DELTA: A Security Assessment Framework for Software-Defined Networks

Seungsoo Lee† Changhoon Yoon† Chanhee Lee† Seungwon Shin† Vinod Yegneswaran‡ Phillip Porras‡
† KAIST ‡SRI International
{lss365, chyoon87, mitzvah, claude}@kaist.ac.kr {vinod, porras}@csl.sri.com

***Abstract*—Developing a systematic understanding of the attack surface of emergent networks, such as software-defined networks (SDNs), is necessary and arguably the starting point toward making it more secure. Prior studies have largely relied on ad hoc empirical methods to evaluate the security of various SDN elements from different perspectives. However, they have stopped short of converging on a systematic methodology or developing automated systems to rigorously test for security flaws in SDNs. Thus, conducting security assessments of new SDN software remains a non-replicable and unregimented process. This paper makes the case for automating and standardizing the vulnerability identification process in SDNs. As a first step, we developed a security assessment framework, DELTA, that reinstantiates published SDN attacks in diverse test environments. Next, we enhanced our tool with a protocol-aware fuzzing module to automatically discover new vulnerabilities. In our evaluation, DELTA successfully reproduced 20 known attack scenarios across diverse SDN controller environments and discovered seven novel SDN application mislead attacks.**

I. Introduction

With the increasing interest and visibility of SDN protocols, so too grows the increasing need for thorough security assessments of SDN components and component interactions. This need has not gone unnoticed by security researchers. Indeed, several security challenges have been raised in prior work [23], [31], [38], and an even wider range of attack scenarios have been enumerated against SDN environments [1], [17], [22].

Such security-critical reviews of SDNs offer a view into various breaches, but overall, the attack surfaces thus far explored have been quite limited to either highly targeted exploits, such as ARP spoofing, or specific vulnerabilities that arise in various SDN components. Each previous result may not be applicable to other SDN environments (e.g., different control planes). Hence, operators seeking to assess security issues in their SDN environments need to survey existing SDN security-related studies and determine relevance on a case-by-case basis. Furthermore, an operator may have to adapt or redesign deployment-specific security test suites.

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23457

This paper introduces a new SDN security evaluation framework, called DELTA, which can automatically instantiate attack cases against SDN elements across diverse environments, and which may assist in uncovering unknown security problems within an SDN deployment. Motivated by security testing tools in the traditional network security domain [13], [36], DELTA represents the first security assessment tool for SDN environments. Moreover, we enhanced our tool with a specialized fuzzing module [26] to exploit opportunities for discovering unknown security flaws in SDNs.

In designing DELTA, we first assessed the overall operation of an SDN by tracking its operational flows. Operational flow analysis provides a basis for understanding the attack surfaces available to external agents across an SDN deployment, and is a generally applicable strategy for approaching any SDN stack. Based on the popular OpenFlow protocol specification [29], we categorize operational flows into three categories (see Section II). In each category, we explore possible security issues and assess which ones are covered by existing studies.

Our intent is to design a testing framework that automates the systematic exploration of vulnerabilities exposed in SDN deployments from diverse perspectives. To date, previous studies are limited in their coverage of the SDN attack surface, in that they usually depend on specific SDN elements or network environments. To overcome this issue, we devised a method to reveal possible unknown security problems in an SDN by employing a blackbox fuzzing technique, which randomizes message input values to detect vulnerabilities in the direct interface or failures in the downstream message processing logic. When generating random test vectors, DELTA uses the information from the analysis of the SDN operations and focuses on the cases where vulnerabilities are likely to be located.

We implemented a prototype framework for DELTA and evaluated it with real-world SDN elements. For each controller, DELTA is customized with a simple configuration file. The flexible design of DELTA accommodates both open source and commercial SDN controller implementations. Our prototype can (currently) reproduce 20 known SDN-related attack scenarios targeting several well-known SDN elements, such as the ONOS controller [2], the OpenDaylight (ODL) controller [25], the Floodlight controller [27] and the commercial Brocade Vyatta SDN controller [3]. In addition, DELTA was able to discover seven new attack scenarios by applying protocol fuzzing techniques.

The new attack scenarios exposed by DELTA have been reported to the Open Networking Foundation (ONF) [28], the

standards body dedicated to the promotion of SDN and defining the OpenFlow protocol [12]. The results of our analysis have also informed our collaborations with the ONF security working group in drafting white papers defining best practices for securing SDN environments. Furthermore, DELTA has been open sourced as one of ONF's official open SDN projects [33].

This paper describes the following contributions:

- An analysis of vulnerabilities in the SDN stack that can mislead network operations. Through this analysis, we can reconcile test input with erroneous SDN errors and operational failures. We introduce seven criteria for automatically detecting a successful attack scenario from these failure conditions. We then show how to combine this information for assessing root cause analysis on successful attacks.
- The development of an automated security assessment framework for SDN capable of reproducing diverse attack scenarios. This framework currently reproduces 20 attack scenarios against real-world SDN elements with simple configurations and is readily extensible to support more scenarios.
- The incorporation of blackbox fuzzing techniques into our framework to detect new unknown attack scenarios. Through our evaluation, we verified that this technique found seven previously unknown attack cases.
- The demonstration of flexibility of system design by evaluating it against three popular, open-source SDN controllers and the commercial Brocade Vyatta SDN controller.

We have shared our results with ONF and they are being referenced by its security working group in defining new standards. Furthermore, our framework has been promoted to an official open source project by ONF [33].

II. Background and Motivation

A. SDN and OpenFlow

In traditional networks, a control plane, computing sophisticated networking functions, and a data plane, handling low-level packet forwarding based on the policies of the control plane, are tightly coupled and usually colocated within a single device. Since these two planes are often embedded within a proprietary network device, it is inherently challenging to insert new functions into the device without specialized knowledge or vendor cooperation.

To overcome this fundamental problem, software-defined networking 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. This separation allows network administrators to manage complicated traffic within a centralized network view. The key features of SDN are high-level network abstraction and programmability. Another difference between SDNs and traditional networks is a strong movement away from static network policies, in which an administrator manually configures network flow-handling policies for each network device. Instead, an SDN abstracts the complexity of the underlying networks, and this abstraction hides network topologies and network devices from end-users. Thus, unlike traditional networks, SDN contains the global view of the whole network through the high-level network abstraction.

OpenFlow: OpenFlow is the de-facto standard protocol for the communication between the control plane (a.k.a., the OpenFlow controller[1]) and the data plane. Hence, considering OpenFlow in SDN networks is quite natural, and many commercial deployments have successfully employed OpenFlow as the primary interface between those two planes [16], [20].

B. SDN Control Flows

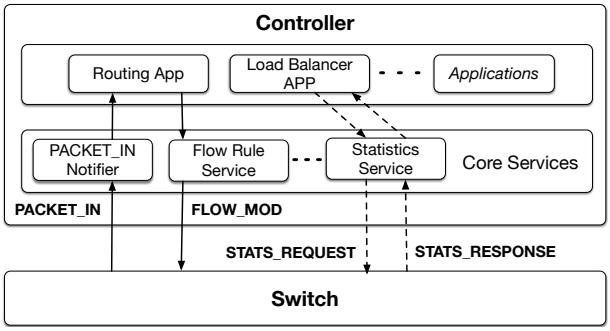

Fig. 1: Examples of SDN/OpenFlow control flows. Dotted lines indicate symmetric control flows and solid lines correspond to asymmetric control flows.

The operations of SDN (specifically, those based on the OpenFlow protocol) can be classified into three types of control based on the control flow of the OpenFlow protocol: (i) symmetric control flow operations, (ii) asymmetric control flow operations, and (iii) intra-controller control flow operations.

Symmetric control flow operations: In these operations, an SDN component sends a request to another component and receives a reply back (i.e., a request-reply pair). Figure 1 presents an example illustrating this operation using dotted lines. Consider a load balancer on a controller that needs switch statistics to distribute traffic loads. To retrieve statistics from the switch, the load balancer first issues a *statistics request* event to the statistics service in the controller core. Once the service notices the event, it sends the `STATS_REQUEST` message to the switch through an OpenFlow message. Then, the switch packs its statistics information in the `STATS_RESPONSE` message and sends it to the controller again. Finally, the statistics service returns the received statistics to the load balancer.

Asymmetric control flow operations: In contrast to the previous operation, some SDN operations only involve unidirectional messaging (e.g., messages that do not require a reply). Technically, most SDN control-flow interactions fall under asymmetric control flows (e.g., control for handling packet

[1]In the case of OpenFlow-based SDN networks, the term *controller* is commonly used to denote the control plane. This paper uses both terms interchangeably.

arrival and inserting flow policy). The solid lines in Figure 1 represent two kinds of asymmetric control flows (PACKET_IN and FLOW_MOD). Once a packet arrives at the switch, the switch first matches the packet with the flow entries in its flow table. If the switch cannot find any matching flow entries, it sends a PACKET_IN message containing a portion of the packet header to the controller. Then, the controller delivers the packet-arrival event to its applications. This message passing occurs in one asymmetric control flow as packets arrive. The other asymmetric control flow is started from the application on the controller. For example, once a routing application receives the packet arrival event, it must decide how best to process the event (e.g., forwarding the packet to somewhere or dropping the packet). Thus, the routing application may inform the data plane at which port the packet should be output. After the routing application issues a packet-forwarding policy to the Flow-Rule service, the service sends a FLOW_MOD message to the switch. Finally, the switch inserts the packet-forwarding policy into its flow table and forwards the packet.

