANDOMIZEVALUE($v.type$)
 7:         $T.modify(v, v_{mutated})$
 8:         $M.append(T)$
                                                ▷ Randomizing Values
 9:     for $e \in E_{dt}$ 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:             $M.append(T)$
16:         else
17:             $v \leftarrow$ GETRANDOMNODE($T_{seed}$, $element$)
18:             $T.addDependency(v, v_{parent})$
19:             $T.addDependency(v_{parent}, v_{child})$
20:             $M.append(T)$
                                                ▷ Randomizing Semantics
21:     return $M$
```

Algorithm. To automatically randomize the values and semantics of flow rules, we present a fuzzing algorithm using tree traversal and graph matching concepts [22] as shown in Algorithm 1. This algorithm requires two inputs. The first input is the dependency tree $T_{dt} = (V_{dt}, E_{dt})$, where V_{dt} is a set of the nodes whose types are *element* and *value*, and E_{dt} is a set of dependency relations among V_{dt}. The second input is a seed flow rule $flow_{seed}$. The output is a set of the mutated flow rules M.

The algorithm initializes the output M as an empty set, and translates the seed flow rule into a seed tree T_{seed} (line 2, 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. It then modifies the original node v to the mutated one $v_{mutated}$ 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_{dt}, 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 pair v_{parent}, v_{child} (lines 17 to 20) to the element node v in turn.

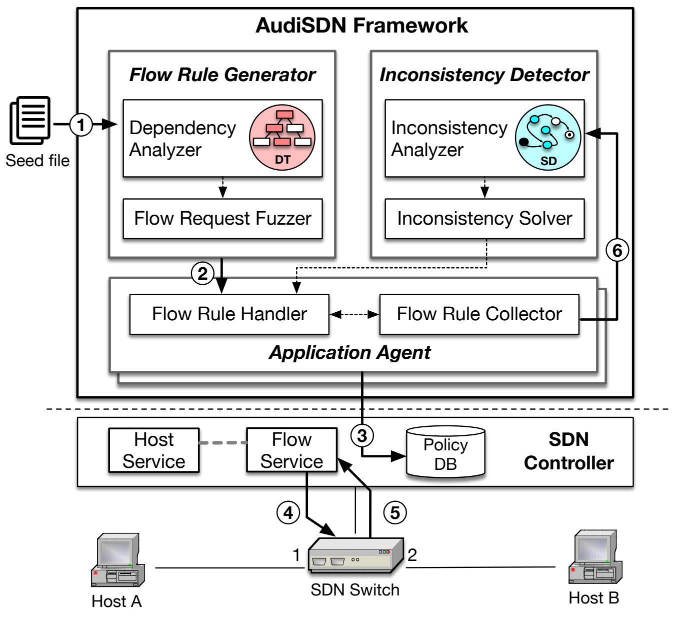


Fig. 7: Overall architecture and its workflow of AudiSDN with three key components: (i) Flow Rule Generator, (ii) Application Agent, and (iii) Inconsistency Detector.

C. System Architecture

This section presents the overall architecture of AudiSDN and explains its components. As illustrated in Figure 7, our framework consists of three main components; *flow rule generator, application agent, and inconsistency detector*. Our framework supports two different modes of operation. First, the *testing mode* aims to detect network policy inconsistency problems in the SDN stacks by using fuzzing techniques as described in the previous section. Second, the *runtime mode* enables us to detect inconsistencies through real-time monitoring.

Workflow. Figure 7 illustrates how an SDN policy inconsistency is detected by AudiSDN. (1) First, 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. 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 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) Finally, the inconsistency detector inspects the rule history based on the state diagram, and renders a verdict on the existence of flow rule inconsistencies.

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 based on a `JSON` or `XML` format from a user. The seed file is first forwarded to the dependency analyzer module. The dependency analyzer maintains the internal dependency tree of flow rules (described in the previous section). 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.

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.

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 corresponding 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 is provided to the inconsistency detector.

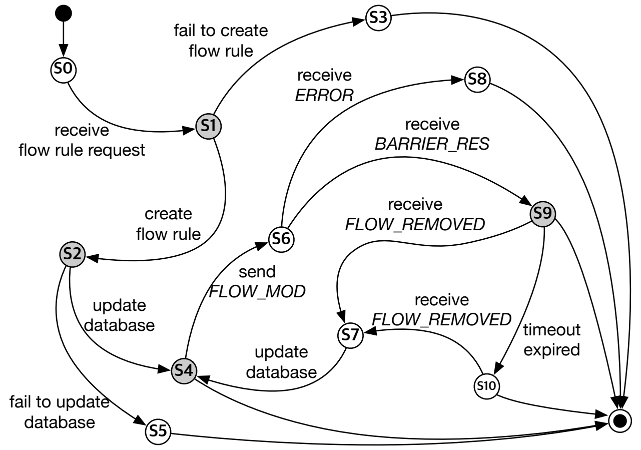


Fig. 8: Flow rule state diagram with four different states in grey color used to detect the inconsistencies.

