Computers & Security 91 (2020) 101720

Contents lists available at ScienceDirect

Computers & Security

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

A comprehensive security assessment framework for software-defined networks

Seungsoo Lee[a], Jinwoo Kim[b], Seungwon Woo[c], Changhoon Yoon[d], Sandra Scott-Hayward[f], Vinod Yegneswaran[e], Phillip Porras[e], Seungwon Shin[a,b,*]

[a] Graduate School of Information Security, School of Computing, KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon 34141, Republic of Korea
[b] School of Electrical Engineering, KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon 34141, Republic of Korea
[c] ETRI, 218 Gajeong-ro, Yuseong-gu, Daejeon 34129, Republic of Korea
[d] S2W Lab, 240 Pangyoyeok-ro, Bundang-gu, Seongnam-si, Republic of Korea
[e] Computer Science Laboratory, SRI International, Menlo Park, CA, USA
[f] Centre for Secure Information Technologies, Queen's University Belfast, Belfast, U.K.

ARTICLE INFO

Article history:
Received 5 August 2019
Revised 2 December 2019
Accepted 16 January 2020
Available online 18 January 2020

Keywords:
Software-Defined Networking
Security
Network security
Penetration testing

ABSTRACT

As Software-Defined Networking (SDN) is getting popular, its security issue is being magnified as a new controversy, and this trend can be found from recent studies of presenting possible security vulnerabilities in SDN. Understanding the attack surface of SDN is necessary, and it is the starting point to make it more secure. However, most existing studies depend on empirical methods in different environments, and thus they have stopped short of converging on a systematic methodology or developing automated systems to rigorously test for security flaws in SDNs. Therefore, we need to disclose any possible attack scenarios in diverse SDN environments and examine how these attacks operate in those environments. Inspired by the necessity for disclosing the vulnerabilities in diverse SDN operating scenarios, we suggest an SDN penetration tool, **DELTA**, to regenerate known attack scenarios in diverse test cases. Furthermore, DELTA can even provide a chance of discovering unknown security problems in SDN by employing a fuzzing module. In our evaluation, DELTA successfully reproduced 26 known attack scenarios, across diverse SDN controller environments, and also discovered 9 novel SDN application mislead attacks.

1. Introduction

Security has been a subject of controversy in many newly emerged networked systems, such as peer-to-peer networks and cloud networks. After their appearance, researchers and practitioners have examined their security issues from var- ious angles to verify their safeness, and this process makes them more secure so that they can be adapted in a real-world system. Software-Defined Networking (SDN), which manages a network in a centralized way, is a recently proposed networking technology, and now it is endorsed by both industry and academia. As SDN technology is getting popular, its security problem is being at issue, and thus researchers are investigating its security issues as they have conducted in other networked systems (Benton et al., 2013; Hong et al., 2015; Kreutz et al., 2015; 2013; Porras et al., 2012; Shin and Gu, 2013).

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.

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 (Fyodor, 2020; Security, 2020), DELTA represents the first security assessment tool for SDN environments. Furthermore, we enhanced our tool with a specialized fuzzing module (Miller et al., 1990) to exploit opportunities for discovering unknown security flaws in SDNs.

* Corresponding author.
E-mail address: seungwon.shin@gmail.com (S. Shin).

https://doi.org/10.1016/j.cose.2020.101720
0167-4048/

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 (OpenFlow, 2009), we categorize operational flows into five categories (see Section 2). 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 multiple perspectives. 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 26 known SDN-related attack scenarios targeting several well-known SDN elements, such as ONOS (Berde et al., 2014), OpenDaylight (ODL) (Medved et al., 2014), Floodlight (Big Switch Networks, 2020; NTT Communications, 2020), and the commercial Brocade Vyatta SDN controller (Brocade, 2016). In addition, DELTA was able to discover 9 new attack scenarios by applying control flow fuzzing techniques.