Intra-controller control flow operations: Unlike symmetric and asymmetric control flows, *intra-controller control flows* are initiated by applications running on a controller (i.e., control plane). When applications interact with one another or use the internal services of the controller, they do so by employing the APIs exposed by the controller. If a routing application requires the topology information from the internal services to compute a dynamic routing path, the routing application calls an API that returns the current topology information. This API function may in turn invoke several internal APIs. Finally, the topology information is delivered to the routing application. This call-chain is an example of intra-controller control flow.

C. Motivating Example

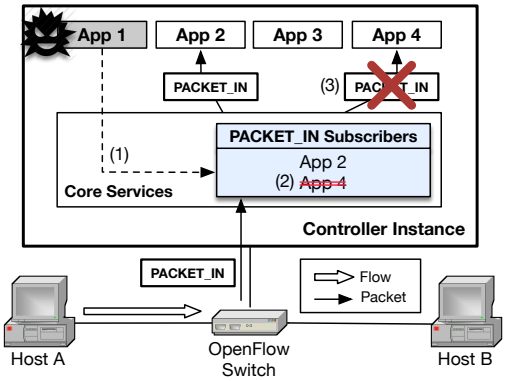

Fig. 2: Event Listener Unsubscription attack

Figure 2 illustrates how a malicious application could make a benign application incapable of receiving any of the necessary control messages from a switch. In this instance, a malicious application (App 1) accesses the list knowing which application receives the PACKET_IN control message (the most important control message), and discovers that App 4 is waiting for PACKET_IN messages (1). Then, App 1 unsubscribes App 4 from the list (2), and thus App 4 is unable to receive any PACKET_IN messages (3).

This is a real working example (applicable to Floodlight [27] and OpenDaylight [25] controllers), and it illustrates how a malicious application confuses a benign application by manipulating the intra-controller control flow operation.

III. Related Work

Our work is inspired by prior work in SDN security and vulnerability-analysis techniques.

SDN Security and Attacks: There have been several studies [1], [22] dealing with attack avenues in SDNs. Kreutz et al. argue that the new features and capabilities of SDN, such as a centralized controller and the programmability of networks, introduce new threats [22]. Benton et al. point out that failures due to lack of TLS adoption by vendors for the OpenFlow control channel can make attacks such as man-in-the-middle attacks and denial of service attacks easier [1]. Moreover, some researchers have raised other issues, such as inter-application conflicts, access control, topology manipulation, and sharing relationships [5], [9], [17], [31], [39]. Röpke et al. [32] have demonstrated that SDN applications can launch stealth attacks and discussed how such applications can be easily distributed via third-party SDN app stores, such as the HP App Store [19]. Besides, even without delivering malicious SDN applications to SDNs, Dover et al. have also shown that it is possible to launch denial-of-service and spoofing attacks by exploiting the implementation vulnerability that exists in the switch management module of Floodlight [10], [11]. Although there have been several studies on SDN vulnerabilities, contemporary controllers remain vulnerable to many of these attacks. Hence, we propose a software framework that can simplify reproducibility and verification of diverse attack scenarios.

Vulnerability Detection Tools and Techniques: Traditional network security testing tools such as Metasploit [24], Nessus [36], and Nmap [13] are equipped with a rich library of vulnerabilities and composable attack modules. However, because these tools are specialized for legacy and wide-area networks, they are directly unsuitable for SDN networks. In a recent BlackHat briefing, the presenters explored the SDN attack surface by actually attacking each layer of SDN stack, and then demonstrated some of the most critical attacks that directly affect the network availability and confidentiality. Thus, SDN-specific security threats are complex and cannot be revealed by existing network security testing tools, such as Metasploit [24] and Nessus [36], as they are not SDN-aware.

Our goal is to develop an analogous tool for OpenFlow networks. Fuzz testing was first proposed by Miller et al. in the early 1990s and has steadily evolved to become a vital tool in software security evaluation [26], [41]. The current body of work in black-box fuzz testing may be broadly divided into mutational and generation- (or grammar-) based techniques. While the former strategies rely on mutating input samples to create test inputs, the latter develop models of input to derive new test data. DELTA makes use of both strategies, with mutational being the primary approach.

Examples of mutational fuzzers include SYMFUZZ [4], zzuf [15], BFF [18], AFL-fuzz [46] and PacketVaccine [43]. Unlike these approaches, our system employs a fuzz-testing methodology that is specialized for SDNs. We recognize that because the operations and topologies of SDNs are more dynamic than traditional networks, randomization of a specific portion of the packets is insufficient. Hence, we classify the

Flow Type	*Attack Code*	*Attack Name*	*Controller*		
			ONOS	*OpenDaylight*	*Floodlight*
Symmetric Flows	SF-1	Switch Table Flooding [11]	X	X	O
	SF-2	Switch Identification Spoofing [10]	X	O	O
	SF-3	Malformed Control Message [37]	X	O	O
	SF-4	Control Message Manipulation [35]	O	O	O
Asymmetric Flows	AF-1	Control Message Drop [35]	O	O	O
	AF-2	Control Message Infinite Loop [35]	O	O	O
	AF-3	PACKET_IN Flooding [21], [38], [40]	O	O	O
	AF-4	Flow Rule Flooding [8], [38], [45]	O	O	O
	AF-5	Flow Rule Modification [35]	O	O	O
	AF-6	Switch Firmware Misuse [35]	O	O	O
	AF-7	Flow Table Clearance [35]	O	O	O
	AF-8	Eavesdrop [35]	O	O	O
	AF-9	Man-In-The-Middle [35]	O	O	O
Intra-Controller Control Flows	CF-1	Internal Storage Misuse [39]	O	O	O
	CF-2	Application Eviction [39]	O	O	N/A
	CF-3	Event Listener Unsubscription [39]	N/A	O	O
Non Flow Operations	NF-1	System Command Execution [39]	O	O	O
	NF-2	Memory Exhaustion [39]	O	O	O
	NF-3	CPU Exhaustion [39]	O	O	O
	NF-4	System Variable Manipulation [35]	X	O	O

TABLE I: Summary of known SDN attack cases: N/A means this attack is not available. O means the controller is vulnerable to this attack, and X means that it is not vulnerable.

operations of SDN into three types of control based on the control flow, and incorporate the features of those operations into DELTA's fuzzing module. ShieldGen is an example of a grammar-based fuzzer, that uses knowledge of data formats and probing, to automatically generate vulnerability signatures and patches from a single attack instance [7]. Godefroid et al. present a grammar-based whitebox fuzz-testing approach inspired by symbolic execution and dynamic test generation [14]. Unlike such approaches, DELTA does not require the entire source code of the target system. Scott et al. introduced a troubleshooting system called STS that automatically inspects vulnerabilities in control platforms using a fuzzing technique [34]. The focus of STS is identifying the MCS (minimal causal sequence) associated with a bug. However, DELTA reproduces known vulnerabilities and even finds unknown ones by changing the parameters of its fuzzing modules without MCS. Yao et al. proposed a new formal model and corresponding systematic blackbox test approaching for SDN data plane [44]. While this approach mainly focuses on the testing paths of SDN data planes, DELTA applies fuzzing functions to discover unknown security flaws.

IV. Vulnerabilities in SDN Flows

This section explores how the three SDN flow operations described in Section II are related to vulnerabilities that can harm SDN operations. Vulnerabilities related to the SDN control flows are discussed in Section IV-A, and the locations of vulnerabilities resulting from irrelevant flow operation are described in Section IV-B. Table I provides a high-level overview of SDN vulnerabilities. It also denotes which attacks relate to the controller (i.e., control plane). However, some attacks highly depend on the architecture of a specific controller. Here, we consider three well-known controllers: ONOS, OpenDaylight, and Floodlight, as these controllers arguably represent the most popular and widely used SDN controllers in use today.

A. SDN Control Flow Operation Vulnerabilities

Symmetric Control Flow Vulnerabilities: Table I identifies four symmetric control flow vulnerabilities. One vulnerability arises in the presence of weak authentication during the handshake step. This vulnerability can lead to replay attacks to the controller, such as the *Switch Identification Spoofing* attack. For example, the Floodlight controller classifies the identification of the connected switch according to a Data Plane ID (DPID). However, a man-in-the-middle (MITM) concern arises, in which an attacker replays handshake steps with the DPID of an already connected switch causing Floodlight to disconnect itself from the switch. Also, as Floodlight manages the connected switch's information in its internal storage, it consumes the memory resources within the host. An attacker can persistently replay meaningless handshake messages to exhaust the internal storage of the controller (i.e., *Switch Table Flooding* attack). Such an attack could result in a controller shutdown.

In addition to the relay attack, symmetric control flow messages also enable *Malformed Control Message* attacks. Each control message carries the OpenFlow version information in its header, which should be consistent with other messages. If the attacker replaces the version value in a response message within a symmetric control flow with an invalid value, an inconsistency arises that may result in a switch disconnection. The *Control Message Manipulation* attack is similar to the *Malformed Control Message* attack; however, in this case, the attacker manipulates the header type of symmetric control flows leading to switch disconnection.

Asymmetric Control Flow Vulnerabilities: Table I identifies nine asymmetric control flow vulnerabilities. Most controllers maintain a listener mechanism that allows applications to register to receive specific messages from the data plane. When a message arrives at the controller, this mechanism delivers the message to the applicable registered applications,

either in sequence or parallel, depending on the implementation of controllers. Misbehaving or rogue applications can interfere with the order of applications in the list, and cause the application to drop the message (i.e., *Control Message Drop* attack). Malicious applications can alternatively implement an infinite loop (i.e., *Control Message Infinite Loop* attack) to prevent other applications from acting on the message.