Inconsistency Detector. The inconsistency detector component has two main modules: the inconsistency analyzer and the inconsistency solver. For detecting the flow rule inconsistencies, the inconsistency analyzer module maintains a flow-rule state diagram (as shown in Figure 8). In this diagram, 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 a user request, the state of the flow rule transitions to state `S2`, otherwise, it moves to state `S3`.

The inconsistency analyzer module investigates the occurrence of flow-rule inconsistencies by exploring each rule history based on the state diagram. First, the analyzer module filters out three abnormal cases by checking whether the flow-rule request in state `S1` is malformed. The first case arises when the flow request is malformed, but the flow rule state successfully arrived at the state `S9`. Second, we filter cases where the flow request is protocol-compliant but the final states were in the failed set (`S3`, `S5` or `S8`). The last case arises when the number of the flow rules in the switch is not equal to the requests. Then, the analyzer module extracts the flow rules in the four states `S1`, `S2`, `S4` and `S9`[2] from the filtered cases and interrogates these states for any inconsistency between them.

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 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.

[2]State S1, S2, S4 and S9 correspond to network administrator, application, database, and switch respectively in Figure 2.

IV. Implementation

We have implemented an instance 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., ONOS [23] and Floodlight [20]), 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 [24] for various malformed contents. The dependency trees are derived from the OpenFlow 1.3 specification [21] 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. 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 5,000 lines of code.

TABLE I: The number of elements and dependencies in each dependency tree.

	OpenFlow	**ONOS**	**Floodlight**
Element	66	106	82
Dependency	44	80	77

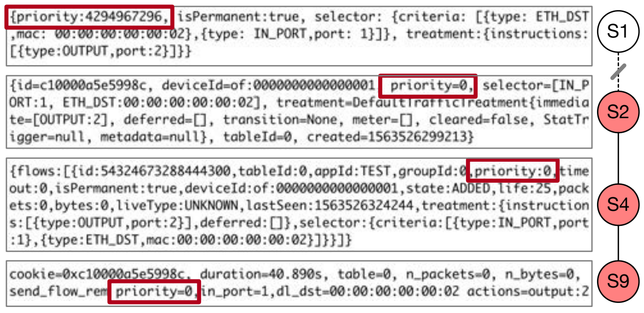
V. Network Policy Inconsistency Case Studies

This section discusses 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. For the testbed, we used a Mininet [25] as the infrastructure layer in the SDN. In the case of the SDN controller, the latest versions of Floodlight [20] and ONOS [19] controllers were tested.

A. Flow Rule Priority OverFlow (CVE-2019-1010249)

A priority value, which is one of elements of a flow rule, is used by the switch to determine which flow rule that matches an incoming packet stream will be applied first. In this instance, we demonstrate how very large flow rule priority values can switch to the lowest one in the switch due to numeric overflows. This leads to a network policy inconsistency between 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.

Detection and Results. After the inconsistency detector fetches the rule histories from the application agent, as illustrated in Figure 7, it filters out the abnormal cases first. In this instance, one rule history was filtered out as the flow rule



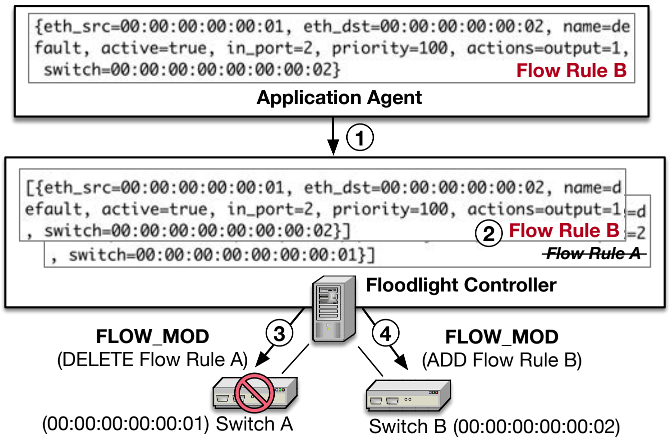
```
{priority:4294967296, isPermanent:true, selector: {criteria: [{type: ETH_DST
,mac: 00:00:00:00:00:02},{type: IN_PORT,port: 1}]}, treatment:{instructions:
[{type:OUTPUT,port:2}]}}