The results of our analysis have contributed to the Open Networking Foundation (ONF) (Open Networking Foundation, 2020) technical reports defining best practices for securing SDN environments. New attack scenarios exposed by DELTA have also been reported to the ONF and, most recently, DELTA has been adopted by ONOS for integration in the development test suite to profile the ongoing security of the controller (Secci et al., 2017). Furthermore, we have shared our findings gained from DELTA with the attendees in Black Hat USA that is one of the influential security conferences in the world (BLACK-HAT-USA-2016, 2020; BLACK-HAT-USA-2017, 2020; BLACK-HAT-USA-2018, 2020).

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 26 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 unknown attack scenarios. To conduct an efficient fuzz-test, we present an operational state diagram that describes typical state transitions in OpenFlow protocol. Based on the derived state diagram, we also propose a fuzzing algorithm that randomizes OpenFlow control flow sequences. Through our evaluation, we verified that this technique found 9 new attack cases.
- The demonstration of flexibility of system design by evaluating it against four popular open-source SDN controllers and one commercial SDN controller, Brocade Vyatta controller.

The remainder of this paper is structured as follows; In Section 2, the background and motivation for this work is outlined. Related work is discussed in Section 3. The vulnerabilities in SDN flows are described in Section 4. Section 5 introduces the system design. The implementation and evaluation of DELTA are presented in Sections 6 and 7, respectively. And the limitation and discussion are described in 8. Finally, in Section 9, the conclusions are summarized.

2. Background and motivation

2.1. 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, SDN presents a new 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. The key features of SDN are high-level network abstraction providing a global view of the network, and programmability.

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, and is widely deployed (Hong et al., 2013; Jain et al., 2013). Although, OpenFlow does not cover all parts of SDN, it reflects the most important part (i.e., an interface between the control plane 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. As a protocol, OpenFlow has been rapidly evolving, most recently with the release of OpenFlow version 1.5 protocol specification (OpenFlow, 2014). However, most production deployments rely on OpenFlow 1.0 and 1.3 because most network devices (e.g., switches and routers) still only support them.

2.2. SDN control flows

In addition to the OpenFlow protocol, there are many types of control messages to operate SDN, and in this work, we call it *SDN control flows*. The operations of SDN can be classified into five types of control flows as shown in Fig. 1: (1) symmetric, (2) asymmetric, (3) intra-controller, (4) inter-controller, and (5) admin 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). Example (1) in Fig. 1 presents an instance illustrating this operation. Consider a load balancer application 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 receives the event, it sends

[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.

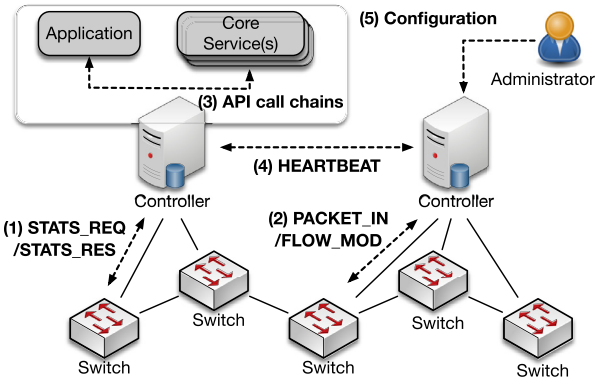


Fig. 1. Examples of the control flows in SDN: (1) symmetric, (2) asymmetric, (3) intra-controller, (4) inter-controller, and (5) admin.

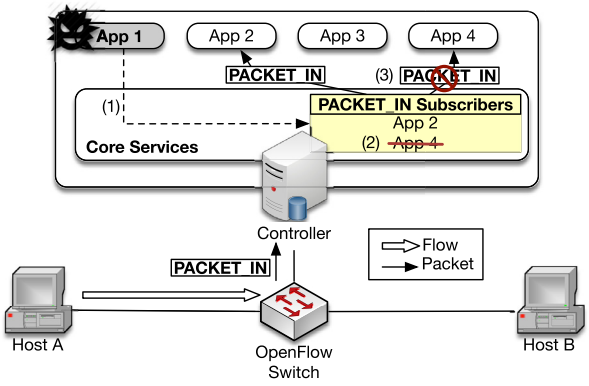


Fig. 2. Event Listener Unsubscription attack.

the STATS_REQUEST message to the switch through an OpenFlow message. Then, the switch packs its statistics information in the STATS_RESPONSE message and returns it to the controller. Finally, the statistics core service returns the received statistics to the load balancer application.