Controllers and switches are also vulnerable to performance degradation by malicious or erroneous applications. One such example is the *PACKET_IN Flooding* attack. In this scenario, an adversary generates a number of meaningless flows to other hosts in order to trigger a flood of PACKET_IN messages to the controller, which eventually degrades the performance of the controller. On the other hand, a *Flow Rule Flooding* attack is also feasible. A malicious application can install a number of flow rules through FLOW_MOD messages to the target switch to overflow the flow table, which could lead the switch into an unpredictable state.

A malicious application may also manipulate resident flow rules in the switch that have been installed by other applications. For instance, although a flow rule installed by a firewall application may instruct the switch to drop the flows from the malicious host, a peer application could modify the flow rule to forward corresponding flows from the malicious host (i.e., *Flow Rule Modification* attack). Also, by changing the rules, the malicious application can manipulate the flow table in the switch (i.e., *Firmware Misuse* and *Flow Table Clearance* attack).

When the control messages between the controller and the switch are unencrypted, an attacker located between them can guess what topology is constructed by sniffing control messages (i.e., *Eavesdrop* attack) in a passive manner. The attacker may also intercept the control message and then change some field values of the control messages with malicious intent (i.e., *Man-In-The-Middle* attack).

Intra-Controller Control Flow Vulnerabilities: Table I identifies three intra-controller control flow vulnerabilities. Unfortunately, most controllers do not provide access control mechanisms to limit API usage among applications. A malicious application may access and alter network topology data within the internal storage of the controller, impacting all peer applications that derive flow control decisions based on this network topology data (i.e., an *Internal Storage Misuse* attack).

Some controllers provide a mechanism that dynamically controls applications running on the controller. The problem with this is that a malicious application can also abuse this mechanism without any constraint. For example, the malicious application can dynamically unload a Firewall application (i.e., *Application Eviction* attack). Also, the malicious application can prevent some applications which want to receive the control message from the switch from being notified of the control message (i.e., *Event Listener Unsubscription* attack).

B. Non Flow Operation Vulnerabilities

Table I identifies four non-flow operation vulnerabilities. Although SDN controllers have been referred to network operating systems (NOS), most controllers are implemented as general networking applications. Thus, controllers are unfortunately subject to the same vulnerabilities as found in normal applications. For instance, a developer who implements an application running on the controller could make a mistake inside the application logic, which can cause the termination of the application (i.e., *System Command Execution* attack). Since most controllers employ the multi-threaded programming paradigm, the termination of the application can mislead the controller into shutdown. If a target network does not have controller redundancy, this could result in a network-wide outage.

The malicious application can intentionally consume all available system resources of a controller to affect other applications or even the controller. For instance, malicious applications can halt the control layer by intentional unconstrained memory consumption (i.e., *Memory Exhaustion* attack), or by unconstrained thread creation to exhaust available CPU cycles (i.e., *CPU Exhaustion* attack). System time is also considered a system resource that is used to check the response time of symmetric control flows. If the malicious application manipulates this system time, the switch connected to the controller could enter an unexpected state (i.e., *System Variable Manipulation* attack).

V. System Design

This section discusses the design considerations motivating our design and then describes the DELTA system architecture.

A. Design Considerations

Table I reviewed 20 known attack scenarios, which can be reproduced without much difficulty. If testing each case requires a different testing environment, the costs of conducting these tests can rapidly become prohibitive, even within highly sensitive computing environments. Thus, the testing framework should be easily configured and correctly reproduced. In addition, these attack scenarios should be operated with diverse SDN components, such as different control planes and network applications, to cover most possible attack surfaces of SDN.

Given these practical testing concerns, the requirements driving our penetration framework can be summarized as follows: (i) it should cover as many attack scenarios as possible, (ii) it should be highly automated, to minimize the human skills and time necessary to conduct testing, and (iii) it should be inter-operable with a diverse set of SDN components. In addition, we also require that our framework be easily extensible to new test cases, and assist in the identification of entirely new attack scenarios. The following sections will consider these requirements in more detail.

B. Blackbox Fuzzing

As previously summarized, 20 attacks against SDN have been presented so far. However, a wider range of undiscovered attack scenarios against SDNs remain, which our framework can help operators explore and discover. To identify such unknown attack cases, we borrow the notion of *fuzz testing* developed in the context of legacy software and protocol testing. Fuzz testing allow the development of entirely randomized testing vectors to determine if program interfaces are subject to unexpected input handling errors. We choose blackbox fuzzing, rather than whitebox fuzzing, because the former does not require the source code of target programs, and it can be

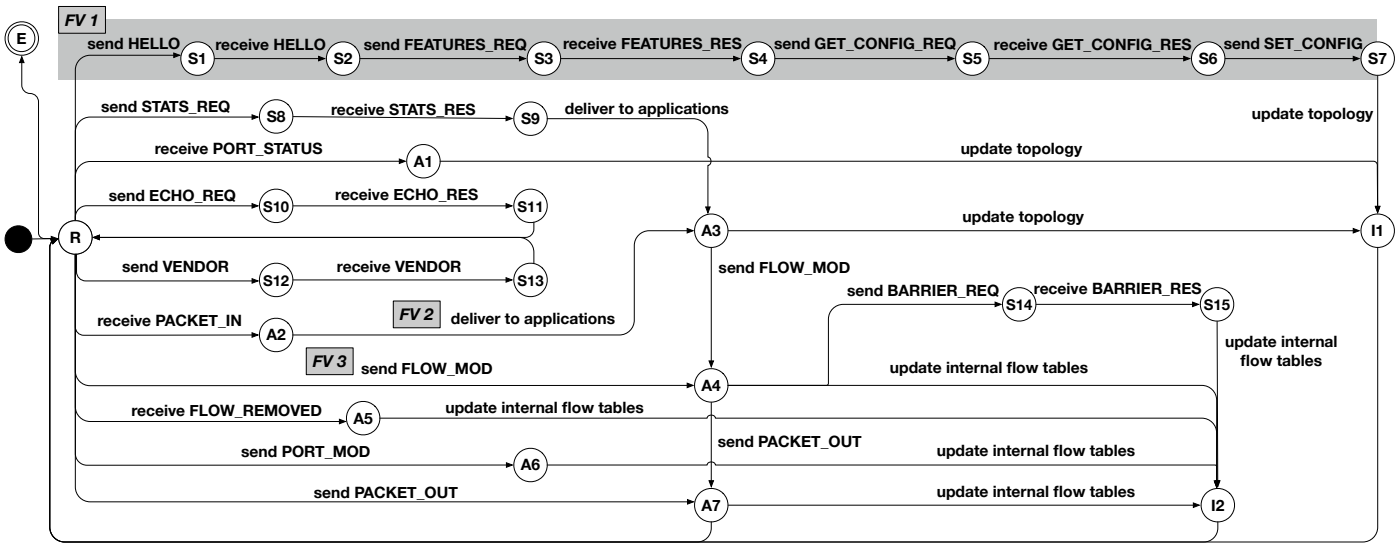

Fig. 3: Operational state diagram of typical SDN controller and fuzzing vector (FV) examples

applied to both open source and proprietary SDN components and devices.

State Diagram of SDN Control Message: A key analysis in blackbox fuzzing is that of determining the input parameters that must be subject to input randomization, which is a central consideration in our framework design. Instead of selecting values for randomization in an ad hoc manner, we derive those values from the analysis of SDN control flows.

The SDN operations of a typical SDN controller, which employs OpenFlow protocol v1.0 [29], can be represented in an operational state diagram as shown in Figure 3. Although we only present the state diagram for OpenFlow v1.0, we have also analyzed OpenFlow v1.3 [30] and it is a straightforward extension.

In the state diagram (Figure 3), label `R` stands for a ready state to receive or send the control messages. The other states are labeled in accordance with the type of control messages: (i) states labeled `S` involve symmetric control message transitions, (ii) states labeled `A` involve asymmetric control message transitions, and (iii) states labeled `I` involve API calls generating intra-controller control messages.

Each edge designates the type of control message or the specific controller behavior that triggered the state transition. For example, as shown in Figure 3, when a controller in `R` state receives a PACKET_IN message from a switch, the state of the controller transitions to `A2`. In `A2`, the controller delivers the message to the applications, causing another state transition to the next state `A3`, where the controller verifies if the PACKET_IN includes link information. Once the packet is verified to include link information, the controller updates the network topology, moves to `I1` state, and finally, comes back to the ready state. As illustrated, the state diagram can clearly describe the points at which the controller takes the input and how each input induces the state transition. Therefore, based on such an operational analysis result, we can effectively perform the input randomization against the SDN controllers.

Next, based on the state diagram, we investigated (i) the sequence of control flows, presented in Section II, to determine whether there are candidate control flows for randomization (Fuzzing Vectors FV 1 and FV 2 in Figure 3), and then examined the (ii) input values conveyed in each control flow (FV 3 in Figure 3).

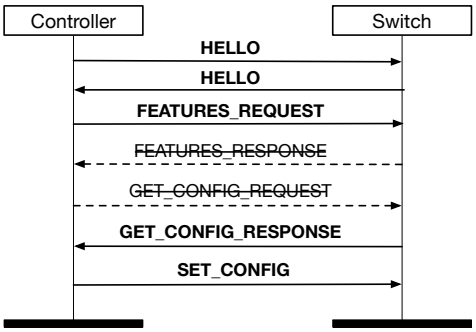

Fig. 4: Symmetric flow sequence randomization example

Randomizing Control Flow Sequence: We can randomize the control flow sequence in two major steps: (i) inferring current state of an SDN controller, and (ii) manipulating the control flow sequence.