{id=c10000a5e5998c, deviceId=of:0000000000000001, priority=0, selector=[IN_P
ORT:1, ETH_DST:00:00:00:00:00:02], treatment=DefaultTrafficTreatment{immedia
te=[OUTPUT:2], deferred=[], transition=None, meter=[], cleared=false, StatTr
igger=null, metadata=null}, tableId=0, created=1563526299213}

{flows:[{id:54324673288444300,tableId:0,appId:TEST,groupId:0,priority:0,time
out:0,isPermanent:true,deviceId:of:0000000000000001,state:ADDED,life:25,pack
ets:0,bytes:0,liveType:UNKNOWN,lastSeen:1563526324244,treatment:{instruction
s:[{type:OUTPUT,port:2}],deferred:[]},selector:{criteria:[{type:IN_PORT,port
:1},{type:ETH_DST,mac:00:00:00:00:00:02}]}}]}

cookie=0xc10000a5e5998c, duration=40.890s, table=0, n_packets=0, n_bytes=0,
send_flow_rem priority=0,in_port=1,dl_dst=00:00:00:00:00:02 actions=output:2
```

Fig. 9: Four different states of the flow rule based on the state diagram shown in Figure 8 of the ONOS controller.

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 (S1, S2, S4, and S9 as shown in Figure 8) 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 (i.e., the invalid priority value in state S1 became 0 in state S2). 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.

B. Improper Flow Rule Association and Overwriting

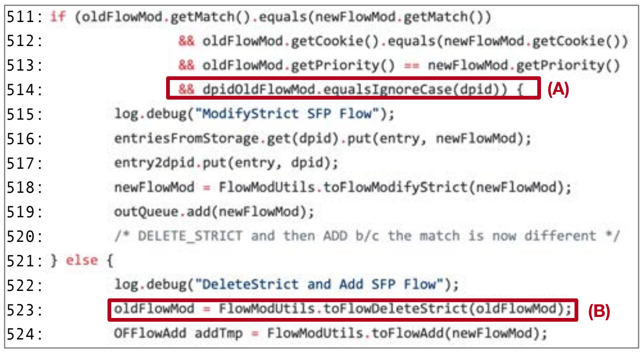
For identifying many flow rules correctly, various information from each flow rule could be maintained in the SDN controller such as the priority, match fields, switch ID, and flow ID. However, we discovered that improper flow association can overwrite other irrelevant flow rules using AudiSDN. In this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



```
{eth_src=00:00:00:00:00:01, eth_dst=00:00:00:00:00:02, name=de
fault, active=true, in_port=2, priority=100, actions=output=1,
 switch=00:00:00:00:00:00:00:02}

[{eth_src=00:00:00:00:00:01, eth_dst=00:00:00:00:00:02, name=d
efault, active=true, in_port=2, priority=100, actions=output=1
, switch=00:00:00:00:00:00:00:02}]