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 arrival and inserting flow policy). Example (2) in Fig. 1 represents 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. 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). 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 or core services running on a controller as shown in Example (3) in Fig. 1 (i.e., control plane). When applications interact with one another or use the core 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.

Inter-controller control flow operations: To resolve a single point of failure problem in one SDN controller, distributed SDN controller designs have been suggested (Berde et al., 2014; Oktian et al., 2017). The distributed computing is a key to guarantee high-performance and fault tolerance in large-scale networks. Such distributed notion has also emerged as a critical concept in many customers such as data center and telecommunication operator who need a guarantee of availability (ONOS, 2020). In the distributed SDN controllers, it is necessary to maintain the same states among the individual controller instances organizing a SDN cluster. To do this, the distributed SDN controller utilizes a west-east protocol that provides keepalive messages (Example (4) in Fig. 1), a state synchronization, and leader node decisions among the controller instances.

Admin control flow operations: By providing external interfaces (e.g., RESTful services), SDN controllers allow network administrators to manage complicated traffic within a centralized network view manually as well. As shown Example (5) in Fig. 1, they can manually configure network flow-handling policies (i.e., flow rules) for each network device through the interfaces. Besides, the administrators can get the network (device) states by leveraging the key features of SDN, which are high-level network abstraction and programmability. With this feature, it enables flow management to be performed by remote location, offering much greater diversity.

2.3. Motivating example

Fig. 2 illustrates how a malicious application could render a benign application incapable of receiving any of the necessary control messages from a switch. In this example, we assume that a network operator has downloaded and installed a malicious SDN application because an SDN application ecosystem is similar to Android in a sense that anyone can develop and distribute applications using open APIs.

First, a malicious application (App 1) accesses the list identifying which application receives the PACKET_IN control message (the most important control message in OpenFlow-based SDNs), 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). If App 4 is a security-sensitive application such as firewall, it can cause unexpected network states because the application cannot make a decision based on the PACKET_IN messages.

This example shows that abusing the inter-controller control flows can remove the specific application that wishes to listen the PACKET_IN message without any constraints. And, this is a real working example (applicable to Floodlight (Big Switch Networks, 2020) and OpenDaylight (Medved et al., 2014) controllers), and it illustrates how a malicious application confuses a benign application by manipulating the intra-controller control flow operation. The reason why the malicious application can manipulate the other ones is that there is no mandatory permission system in SDN.

Considering this example, SDN-specific attack and vulnerability cases are not trivial anymore. Besides, most attack scenarios (will be described later) are based on SDN control flows, and it cannot be revealed by existing pen-testing frameworks, such as metasploit (Maynor, 2011) and nessus (Security, 2020) because those frameworks do not know how SDN works. We believe that this example

scenario clearly presents why we need a new pen-testing framework for SDN.

3. 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 (Benton et al., 2013; Kreutz et al., 2015) 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 (Kreutz et al., 2015). 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 (MITM) attacks and denial-of-service (DoS) attacks easier (Benton et al., 2013). Furthermore, some researchers have raised other issues, such as inter-application conflicts, access control, topology manipulation, and sharing relationships (Dhawan et al., 2015; Hong et al., 2015; Porras et al., 2012; Shin et al., 2014). Röpke et al. (Röpke and Holz, 2015) 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 (HP, 2020). Even without delivering malicious SDN applications to SDNs, Dover et al. have also shown that it is possible to launch DoS and spoofing attacks by exploiting the implementation vulnerability that exists in the switch management module of Floodlight (Dover, 2013; 2017).

Although there have been several studies on SDN vulnerabilities, contemporary controllers remain vulnerable to many of these attacks. Hence, we proposed a software framework that can simplify reproducibility and verification of diverse attack scenarios (Lee et al., 2017). Inspired by our effort, recently, several works for evaluating SDN security have been presented (Jero et al., 2020; Ujcich et al., 2017). However, while these works have mainly focused on manipulating OpenFlow messages by sniffing the control channel between the controller and the switch, DELTA provides comprehensive security test cases including the vulnerabilities of the application layer.