Inferring the current state of an SDN controller is simple: intercept the ongoing control messages to understand and track down the operations of the controller. In the case of the symmetric control flows, the current state of the controller can be inferred from the control messages intercepted from the control channel between the controller and the switches. For example, as shown in Figure 3, the controller states from `R` to `S7` represents the OpenFlow handshake process. Based on which type of OpenFlow message is sent or received, it is possible to infer in which state the controller resides. Meanwhile, in the case of the asymmetric control flows, the state of the controller can be detected by not only intercepting the control messages but also by monitoring the changes in the controller behaviors, because some of the state transitions

in asymmetric control flows are triggered by the controller operations. As illustrated in Figure 3, the states from R to A3 describes how PACKET_IN messages are delivered to applications, and it is difficult to detect state transitions to A3 by intercepting the control messages. Thus, to detect such state transitions, we monitor any changes in the controller behavior and specifically in this example deploy an additional application to confirm the reception of PACKET_IN.

Once the state of the controller is analyzed, we can manipulate the sequence of the control flow. To randomize the sequence of the symmetric control flows, we intentionally drive an SDN controller to violate the standard protocol. Figure 3 (Fuzzing Vector 1) illustrates the control flow of the standard OpenFlow handshake that could be potentially manipulated. For example, as shown in Figure 4, it is possible to manipulate the sequence by omitting a couple of message exchanges (crossed out) to test if the controller or the switch is subject to such protocol violations.

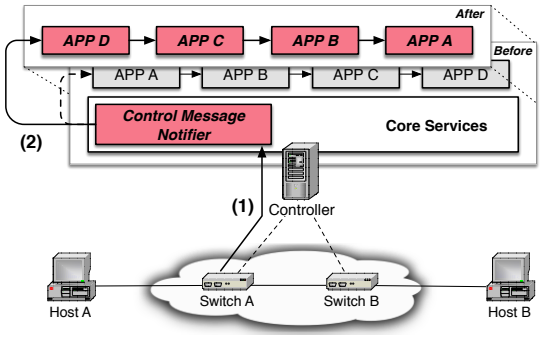

Fig. 5: Asymmetric flow sequence randomization example in the sequential order

Such control flow manipulation can be also applied to the asymmetric flows, such as the flow shown in Figure 3 (from A2 to A3 in FV 2). If a PACKET_IN message is sent to the controller by the network device (step 1 in Figure 5), the controller sequentially delivers the message to the applications in a specific order (step 2 in Figure 5). Figure 5 (Before) shows the default sequence where App A first receives the PACKET_IN message, and App D receives the message last (i.e., A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D). Here, we can change the control flow (i.e., change the order of the applications) randomly at runtime as shown in Figure 5 (After).

In addition to the sequential asymmetric control message delivery mechanism, messages can be delivered to applications in parallel as shown in Figure 6. For example, when the controller receives a PACKET_IN message (step 1 in Figure 6), it simultaneously delivers the asymmetric message to the applications (step 2 in Figure 6). However, of those applications concurrently running on the controller, a certain set of applications may be defined to follow a particular order in receiving the message. In this example, we arbitrarily injected App X, so that this application can receive the message ahead of App B (Figure 6 (After)). As demonstrated, it is possible to randomize such sequences.

Randomizing Input Values: Besides the control flow, input values of a control flow can also be randomized. For example, we selected the FLOW_MOD message, which allows

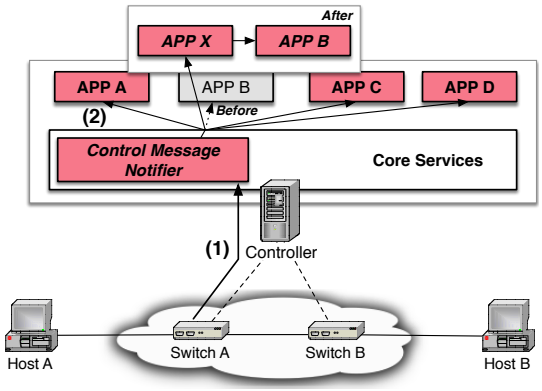

Fig. 6: Asymmetric flow sequence randomization example in the parallel order

the controller to modify the state of a switch (FV 3 in Figure 3). Most fields are defined as an unsigned integer type, and we can randomize these values to mislead the switch into parsing it (e.g., 0 or maximum). Since control messages between the data plane and the control plane are commonly delivered through a plain TCP channel[2], all field values of the control messages can be intercepted at the control channel and manipulated easily, which could result in critical network instabilities. For example, a priority field in a FLOW_MOD message can be maximized. Such field-value randomization can be also applied to the symmetric flows. Also, most controllers provide their own APIs to improve the flexibility of intra-controller control flows. These APIs may be used by any hosted network application, which means that any application has a chance to change (or randomize). Our framework adopts this idea to randomize input values of a control flow.

DELTA uses fuzzing techniques and ex-post-facto analysis to identify vulnerabilities in the target program. We have outlined seven test criteria as vulnerability detectors that trigger ex-post-facto analysis: (i) a controller crash, (ii) an application crash, (iii) internal-storage poisoning (iv) a switch disconnection, (v) switch-performance downgrade, (vi) inter-host communication disconnection, and (vii) error-packet generation. If a fuzz case generated by DELTA results in any of the following, the test inputs will be flagged for ex-post-facto vulnerability assessment.

C. System Architecture

This section presents the overall architecture of DELTA and explain each of its components. As shown in Figure 7, our framework consists of a centralized agent manager and multiple agents. The agents are classified into three different types based on their location: application, channel, and host. Those agents are located in the middle of SDN control flows and implement attack scenarios.

Agent Manager: The agent manager (AM) assumes the role of a controller that manages all the agents. The AM consists of four modules: Controller Manager, Attack Conductor, Agent Handler, and Result Analyzer. The AM is not

[2]The OpenFlow specification suggests an encryption transport (e.g., TLS) to encrypt outgoing messages; however, it is frequently disabled in favor of performance [1].

	ONOS				*OpenDaylight*				*Floodlight*				***Brocade Vyatta***
V	1.2	1.3	1.4	1.5	Hydrogen	Helium	Lithium	Beryllium	0.91	1.0	1.1	1.2	***2.3.0***
RD	6/5/15	9/18/15	12/16/15	3/10/16	2/4/14	9/29/14	6/29/15	2/22/16	12/8/14	12/30/14	4/17/15	2/7/16	***2016***
SP	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	✓	✓	✓

TABLE II: Supported application agents for various controller versions: V indicates version, RD indicates release date (MM/DD/YY), and SP indicates whether or not it is supported.

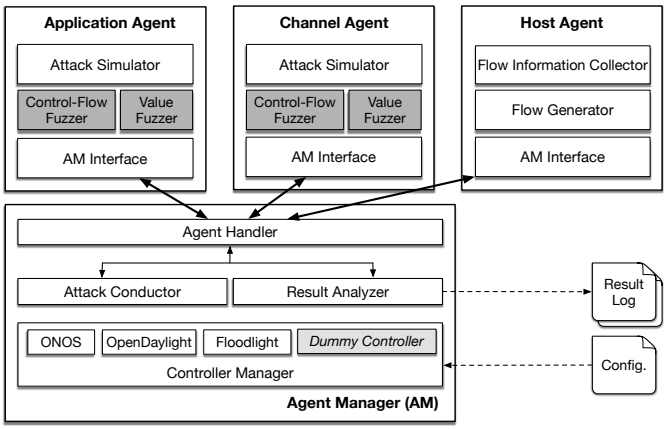

Fig. 7: Overall architecture of DELTA with four key components: (*i*) Agent Manager, (*ii*) Application Agent, (*iii*) Channel Agent, and (*iv*) Host Agent.

coupled with SDN components; it independently conducts two functions: (i) controls other agents remotely, to replay known attack scenarios or discover unknown attack scenarios against the target network, and (ii) retrieves the executed results from each agent.

In the initial stage of our framework, the controller manager reads a configuration file containing the information of a target controller (e.g., version information and installed path) because each target controller has a different way of loading network applications. For testing SDN-enabled switches, the controller manager invokes the dummy controller that behaves as if it was a simple SDN controller. Then, the attack conductor initiates known attack scenarios. Its operational scenarios are predefined in this module. When our framework replays a known attack scenario, this module controls each agent to conduct the scenario based on the defined information. This module controls other agents through the agent handler. Finally, if an attack is completed by each agent, the result will be sent back to the result analyzer module, which will report these results to the operator. The attack conductor also controls the fuzzing modules in each agent to find unknown attack cases.

Application Agent: The application agent is an SDN application running inside the controller, and it launches attacks under the supervision of the AM. Since an SDN application can be directly involved in SDN control flows, our framework inserts an application (i.e., application agent) into a controller to intercept, forge, and change SDN control flows and input variables, as applicable to the attack scenario. Application agents are controller specific, as they must interact directly with each controller APIs.