, switch=00:00:00:00:00:00:00:01}]
```

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.

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 the rule B (2). Thus, the controller sends FLOW_MOD messages to delete rule A in the switch A (3) and then installs rule B into switch B (4). The reason is that the controller considers each flow name as the identification method, regardless of the switch ID and the flow rule information. In this case, both flow rules have their name property set to default. Thus, although the two flow rules (A and B) have distinct switch device IDs, match fields, and actions, the inconsistency between the application and the controller occurs, causing network corruption at the switches.



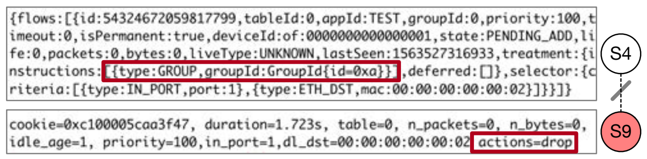
```
if (oldFlowMod.getMatch().equals(newFlowMod.getMatch())
                && oldFlowMod.getCookie().equals(newFlowMod.getCookie())
                && oldFlowMod.getPriority() == newFlowMod.getPriority()
                && dpidOldFlowMod.equalsIgnoreCase(dpid)) {
        log.debug("ModifyStrict SFP Flow");
        entriesFromStorage.get(dpid).put(entry, newFlowMod);
        entry2dpid.put(entry, dpid);
        newFlowMod = FlowModUtils.toFlowModifyStrict(newFlowMod);
        outQueue.add(newFlowMod);
        /* DELETE_STRICT and then ADD b/c the match is now different */
} else {
        log.debug("DeleteStrict and Add SFP Flow");
        oldFlowMod = FlowModUtils.toFlowDeleteStrict(oldFlowMod);
        OFFlowAdd addTmp = FlowModUtils.toFlowAdd(newFlowMod);
```

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

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.

C. Infinite Synchronization by Broken Precondition (CVE-2019-1010252)

The OpenFlow protocol has evolved with support for more message types and features across versions. Thus, when adding a flow rule in the switch, we should carefully consider which OpenFlow version is installed. In the case of the group action, it was added from version 1.1, and it is associated with a predefined group ID that is used in the switch as the precondition. In this example, we identified an instance of infinite flow rule synchronization between the ONOS controller and the switch due to the flow rule inconsistency between them.

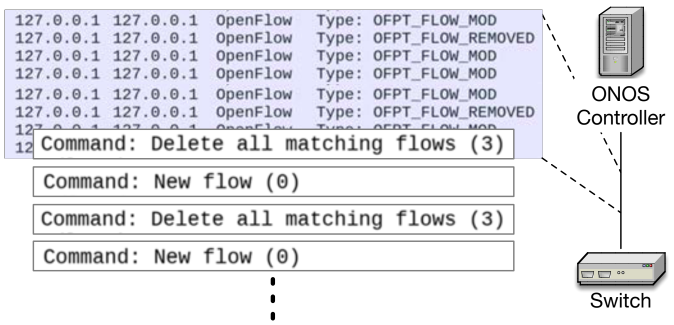


```
{flows:[{id:54324672059817799,tableId:0,appId:TEST,groupId:0,priority:100,t
imeout:0,isPermanent:true,deviceId:of:0000000000000001,state:PENDING_ADD,li
fe:0,packets:0,bytes:0,liveType:UNKNOWN,lastSeen:1563527316933,treatment:{i
nstructions:[{type:GROUP,groupId:GroupId{id=0xa}}],deferred:[]},selector:{c
riteria:[{type:IN_PORT,port:1},{type:ETH_DST,mac:00:00:00:00:00:02}]}}]}
```

```
cookie=0xc100005caa3f47, duration=1.723s, table=0, n_packets=0, n_bytes=0,
idle_age=1, priority=100,in_port=1,dl_dst=00:00:00:00:00:02 actions=drop
```

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

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.



```
127.0.0.1 127.0.0.1  OpenFlow   Type: OFPT_FLOW_MOD
127.0.0.1 127.0.0.1  OpenFlow   Type: OFPT_FLOW_REMOVED
127.0.0.1 127.0.0.1  OpenFlow   Type: OFPT_FLOW_MOD
127.0.0.1 127.0.0.1  OpenFlow   Type: OFPT_FLOW_MOD
127.0.0.1 127.0.0.1  OpenFlow   Type: OFPT_FLOW_MOD
127.0.0.1 127.0.0.1  OpenFlow   Type: OFPT_FLOW_REMOVED
```

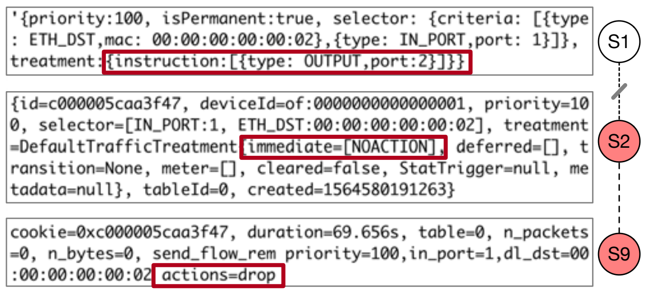
```
Command: Delete all matching flows (3)
Command: New flow (0)
Command: Delete all matching flows (3)
Command: New flow (0)
```

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

Furthermore, when reproducing this example, we discovered that messages are continuously exchanged between the controller and the switch [26], (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.

D. 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.