Vulnerability Detection Tools and Techniques: Traditional network security testing tools such as Metasploit (Maynor, 2011), Nessus (Security, 2020), and Nmap (Fyodor, 2020) 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 unsuitable for SDN networks. In a recent BlackHat briefing, the authors explored the SDN attack surface by systematically attacking each layer of the SDN stack, and then demonstrating some of the most critical attacks that directly affect the network availability and confidentiality (Hizver, 2020). This illustrates that SDN-specific security threats are complex and cannot be revealed by existing network security testing tools 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 (Miller et al., 1990; Takanen et al., 2020). 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 (Cha et al., 2015) and zzuf (Hocevar, 2020). 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 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 (Cui et al., 2007). Godefroid et al. present a grammar-based whitebox fuzz-testing approach inspired by symbolic execution and dynamic test generation (Godefroid et al., 2008). 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 (Scott et al., 2014). 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 approach for the SDN data plane (Yao et al., 2014). While this approach mainly focuses on the testing paths of SDN data planes, DELTA applies fuzzing functions to discover unknown security flaws across the SDN stack.

4. Vulnerabilities in SDN flows

This section explores how the SDN flow operations described in Section 2 are related to vulnerabilities that can harm SDN operations. Vulnerabilities related to the SDN control flows are discussed in Section 4.1 and the locations of vulnerabilities resulting from non-flow-related operations are described in Section 4.2, and from the vulnerability discussion we derive the seven vulnerability detection criteria, which are explained in Section 4.3.

Table 1 provides a high-level overview of the feasible SDN attack cases against each SDN controller (i.e., control plane). Here, we tested the control plane attacks against the four most prevalent and well-known SDN controllers (ONOS, OpenDaylight, Floodlight, and Ryu controllers).

4.1. SDN control flow operation vulnerabilities

Symmetric Control flow Vulnerabilities: For the control plane, Table 1 identifies eight symmetric control flow vulnerabilities. Two vulnerabilities raise in the presence of weak authentication during the handshake step between the controller and the switch as follows. First, the Floodlight controller classifies the identification of the connected switch according to a Data Plane ID (DPID). However, a 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 (i.e., SF-1). Next, as the Floodlight controller 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., SF-2). Such an attack could result in a controller shut down.

Failing to conform to specifications also incurs security problems in SDNs. In OpenFlow networks, the control and data plane components must exchange HELLO messages to initiate the connection. A handshake process must then be completed before the specified timeout and any incomplete controller-switch connection should be torn down to avoid an adversary taking over the open (incomplete) connection. Otherwise, the attacker can exploit the SDN controllers that allow the initiation of controller-switch connections without the exchange of HELLO messages (i.e., SF-3). In addition, each controller-switch connection should be reliably

Table 1
Summary of known SDN attack cases against control plane: O means the controller is vulnerable to this attack, and X means that it is not vulnerable.