The application agent consists of four modules: (i) Attack Simulator, (ii) AM Interface, (iii) Control-Flow Fuzzer, and (iv) Value Fuzzer. The attack simulator includes known malicious functions for a target controller, and it executes malicious functions as indicated by the AM. For example, if the AM initiates an *Internal Storage Misuse* attack [39], its operational scenario is already located within the application agent. Command messages from the AM are delivered to the agent via the AM interface. The other modules (i.e., control-flow fuzzer and value fuzzer) are used to randomize SDN control flows and their input values. They will be invoked by a command message from the AM, and they will randomize required elements to detect unexpected reactions produced by the controller. More detailed descriptions of those modules are provided in the last part of this section.

Channel Agent: The channel agent sniffs and modifies the control messages passing through the control channel between the control plane (i.e., controller) and the data plane. As the communication is often unencrypted, the channel agent can manipulate control messages by intercepting them. While the application agent is controller-dependent, the channel agent is SDN protocol-dependent. DELTA currently supports OpenFlow 1.0 and 1.3, and the channel agent automatically catches this information by inspecting the header field of control messages. The internal architecture of the channel agent shares many things with the application agent. The channel agent consists of four modules: (i) Attack Simulator, (ii) AM Interface, (iii) Control-Flow Fuzzer, and (iv) Value Fuzzer. The functions of these four modules are the same as those of the application agents.

Host Agent: The host agent behaves as a host (or multiple hosts) participating in the target SDN network. It is capable of generating network traffic to any reachable targets (e.g., switch and host), and such a remotely controllable host is useful for launching some attacks initiated by hosts. For example, a host agent can send a large volume of network traffic to an SDN-enabled switch, causing a PACKET_IN flooding attack [21], [38], [40]. Unlike other agents, the host agent does not have known attack scenarios, but it can be employed to generate small or massive flows (mice or elephant flows), which can be used during attack scenarios.

The host agent consists of three modules: (i) Flow Information Collector, (ii) Flow Generator, and (iii) AM Interface. Here, the AM interface performs the same operations as that of other agents. The flow information collector captures diverse flow-related information, such as latency and the number of sent and received flows. This information is used to detect some attack types. The flow generator produces network flows under the control of the AM.

Fuzzing Modules: An administrator who chooses to employ the blackbox fuzzing functions of our framework can ask

the AM to activate fuzzing functions for the application and channel agents. If no guidelines are presented to the fuzzing functions, they operate continuously until manual termination. The operator can alternatively supply input to narrow fuzzy testing to a boundary range of randomization for the specific cases.

Currently, our framework provides two fuzzing module randomizing functions: (i) Control-Flow Fuzzer and (ii) Value Fuzzer, which are both located within each agent. As their name implies, the control-flow fuzzer randomizes SDN control flow operations, and the value fuzzer randomizes the input values of each function. These modules may operate in tandem or independently.

Two fuzzing modules in the application agent use APIs provided by a controller to randomize control flows and values respectively. Since each controller provides different types of APIs, we analyze APIs provided by well-known controllers and extract common functionalities. Then, we try to design a generalized module that can cover diverse randomization scenario in each controller. Of course, the implementation will be different in each controller, but the conceptual architecture is similar across all controllers, which simplifies the DELTA framework. In the case of the channel agent, the fuzzing modules intercept and parse ongoing control messages between a controller and the data plane to randomize control flows and values. For example, the control-flow fuzzer randomly holds one of control messages and later resends the message to manipulate the symmetric flow sequence. If a controller manages all flow-rule installations to the data plane, this information will be delivered through a network message (e.g., a TCP channel). In this case, the value fuzzer can capture and forge this message (i.e., mounting a form of MITM attack [35]).

Whenever a randomization procedure is completed, the test results will be delivered to the result analyzer in the AM, which then analyzes the results to verify the effectiveness of an attack scenario. This evaluation for detecting new successful attacks is currently based on the set of seven test criteria mentioned in the previous subsection on Blackbox Fuzzing. If any of these seven outcomes is detected, the result analyzer regards this as a new attack and reports the test case to the operator.

VI. Implementation

We have implemented an instance of DELTA to verify its feasibility and effectiveness. To support the design features described in Section V, we implemented three types of agents and an agent manager in Java, in approximately 11,000 lines of code.

DELTA currently includes application agents for three well-known open source controllers (i.e., ONOS, OpenDaylight, and Floodlight) and one commercial controller, enabling it to replay attack scenarios and launch fuzzing functions as shown in Table II (OpenDaylight-Beryllium is under development). As the controller integration design involves the user of modular application agents, we are able to minimize the integration cost (and impact) of extending DELTA to other controllers. The channel agent employs a packet capture library to capture and modify control messages between a controller and network devices, and it currently understands OpenFlow version 1.0 and 1.3. The host agent is a Java application program that generates network flows by creating new TCP connections or by using existing utilities, such as Tcpreplay [42]. It can also collect network flow information by passively sniffing network packets. All agents have direct connections to the agent manager (AM) with TCP connections. We implement fuzzing modules by modifying functions for controlling SDN operations. In the case of the application agent, the fuzzing modules parse arguments of each function, track the sequence of function call, and randomize arguments or the sequence of function call based on the information provided by the AM. With respect to the channel agent, the fuzzing modules manipulate OpenFlow messages and delay the sequence of message flows.

VII. Evaluation

We conducted a wide range of experiments and performance evaluations involving the DELTA security assessment framework with well-known SDN controllers, ONOS(v1.4.0), OpenDaylight (Helium), Floodlight (v1.2), and a commercial controller (Brocade Vyatta v2.3.0).

A. Use Case 1: Finding Unknown Attacks

Among the key features of DELTA is its ability to use specialized fuzz testing to uncover new SDN vulnerabilities. Here, we highlight this capability using experiments we conducted on ONOS, OpenDayLight (ODL), and Floodlight controllers. Table III summarizes seven new attack scenarios that were revealed through our evaluation. These scenarios span all three SDN control flow categories (symmetric, asymmetric, and intra-controller).

Unknown Attack Name	*Flow*	*Target*
Sequence and Data-Forge	ASY	Floodlight
Stats-Payload-Manipulation	SYM	Floodlight, ODL
Echo-Reply-Payload-Manipulation	SYM	ODL
Service-Unregistration	INT	ODL
Flow-Rule-Obstruction	INT	ONOS
Host-Tracking-Neutralization	INT	ONOS
Link-Discovery-Neutralization	INT	Floodlight

TABLE III: Unknown attack case classification: ASY (Asymmetric), SYM (Symmetric), and INT (Intra-controller) flows.

1) Asymmetric Control Flows: In this scenario, a previously unknown asymmetric control flow attack involves the PACKET_IN message and the Floodlight controller.

Sequence and Data-Forge Attack: In the implementation of the Floodlight controller, when PACKET_IN messages arrive at the controller, it sequentially delivers the messages to a set of applications that have registered callbacks. Moreover, any application that receives the messages can `get`, `insert`, and even `remove` the payload within a message. Thus, the combination of these two features can be misused by a malicious or buggy application (e.g., delivering crafted payloads). Furthermore, this problem can result in the network entering an unstable state. The following step by step procedure is used to find an unknown attack.

1) On the initialization of DELTA, each subagent connects to the agent manager (AM). Then, the AM displays the initial commands, which we can choose. We

select 'Finding Unknown attack'. The AM requests a target control flow scenario code, and then selects an asymmetric control flow.

2) We select the target asymmetric control flow messages to randomize. In this case, the PACKET_IN message is selected, and the AM notifies the target message type to each fuzz module in the application and channel agent.
3) As the host agents communicate with each other, the fuzz modules randomize the sequence and the input values of messages matching the target message type.
4) Whenever a fuzz cycle is completed, the AM checks each case for violations of the criteria mentioned in Section V-B. If violations are found, the AM retrieves the randomized inputs from each fuzz module, and saves them in the log file.

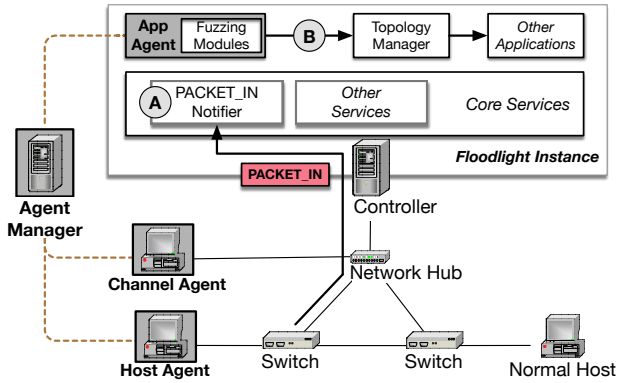

Fig. 8: Fuzz points of the Sequence and Data-Forge attacks

Using the control-flow fuzzer and the value fuzzer in the application agent, Figure 8 illustrates the attack scenario, highlighting with the points where the fuzzing modules randomize. Specifically, the control-flow fuzzer randomizes the delivery sequence of PACKET_IN messages (A in Figure 8), and the value fuzzer randomizes the message payloads (B in Figure 8). When the fuzz modules change the sequence and remove all payload bytes in a PACKET_IN message, DELTA discovers the vulnerability. Due to the removal of the payload, the `Topology Manager` (in Figure 8) is unable to receive the original payload and thus causes an exception error (e.g., NULL pointer exception). As a result, the switch that sends the PACKET_IN message is disconnected because the controller has no exception-handling mechanism. Since the switch disconnection is one of the criteria that determines whether this finding is an unknown attack, the AM determines that this case is an unknown attack scenario.

Based on the log file generated by the result analyzer in the AM, we re-examine the *unknown* case. Figure 9 illustrates the output of the controller's console during this analysis process. Initially, the application agent is located at the end of the sequence (in the 'Before' column of Figure 9). However, after modifying the sequence, the application agent is moved to the first entry of the 'After' column in Figure 9.