```
'{priority:100, isPermanent:true, selector: {criteria: [{type
: ETH_DST,mac: 00:00:00:00:00:02},{type: IN_PORT,port: 1}]},
treatment:{instruction:[{type: OUTPUT,port:2}]}}
```

```
{id=c000005caa3f47, deviceId=of:0000000000000001, priority=10
0, selector=[IN_PORT:1, ETH_DST:00:00:00:00:00:02], treatment
=DefaultTrafficTreatment{immediate=[NOACTION], deferred=[], t
ransition=None, meter=[], cleared=false, StatTrigger=null, me
tadata=null}, tableId=0, created=1564580191263}
```

```
cookie=0xc000005caa3f47, duration=69.656s, table=0, n_packets
=0, n_bytes=0, send_flow_rem priority=100,in_port=1,dl_dst=00
:00:00:00:00:02 actions=drop
```

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

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 it, 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. 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.

E. Summary

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 semantics 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

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**		
Element	NO	IT	BP	NO	IT	BP
priority	✓	✓	-	✓	-	-
timeout	✓	✓	-	✓	-	-
out_port	✓	-	✓	-	✓	-
groupd_id	✓	✓	✓	-	✓	✓
table_id	-	-	✓	-	-	✓
in_port	✓	✓	-	✓	-	-
eth_src	-	-	-	-	-	-
eth_dst	-	-	-	-	-	-
eth_type	✓	-	-	✓	-	-
ip_proto	✓	✓	✓	✓	-	-
ipv4_src	-	-	✓	-	-	-
ipv4_dst	-	-	✓	-	-	-
tcp_src	✓	✓	✓	✓	-	✓
tcp_dst	✓	✓	✓	✓	-	✓
udp_src	✓	✓	✓	✓	-	✓
udp_dst	✓	✓	✓	✓	-	✓
ipv6_src	-	-	✓	-	-	-
ipv6_dst	-	-	✓	-	-	-
Total	11	9	12	9	2	6

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

VI. Related Work

There are several studies describing threat models and attack scenarios in SDNs [4], [27]. Kreutz et al. [27] introduce possible attack vectors of SDN, and Yoon et al. [4] analyze SDN-specific attack models and vulnerabilities according to CIA metrics (Confidentiality, Integrity, and Availability).

For the control plane in SDN, Indago [28] and Shield [29] show how easily SDNs can be corrupted by an SDN application bug and propose means to identify whether an application is malicious or benign. Others have raised inter-application conflict issues, access control problems in SDN, and then propose secure SDN controller architectures [6], [11], [12], [30].

Limited prior work on data plane security [7], [31], [32] largely relied on ad hoc empirical methods to document security flaws from diverse perspectives. In contrast, AudiSDN systematically and synthetically tests and detects network policy inconsistency through the SDN stack. SDN security testing such as DELTA [8] and BEADS [33] use fuzzing techniques to find bugs and vulnerabilities in SDNs by manipulating OpenFlow messages. Most similar to our work, RE-CHECKER [34] and AIM-SDN [9] 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. Specifically, while AIM-SDN [9] mentions the potential for inconsistency between an administrator and the control plane, they find just a single software bug in a policy DB (other use cases are not related to inconsistency) and they do not provide a systematic technique for enumerating network inconsistency issues. As a result, unlike 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 [35], [36]. However, this problem is orthogonal to ours, as they do not address bugs in software implementations. Likewise, Veriflow [13], Header Space Analysis [14], and NetKat [17] provide methods for analyzing SDN flow rules to detect possible conflicts. In addition, Ravana [15] and Covisor [16] introduce methods to update SDN flow rules without conflict. VeriDP [18] proposes a way of checking the flow rule integrity between the control plane and the data plane. While these studies propose an efficient way of finding possible network rule conflicts, they do NOT consider the problem of software bugs leading to inconsistencies between components in the SDN network.

VII. Conclusion

The paper presents 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. The paper is the first 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 (through the extension of its flow-state transition model) be extended to analyze other SDN architectures beyond OpenFlow.

Acknowledgment

This work was partly supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2018-0-00254, SDN security technology development), Electronics and Telecommunications Research Institute(ETRI) grant funded by the Korean government (20ZR1300, Research on Intelligent Cyber Security and Trust Infra), and ONR N00014-18-1-2660.

References

[1] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for openflow networks," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 121–126.

[2] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "Sdn security: A survey," in *Future Networks and Services (SDN4FNS), 2013 IEEE SDN For*. IEEE, 2013, pp. 1–7.

[3] D. Kreutz, F. M. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[4] C. Yoon, S. Lee, H. Kang, T. Park, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Flow wars: Systemizing the attack surface and defenses in software-defined networks," *IEEE/ACM Transactions on Networking (TON)*, vol. 25, no. 6, pp. 3514–3530, 2017.