Flow Type	*Attack Code*	*Attack Name*	*Controller*			
			ONOS	*OpenDaylight*	*Floodlight*	*Ryu*
Symmetric Flows	SF-1	Switch Identification Spoofing (Dover, 2017)	X	O	O	X
	SF-2	Switch Table Flooding (Dover, 2013)	X	X	O	X
	SF-3	Handshake without HELLO Message (Lee et al., 2020)	X	X	X	X
	SF-4	Redundant Main Connection Request (Lee et al., 2020)	X	X	X	X
	SF-5	TLS Connection Misuse (Lee et al., 2020)	X	X	X	X
	SF-6	Auxiliary Connection Mismatch (Lee et al., 2020)	X	X	X	X
	SF-7	Malformed Version Number (Lee et al., 2020)	O	O	O	O
	SF-8	Corrupted Control Message Type (SDNSecurity.org, 2020)	O	O	O	O
Asymmetric Flows	AF-1	Control Message Drop (SDNSecurity.org, 2020)	O	O	O	X
	AF-2	Control Message Infinite Loop (SDNSecurity.org, 2020)	O	O	O	O
	AF-3	Flow Rule Modification (SDNSecurity.org, 2020)	O	O	O	O
	AF-4	PACKET_IN Flooding (Kotani and Okabe, 2014; Shin and Gu, 2013; Shin et al., 2013)	O	O	O	O
	AF-5	Flow Rule Flooding (Curtis et al., 2011; Shin and Gu, 2013)	O	O	O	O
	AF-6	Switch Firmware Misuse (SDNSecurity.org, 2020)	O	O	O	X
	AF-7	Flow Table Clearance (SDNSecurity.org, 2020)	O	O	O	O
	AF-8	Eavesdrop (SDNSecurity.org, 2020)	O	O	O	O
	AF-9	Man-In-The-Middle (SDNSecurity.org, 2020)	O	O	O	X
	AF-10	Control Message before Connection (Lee et al., 2020)	X	X	X	X
	AF-11	Unflagged Flow Remove Notification (Lee et al., 2020)	O	X	X	X
Intra-Controller Flows	CF-1	Internal Storage Misuse (Shin et al., 2014)	O	O	O	X
	CF-2	Application Eviction (Shin et al., 2014)	O	O	X	O
	CF-3	Event Listener Unsubscription (Shin et al., 2014)	X	O	O	O
Non Flow Operations	NF-1	System Command Execution (Shin et al., 2014)	O	O	O	O
	NF-2	Memory Exhaustion (Shin et al., 2014)	O	O	O	O
	NF-3	CPU Exhaustion (Shin et al., 2014)	O	O	O	O
	NF-4	System Variable Manipulation (SDNSecurity.org, 2020)	X	O	O	X

maintained. However, an attacker could interrupt existing legitimate controller-switch connections by sending out additional connection requests in order to affect the network availability (i.e., SF-4).

For a secure connection between the controller and the switch, OpenFlow recommends the use of TLS. However, if this protection is not in place, it is possible for an attacker to intentionally attempt a failed TLS connection to gain an insecure TCP connection (i.e., SF-5). For the high availability, the OpenFlow provides two types of connection; main and auxiliary connections. Thus, if the main connection is set to TLS, the auxiliary one should also be TLS. Otherwise, data tampering and information disclosure are feasible at the controller through an auxiliary TCP connection. (i.e., SF-6). Besides, there could be some malformed control message attacks. For instance, the attacker manipulates the OpenFlow version value (i.e., SF-7) or the header type value (i.e., SF-8) in symmetric control messages with an invalid value in order to cause an inconsistency issue, which may result in a switch disconnection.

Asymmetric Control flow Vulnerabilities: Table 1 identifies 11 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. However, misbehaving or rogue applications can interfere with the order of applications in the list, and cause the application to drop the message (i.e., AF-1). Also, malicious applications can alternatively implement an infinite loop to prevent other applications from acting on the message (i.e., AF-2). Further, 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., AF-3).

Controllers and switches are vulnerable to performance degradation by malicious or erroneous applications. One such example is that 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 (i.e., AF-4). On the contrary to this, it is also possible for a malicious application to generate numerous FLOW_MOD messages to the flow table, which can lead the switch into an unpredictable state (i.e., AF-5). Also, by changing the rules, the malicious application can manipulate the flow table in the switch to make the switch performance be unstable (i.e., AF-6 and AF-7).

If the control messages between the controller and the switch are unencrypted, an attacker located between them can confuse the control plane. For example, the attacker can guess what topology is constructed by sniffing control messages in a passive manner (i.e., AF-8). Moreover, the attacker may also intercept the control message and then change some field values of the control messages with malicious intent (i.e., AF-9). In addition, it is feasible for the attacker to send a PACKET_IN message before a connection between the controller and the switch is established in order to install an improper flow rule to the switch (i.e., AF-10). For the flow management, the controller can set the removal notification flag when installing the flow rule on the switch so that they can recognize if the flow rule is expired or not. However, to abuse this feature, the attacker can send a fake flow removal notification to the controller so that the controller may have an erroneous flow management view from the switch (i.e., AF-11).