Finally, the controller shows a NULL pointer exception because the Topology Manager cannot properly handle a PACKET_IN message, as the application agent removes the

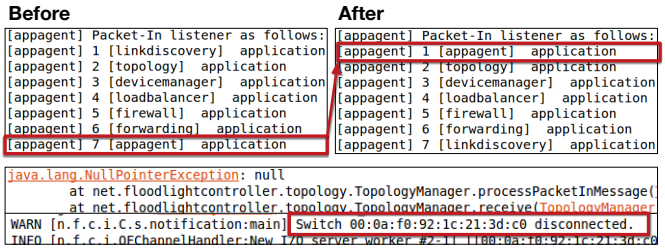

Fig. 9: Results of the Sequence and Data-Forge attack experiment

payload from the message, and then the switch that sent the PACKET_IN message is subsequently disconnected (i.e., the switch disconnection case in the criteria).

2) Symmetric Control Flows: Unlike the previous experiment, this experiment involves symmetric control flows and presents two new attack scenarios. These cases are detected by the control channel fuzz module, using randomizing input values.

Stats-Payload-Manipulation Attack: As mentioned in Section II-B, the STATS_REQUEST and STATS_RESPONSE messages are the representative messages for symmetric control flows. If an application wants to know specific flow statistics, the controller sends a STATS_REQUEST message to solicit switch status information, then the switch responds to the controller with the STATS_RESPONSE message.

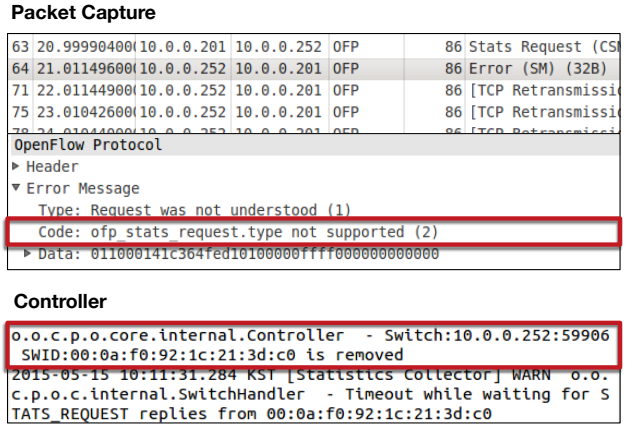

Fig. 10: Results of the Stats-Payload-Manipulation attack experiment

In this case, the DELTA operator first targets symmetric control flows. Then, the value fuzzer in the channel agent randomizes control messages passing through the control channel. Technically, when the fuzzing module modifies the type of STATS_REQUEST message to an undefined value (before fuzzing: flow stats, after fuzzing: `undefined`), the Agent Manager notices the switch disconnection matched to our criteria.

Figure 10 shows the results of the Stats-Payload-Manipulation attack. When the value fuzzer changes the type of the STATS_REQUEST message to a randomized value, the switch sends an error message (see Packet Capture in Figure

10) to the controller, and the switch disconnects from the controller (see Controller in Figure 10), which violates the switch-disconnection criterion.

Echo-Reply-Payload-Manipulation Attack: In another experiment involving symmetric control flows, the ECHO_REQUEST and ECHO_REPLY messages are popularly used in OpenFlow to exchange information about latency, bandwidth, and liveness on connected switches. If the controller does not receive a reply to the ECHO_REQUEST in time, it assumes that the switch is disconnected.

The operator first selects the symmetric control flows as the target flow type. Then, the AM randomly picks the ECHO_REPLY message type, and the value fuzzer in the channel agent starts to randomize the message passing through the control channel. When the fuzz module in the channel agent randomizes the length field of the ECHO_REPLY message as an undefined value (before: 8, after fuzzing: `0`), the switch disconnection event is triggered in the controller.

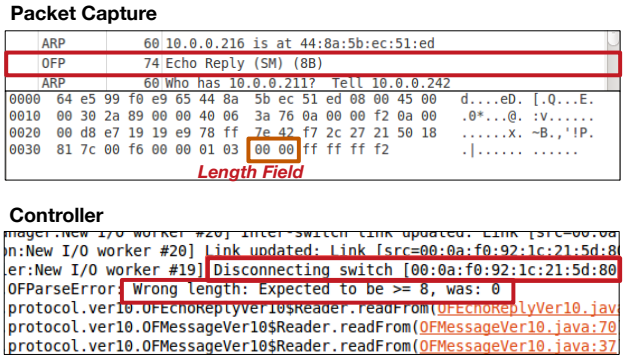

Fig. 11: Results of the Echo-Reply-Payload-Manipulation attack experiment

From the log information, we try to reproduce this attack case. Figure 11 shows the results of the Echo-Reply-Payload-Manipulation attack. When the value fuzzer changes the length field of the ECHO_REPLY message to 0 value (Packet Capture in Figure 11), the controller causes the exception to parse wrong length value of the message. Finally, the switch is disconnected from the controller.

3) Intra-Controller Control Flows: In the case of intra-controller control flows, the fuzz modules in the application agent have a significant role, and the channel agent is not involved. Many services provided by each controller, and these services form the targets. Next, we consider the following four unknown attack cases.

Service-Unregistration Attack: OpenDaylight provides a substantial diversity of network services, and OpenDaylight-hosted applications can dynamically register and use these services. For example, applications can freely register the `DataPacketService` to parse control messages arriving from the switch (e.g., PACKET_IN). While the application can register these services at initialization, the applications can dynamically change the services of other applications without constraint, and potentially with malicious intent.

During one experiment, the value fuzzer in the application agent found that it is possible to unregister certain services from other applications resulting in a significant disruption of network connectivity. For this experiment, a DELTA operator targets intra-controller control flows and fuzzes only input values. The value fuzzer chooses the `DependencyManager`, one of the available services to fuzz. While fuzzing input parameters, DELTA will try to unregister all services of `ArpHandler` which manage ARP packets. Ultimately, the connection between hosts is disconnected. Since this fuzz value causes the disconnection of hosts, the AM determines this case as a newly found attack scenario.

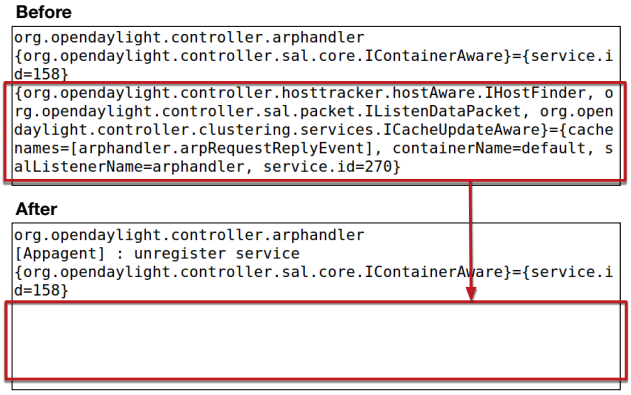

Fig. 12: Results of the Service-Unregistration attack experiment

Based on the log file, we can backtrack this attack scenario. As shown in Figure 12, the `ArpHandler` initially registered three kinds of services: IHostFinder, IListenDataPacket, and ICacheUpdateAware (Before in Figure 12). After the fuzzing modules unregister the services, the network loses its functionality, since ARP packets play a critical role as during the initiation of network communications (After in Figure 12). Therefore, two hosts that are connected to the switch cannot communicate with each other (i.e., criterion (vi): inter-host communication disconnection).

Flow-Rule-Obstruction Attack: In the implementation of ONOS, some applications may have configuration properties. For example, if an application declares a specific variable as a configuration property, the network administrator can change the variable dynamically. In addition to manually changing the properties, ONOS provides `ComponentConfigService`, which tracks and changes configuration properties for its applications. While the service allows applications to dynamically change the configuration of each component, it can also change unnecessary configurations.

A previously unknown attack scenario was discovered by targeting DELTA to the intra-controller flows. The value fuzzer in the application agent chooses ComponentConfigService among available services for randomizing input values. When the value fuzzer randomizes certain properties of `ReactiveForwarding`, the default application to send flow rules to the switch, the AM detects noticeable performance degradation of the switch. More specifically, the fuzzing module randomizes the `Packet_Out_Only` property of the ReactiveForwarding service (default: false, after fuzzing: true),

and the ReactiveForwarding service sends no FLOW_MOD messages to the switch.

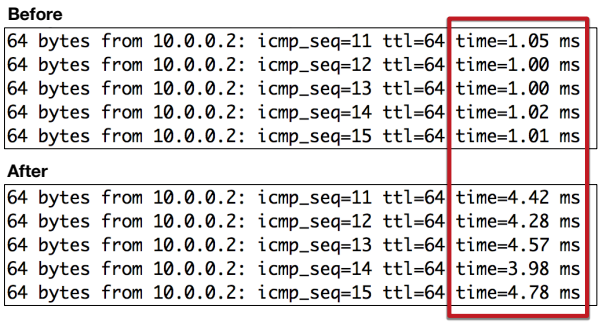

Fig. 13: Results of the Flow-Rule-Obstruction attack experiment

With the log file, we can verify the feasibility of this attack. Figure 13 shows the difference of the latencies before and after the attack. Since the ReactiveForwarding service does not send FLOW_MOD messages to the switch, every new flow arriving at the switch keeps generating PACKET_IN messages to the controller. Thus, the average of latencies becomes slower (about 4 ms in Figure 13 bottom) than the average before the attack (about 1 ms in Figure 13 top) as the workload of the controller increases (i.e., criterion (v): switch performance downgrade).