[5] S. W. Shin and G. Gu, "Attacking software-defined networks: A first feasibility study," in *ACM SIGCOMM workshop on Hot topics in software defined networking*. SIGCOMM, 2013, pp. 165–166.

[6] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, "Rosemary: A robust, secure, and high-performance network operating system," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*, November 2014.

[7] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: Detecting security attacks in software-defined networks." in *NDSS*, vol. 15, 2015, pp. 8–11.

[8] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras, "Delta: A security assessment framework for software-defined networks," in *Proceedings of NDSS*, vol. 17, 2017.

[9] V. H. Dixit, A. Doupé, Y. Shoshitaishvili, Z. Zhao, and G.-J. Ahn, "Aim-sdn: Attacking information mismanagement in sdn-datastores," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 664–676.

[10] J. M. Dover, "A switch table vulnerability in the open floodlight sdn controller," *Relatório técnico*, 2014.

[11] C. Yoon, S. Shin, P. Porras, V. Yegneswaran, H. Kang, M. Fong, B. O'Connor, and T. Vachuska, "A security-mode for carrier-grade sdn controllers," in *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM, 2017, pp. 461–473.

[12] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, "Attacking the brain: Races in the sdn control plane," in *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017, pp. 451–468.

[13] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 15–27. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid

[14] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228311

[15] N. Katta, H. Zhang, M. Freedman, and J. Rexford, "Ravana: Controller fault-tolerance in software-defined networking," in *Symposium on Software Defined Networking (SDN) Research, SOSR 2015*. Association for Computing Machinery, Inc, 2015.

[16] X. Jin, J. Gossels, J. Rexford, and D. Walker, "Covisor: A compositional hypervisor for software-defined networks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 87–101. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/jin

[17] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: ACM, 2014, pp. 113–126. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535862

[18] P. Zhang, H. Li, C. Hu, L. Hu, L. Xiong, R. Wang, and Y. Zhang, "Mind the gap: Monitoring the control-data plane consistency in software defined networks," in *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '16. New York, NY, USA: ACM, 2016, pp. 19–33. [Online]. Available: http://doi.acm.org/10.1145/2999572.2999605

[19] ONF, "Onos project, 2.1.0, 2019," https://onosproject.org/.

[20] Floodlight, "1.2, project floodlight, 2016," http://www.projectfloodlight.org/floodlight.

[21] "Openflow switch specification: Version 1.3.0," 2012, https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf.

[22] D. B. West *et al.*, *Introduction to graph theory*. Prentice hall Upper Saddle River, NJ, 1996, vol. 2.

[23] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "Onos: towards an open, distributed sdn os," in *Proceedings of the third workshop on Hot topics in software defined networking (HotSDN'14)*. ACM, 2014.

[24] FuzzDB, "Dictionary of attack patterns and primitives for black-box application fault injection and resource discovery." https://github.com/fuzzdb-project/fuzzdb.

[25] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.

[26] A. Orebaugh, G. Ramirez, and J. Beale, *Wireshark & Ethereal network protocol analyzer toolkit*. Elsevier, 2006.

[27] D. Kreutz, F. M. V. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*, August 2013.

[28] C. Lee, C. Yoon, S. Shin, and S. K. Cha, "Indago: A new framework for detecting malicious sdn applications," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 2018, pp. 220–230.

[29] C. Lee and S. Shin, "Shield: an automated framework for static analysis of sdn applications," in *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*. ACM, 2016, pp. 29–34.

[30] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the software-defined network control layer," in *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS), San Diego, California*, 2015.

[31] K. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, February 2015.

[32] R. Skowyra, L. Xu, G. Gu, V. Dedhia, T. Hobson, H. Okhravi, and J. Landry, "Effective topology tampering attacks and defenses in software-defined networks," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 374–385.

[33] S. Jero, X. Bu, C. Nita-Rotaru, H. Okhravi, R. Skowyra, and S. Fahmy, "Beads: Automated attack discovery in openflow-based sdn systems," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 311–333.

[34] S. Woo, S. Lee, J. Kim, and S. Shin, "Re-checker: Towards secure restful service in software-defined networking," in *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2018, pp. 1–5.

[35] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: Change you can believe in!" in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, 2011.

[36] T. D. Nguyen, M. Chiesa, and M. Canini, "Decentralized consistent updates in sdn," in *Proceedings of the Symposium on SDN Research*, ser. SOSR, 2017.