Intra-Controller Control flow Vulnerabilities: Table 1 identifies three intra-controller control flow vulnerabilities. Since 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., CF-1). In addition, by abusing those APIs, the malicious application can dynamically unload a security-sensitive application (e.g., firewall application)

without any constraint (i.e., CF-2). Also, the malicious application can prevent some applications which want to receive the control message from being notified of the control message (i.e., CF-3).

4.2. Non flow operation vulnerabilities

Table 1 identifies four non-flow operation vulnerabilities. Although SDN controllers have been referred to as 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. However, since most controllers employ the multi-threaded programming paradigm, the termination of the application can mislead the controller into shutdown (i.e., NF-1). 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., NF-2), or by unconstrained thread creation to exhaust available CPU cycles (i.e., NF-3). 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., NF-4).

4.3. Vulnerability detection criteria

Considering the impacts of the 26 vulnerabilities as described above, we derive the following seven vulnerability detection criteria: (*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. By leveraging those criteria, we judge whether an attack case is feasible against the target SDN environments when reproducing the known attacks. Moreover, in addition to the known attacks, it can also be used for discovering an unknown attack case, which we will detail in later.

5. System design

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

5.1. Design considerations

The attacks outlined in Section 4 are a few examples among the broader set of known SDN vulnerabilities. As possible attack scenarios increase, so too increases the challenge of determining how and where to address these threats among the various deployment-specific SDN instantiations. Today, the cost associated with conducting security testing within specific SDN network instances is high. SDN security testing is ad-hoc and cumbersome, as attack scenarios may arise uniquely from different SDN components, network configurations, and testing inputs. To help reduce this cost, we need a generalized SDN penetration testing framework that can automatically reproduce diverse attack scenarios as we have done in other areas (e.g., web security testing tool), and this framework should be easy to use.

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.

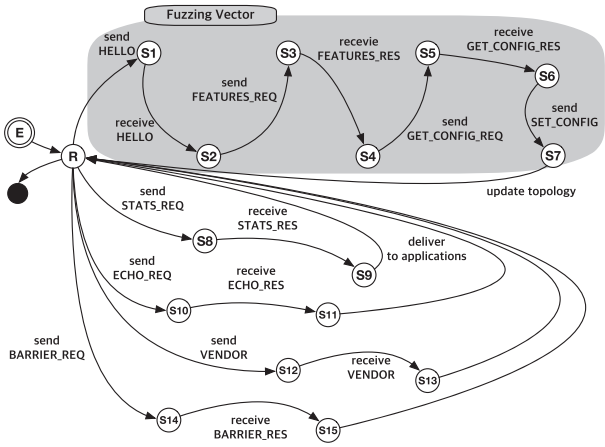


Fig. 3. A partial operational state diagram of typical SDN controller and a fuzzing vector example.

5.2. Blackbox fuzzing

In addition to the known attack scenarios reviewed in Table 1, a wider range of undiscovered attack scenarios against SDNs remain, which our framework can help operators to explore and discover. To identify unknown attack cases, we borrow the notion of *fuzz testing* developed in the context of legacy software and protocol testing. Fuzz testing allows 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 applied to both open source and proprietary SDN components and devices.

State Diagram of SDN Control Messages: 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, can be represented in an operational state diagram. Fig. 3 shows a partial state diagram of typical SDN controller, which specifically is relevant to the symmetric control flows. Although we only present the state diagram for OpenFlow v1.0, we have also analyzed OpenFlow v1.3 (OpenFlow, 2011) and it is a straightforward extension. The state diagram can be represented by a graph abstraction which has vertices and edges. Thus, we can formally define it as the following definition:

Definition 1. An operational state diagram is a directed graph G that consists of V which denotes a set of states (vertices) v, and E which denotes a set of transitions (edges) e.

In the state diagram (Fig. 3), label `E` is an end state and label `R` stands for a ready (initial) state to receive or send the control messages. Each edge's label designates the type of control message and the specific controller behavior that triggered the state transition. For example, as shown in Fig. 3, when a controller sends a HELLO message to a switch for a new connection in `R` state, the state of the controller moves to `S1`. In `S1`, the controller receives the

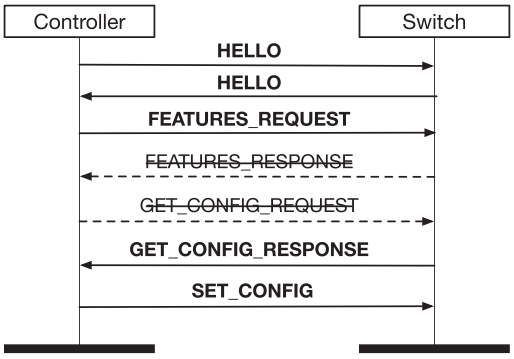


Fig. 4. Symmetric flow sequence randomization example.