Host-Tracking-Neutralization Attack: ONOS keeps track of the location of each end-host connected to switches through the `HostLocationProvider`, which maintains host-related information (e.g., an IP address, a MAC address, a VLAN ID, end a connected port). Thus, if an end-host attaches to a switch and the service notices and updates the information of the end-host. As mentioned in the previous unknown attack scenario, ComponentConfigService can also change some configuration properties belonging to the HostLocationProvider service.

An operator can aim DELTA at the intra-controller flows for input value fuzzing (not flow sequence), then the ComponentConfigService service is selected by the value fuzzer in the application agent for input-value randomization. While the value fuzzer works, the controller receives error messages from the switch. Since the switch sending error messages to the controller matches one of the seven vulnerability detection criteria, the AM logs information that the fuzzing module randomized the `hostRemovalEnabled` property of the HostLocationProvider (default: true, after fuzzing: false). This change effectively prevents the tracking of end-host locations. For example, if a host is disconnected from the switch, the controller does not detect this disconnection.

To verify this unknown attack scenario, we analyzed the log information and backtracked the attack. Figure 14 shows the outputs from a packet capture tool [6] in the channel agent. The channel agent senses the error messages from the switch, which means that the controller for the flow rules is not available due to the invalid host. However, although the communication ends, error messages are sent to the controller every 10 seconds until the controller shuts down (i.e., criterion (vii): error-packet generation).

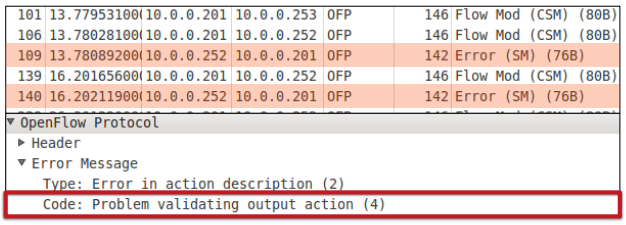

Fig. 14: Results of the Host-Tracking-Neutralization attack experiment

Link-Discovery-Neutralization Attack: Floodlight also provides diverse network services in the controller core for use by applications. Among these services, the `LinkDiscoveryService` offers a way of managing the link information by sending LLDP packet to other applications. For example, an application can read what link is connected to a specific switch, or send LLDP packets to other switches using this service.

We found that an application can prevent the controller from sending LLDP packets to all switches that are connected to the controller. This misleads the controller about tracking the link information. For the discovery, an operator selects intra-controller control flows as the target to be manipulated by the value fuzzer module in the application agent (not in the channel agent). The value fuzzer module feeds all switch information to an API provided by the LinkDiscoverService, which suppresses the sending of LLDP packets.

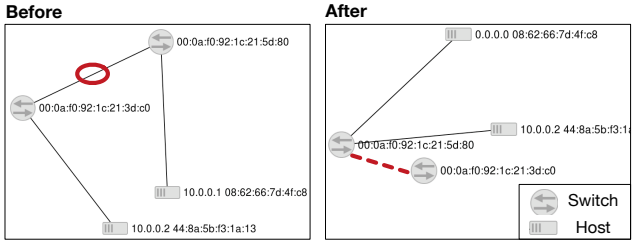

Fig. 15: Results of the Link-Discovery-Neutralization attack experiment. A red circle (before) represents a live link between two switches, and a red dotted line (after) represents a failed link.

As a result of this attack, the controller is forced to misinterpret the link-state information. Using a post-mortem analysis of the log information, we can reproduce this attack scenario to check if this attack really violates the criteria (i.e., criterion (iii) internal-storage poisoning). As shown in Figure 15, the controller web UI displays the correct network topology information (Before in Figure 15). However, after the attack is conducted, the topology information is changed, although the real topology as not been altered (After in Figure 15).

B. Use Case 2: Reproducing Known Attacks

Since the procedures and outputs of known attack scenarios are pre-specified, each agent needs to follow the steps and sequences of those scenarios with the pre-defined parameters. In the case of reproducing the known attack scenarios, we will illustrate two example cases: Man-In-The-Middle attack and Application Eviction attack.

Man-In-The-Middle Attack: As presented in Section IV, most commercial switches do not support the encryption of control messages between the control plane and the data plane (i.e., plain TCP channel) due to the performance issue. Thus, an attacker can freely sniff control messages and manipulate them, which can result in serious problems (e.g., network failures). In the implementation of the channel agent, we embeded the rule-modification function that can change an original OpenFlow action to a crafted action in a FLOW_MOD message, which is used to enforce a flow rule to a switch, into the attack simulator.

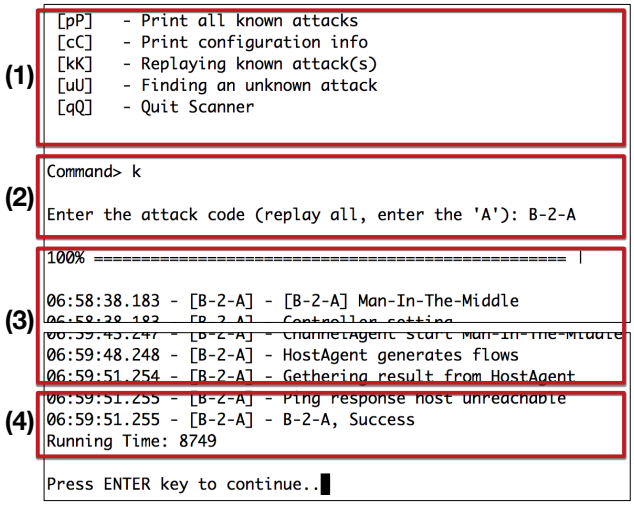

Fig. 16: Steps of the MITM attack experiment

The following four steps elaborate the procedure of this attack conducted against the Floodlight controller presented in Figure 16. (1) Upon the initialization of DELTA, each agent connects to the agent manager (AM). Then, the AM first shows the initial available commands. Here, we selected 'Replaying known attacks'. (2) The AM requests an attack scenario code, or an operator can select all. Here, we selected attack code 'B-2-A' denoting the MITM attack, whose code is determined based on the classification presented in [35]. (3) The AM organizes the actions of each agent based on the steps of the attack. During the attack, the AM also shows the attack process. Internally, the channel agent starts to intercept FLOW_MOD messages, and the attack simulator in the channel agent changes the action field of the messages defined in the configuration file. (4) The AM retrieves the output of the attack from the agents and indicates on the screen whether the attack was successful or failed.

To verify if the attack process is successful, we monitor packets on the control channel and observe the message changes through the Floodlight Web UI. Figure 17 shows the results of the attack. While the output port was 15 at the beginning (see Packet Capture (Before) in Figure 17), the output port has been changed to 19 (see Packet Capture (After) in Figure 17). The crafted FLOW_MOD messages have been installed into the switch (Figure 17 bottom). Thus, flows would be redirected to a wrong port after the attack.

Application Eviction Attack: Most controllers adopt a mechanism that can allow users to dynamically load and

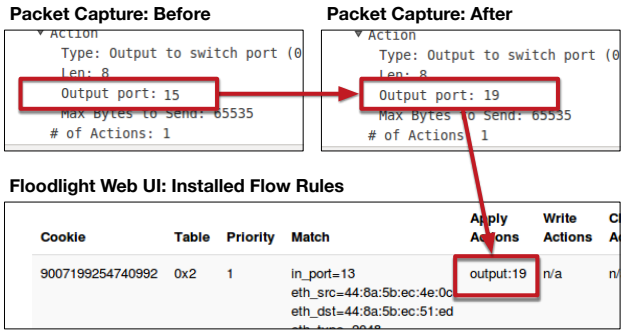

Fig. 17: Results of the MITM attack experiment

unload an application running on the controller (briefly mentioned in Section IV). However, due to no restriction on using the mechanism, an application can arbitrarily unload other applications. Here, we demonstrate an attack against the commercial Brocade Vyatta SDN controller [3], which is based on OpenDaylight.

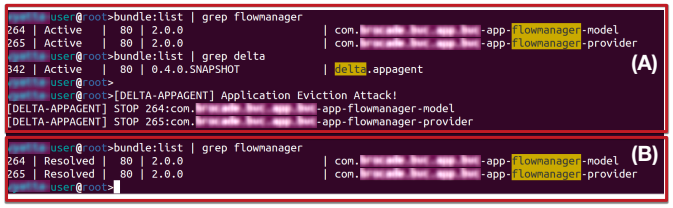

Fig. 18: Results of the Application Eviction attack experiment

Once the target controller has been initialized, as shown in Figure 18 (A), the application agent and the target application to evict are up and running (both are in ACTIVE state). Here, we attempt to evict the *flowmanager* application, which plays a critical role in managing flow rules on the switches. Then, once the target is confirmed, the agent executes the attack to stop the target application. As a result, one can see that the *flowmanager* application is no longer in an ACTIVE state after the attack (Figure 18 (B)).

From the attack case, we can consider that many commercial and open source controllers can be tested by DELTA. Thus, we are planning to test the vulnerabilities of those controllers in the near future.

C. Performance

1) Finding Unknown Attacks: For finding unknown attack cases, DELTA serially executes fuzz modules in each agent. Upon completion of each fuzz test cycle, the analyzer in AM checks if the attack was successful. Table IV shows the amount of time taken to complete one fuzz test cycle. The asymmetric control flow takes the longest average time, because the case uses all fuzzing modules in the application and the channel agent.

2) Reproducing Known Attacks: Table V describes the execution time for reproducing each known attack scenario. In the case of ONOS, the *Event-Listener-Unsubscription* attack is not possible since ONOS does not provide APIs to access

Control Flow Type	*Average Running Time*
Asymmetric Control Flow	82.5 sec
Symmetric Control Flow	80.4 sec
Intra-controller Control Flow	75.2 sec

TABLE IV: Finding unknown attack microbenchmark

event listeners. In the case of Floodlight, it is not possible to reproduce the *Application-Eviction* attack because the controller does not support a dynamic mechanism that loads/unloads other applications. Besides those attack scenarios, we excluded the *Switch Table Flooding* attack from the total execution time since it takes more than 90 minutes to fill up 2GB of memory of a Floodlight controller [11].

As shown in Table V, most of the attacks can be reproduced within a minute. The *System-Command-Execution* attack (NF-1) shows the shortest execution time (less than a second). It takes only five minutes to reproduce all of the aforementioned attacks, with the exception of the *Switch-Table-Flooding* attack. These results serve to underscore the flexibility and usability of DELTA. Specifically, it enables network operators to efficiently reproduce attacks and easily conduct systematic security assessments of OpenFlow networks. In the absence of such a framework, it would be significantly more challenging to create test environments for varying and complex SDN attack scenarios.

Attack Code	*Controller*		
	ONOS	*OpenDaylight*	*Floodlight*
SF-1	-	-	5400 sec
SF-2	16.09 sec	16.34 sec	15.96 sec
SF-3	21.5 sec	12.33 sec	11.99 sec
SF-4	28.1 sec	19.27 sec	18.6 sec
AF-1	12.55 sec	8.47 sec	3.13 sec
AF-2	3.38 sec	8.12 sec	3.21 sec
AF-3	12.59 sec	17.79 sec	11.96 sec
AF-4	43.65 sec	23.28 sec	43.2 sec
AF-5	40.43 sec	40.24 sec	20.35 sec
AF-6	20.52 sec	20.25 sec	20.2 sec
AF-7	20.6 sec	20.32 sec	20.17 sec
AF-8	33.62 sec	33.18 sec	33.14 sec
AF-9	17.8 sec	17.19 sec	7.88 sec
CF-1	2.6 sec	3.14 sec	2.14 sec
CF-2	22.57 sec	13.33 sec	N/A
CF-3	N/A	13.22 sec	13.11 sec
NF-1	0.028 sec	0.095 sec	0.127 sec
NF-2	23.54 sec	23.2 sec	23.16 sec
NF-3	23.43 sec	23.36 sec	23.35 sec
NF-4	3.39 sec	4.86 sec	3.17 sec
Total	346.38 sec	317.98 sec	274.84 sec

TABLE V: Reproducing known attacks microbenchmark: Attack Code is referenced by Table I

VIII. Limitation and Discussion

Like other research work, our system also has some limitations. First, some testing cases require installing a specified agent (i.e., Application Agent) to an SDN controller. For example, to reproduce the Internal Storage Misuse attack in each controller requires the installation of our Agent Manager for each controller. This limitation may slow the adaptation of our tool to diverse control platforms. However, currently our framework covers most well-known open source controllers, and we will provide an interface module for other control platforms to easily integrate or extend our framework.

Second, some operations require human involvement. We have tried to minimize the amount of human interaction, and our framework can be operated with simple configurations. However, some cases, such as adding new attack scenarios, require manual modifications to some parts of the framework. This situation happens when our framework discovers a new type of attack through the fuzzing module. In this case, we can understand an attack scenario through the log information, but this may require a new way to handle SDN control flows or messages. We will revise this in the near future to automatically handle all (or most) operations.

IX. Conclusion

This paper describes an important first step toward developing a systematic methodology for automatically exploring the critical data flow exchanges that occur among SDN components in search of known and potentially unknown vulnerabilities. To our knowledge, this framework, called DELTA, represents the first and only SDN-focused security assessment tool available today. It has been designed for OpenFlow-enabled networks and has been extended to work with the most popular OpenFlow controllers currently available. We also presented a generalizable SDN-specific blackbox fuzz testing algorithm that is integrated into DELTA. This fuzz testing algorithm enables the operator to conduct in-depth testing of the data input handling logic of a range of OpenFlow component interfaces. We demonstrate the effectiveness of this fuzz testing algorithm by presenting seven previously unknown attack scenarios that were detected by this tool.

Acknowledgment

This material includes work supported by the National Science Foundation under Grant No. 1547206. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work is also supported by Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korean government (MSIP) (No. B0126-16-1026, Development of Core Technologies for SDN-based Moving Target Defense).

References

[1] K. Benton, L. J. Camp, and C. Small. Openflow vulnerability assessment. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN'13)*. ACM, 2013.

[2] 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.

[3] Brocade. Brocade SDN Controller, 2016. http://www.brocade.com/en/products-services/software-networking/sdn-controllers-applications/sdn-controller.html/.

[4] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *Proc. of the IEEE Symposium on Security and Privacy*, May 2015.

[5] B. Chandrasekaran and T. Benson. Tolerating sdn application failures with legosdn. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets'14)*. ACM, 2014.

[6] G. Combs et al. Wireshark-network protocol analyzer. *Version 0.99*, 5, 2008.

[7] W. Cui, M. Peinado, H. J. Wang, and M. E. Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 252–266. IEEE, 2007.

[8] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 254–265. ACM, 2011.

[9] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann. Sphinx: Detecting security attacks in software-defined networks. In *NDSS*, 2015.

[10] J. M. Dover. A denial of service attack against the open floodlight sdn controller, 2013.

[11] J. M. Dover. A switch table vulnerability in the open floodlight sdn controller, 2014.

[12] O. N. Foundation. Security Working Group. https://www.opennetworking.org/technical-communities/areas/services.

[13] Fyodor. Nmap security scanner. http://www.nmap.org.

[14] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[15] S. Hocevar. zzuf. https://github.com/samhocevar/zzuf.

[16] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 15–26. ACM, 2013.

[17] 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.

[18] A. D. Householder and J. M. Foote. Probability-based parameter selection for black-box fuzz testing. Technical report, 2012. CERT Technical Report.

[19] HP. HP SDN App Store. https://marketplace.saas.hpe.com/sdn.

[20] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[21] D. Kotani and Y. Okabe. A packet-in message filtering mechanism for protection of control plane in openflow networks. In *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '14, pages 29–40, New York, NY, USA, 2014. ACM.

[22] 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*, 103(1):14–76, 2015.

[23] 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.

[24] D. Maynor. *Metasploit toolkit for penetration testing, exploit development, and vulnerability research*. Elsevier, 2011.

[25] J. Medved, R. Varga, A. Tkacik, and K. Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *2014 IEEE 15th International Symposium on*, pages 1–6. IEEE, 2014.

[26] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of ACM*, 1990.

[27] B. S. Networks. Floodlight. http://www.projectfloodlight.org/floodlight/.

[28] Open Networking Foundation. https://www.opennetworking.org/.

[29] OpenFlow. OpenFlow Specification version 1.0.0. Technical report, 2009. http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf.

[30] OpenFlow. OpenFlow Specification version 1.3.0. Technical report, 2011. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

[31] 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 (HotSDN'12)*, 2012.

[32] C. Röpke and T. Holz. Sdn rootkits: Subverting network operating systems of software-defined networks. In *Research in Attacks, Intrusions, and Defenses*, pages 339–356. Springer, 2015.

[33] S. Lee, and C. Yoon, and S. Shin and S. Scott-Hayward. DELTA: SDN SECURITY EVALUATION FRAMEWORK. http://opensourcesdn.org/projects/project-delta-sdn-security-evaluation-framework.

[34] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, et al. Troubleshooting blackbox sdn control software with minimal causal sequences. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 395–406. ACM, 2014.

[35] SDNSecurity.org. SDN Security Vulnerabilities Genome Project. http://sdnsecurity.org/project_SDN-Security-Vulnerbility-attack-list.html.

[36] T. N. Security. Nessus. http://www.tenable.com/products/nessus-vulnerability-scanner.

[37] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky. Advanced study of sdn/openflow controllers. In *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, CEE-SECR '13, pages 1:1–1:6, New York, NY, USA, 2013. ACM.

[38] S. Shin and G. Gu. Attacking software-defined networks: A first feasibility study (short paper). In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*, August 2013.

[39] 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.

[40] S. Shin, V. Yegneswaran, P. Porras, and G. Gu. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*, November 2013.

[41] A. Takanen, J. D. Demott, and C. Miller. Fuzzing for Software Security Testing and Quality Assurance. http://www.mcafee.com/us/products/network-security-platform.aspx.

[42] A. Turner and M. Bing. Tcpreplay: Pcap editing and replay tools for* nix. *online], http://tcpreplay. sourceforge. net*, 2005.

[43] X. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. Choi. Packet vaccine: Black-box exploit detection and signature generation. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 37–46. ACM, 2006.

[44] J. Yao, Z. Wang, X. Yin, X. Shiyz, and J. Wu. Formal modeling and systematic black-box testing of sdn data plane. In *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*, pages 179–190. IEEE, 2014.

[45] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with difane. *ACM SIGCOMM Computer Communication Review*, 40(4):351–362, 2010.

[46] M. Zalewski. American Fuzzy Lop. lcamtuf.coredump.cx/afl/.