HELLO message from the switch, causing a state transition to S2. If the handshake process with the switch is successful, the state arrives at S7 and the controller then updates the topology information. From this example, we can derive a formal definition of the transition edges:

Definition 2. A set of state transitions E denotes a set of *labeled* edges that represent a combination of a controller's behavior and type of control message.

In addition to the symmetric control flows, we derive other control flows-related states as well, such as the asymmetric, intra-controller, inter-controller, and admin control flows. This 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.

Based on the state diagram, we investigated (i) the sequence of control flows, presented in Section 2, to determine whether there are candidate control flows for randomization, and then examined the (ii) input values conveyed in each control flow.

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.

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 Fig. 3, the controller states from R to S7 represent the OpenFlow handshake process. 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. For example, when PACKET_IN messages are delivered to applications, it is difficult to detect state transitions within the state diagram 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 (Fuzzing Vector in Fig. 3) For example, as shown in Fig. 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 vulnerable to such protocol violations.

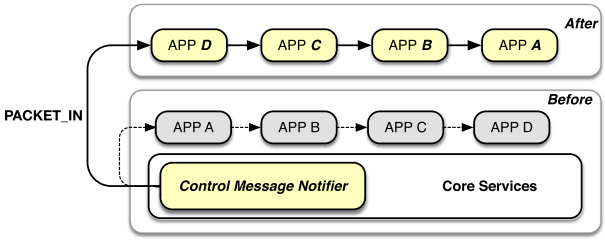


Fig. 5. Asymmetric flow sequence (series) randomization example.

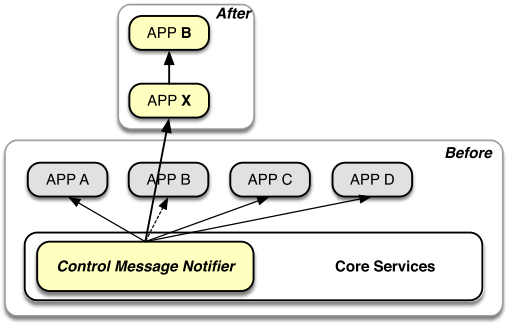


Fig. 6. Asymmetric flow sequence (parallel) randomization example.

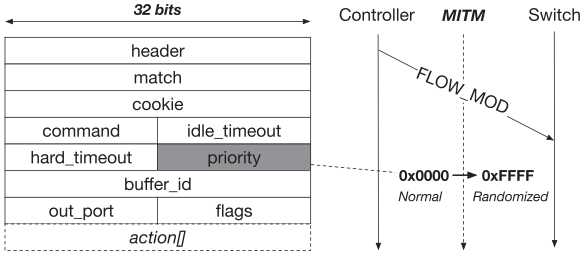


Fig. 7. Input value randomization example: left is a header format of FLOW_MOD message and right represents the flow sequence.

Such control flow manipulation can be also applied to the asymmetric flows. If a PACKET_IN message is sent to the controller by the network device, the controller sequentially delivers the message to the applications in a specific order. Fig. 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 → B → C → D in series). Here, we can change the control flow (i.e., shuffle the order of the applications) randomly at runtime as shown in Fig. 5 (After) and observe the system behavior.

In addition to the sequential asymmetric control message delivery mechanism, messages can be delivered to applications in parallel as shown in Fig. 6. For example, when the controller receives a PACKET_IN message, it can simultaneously deliver the asymmetric message to the applications. 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 (Fig. 6 (After)). Again, it is possible to randomize such sequences to observe the behavior.

Randomizing Input Values: The input values of a control flow can also be randomized. For example, we can select the FLOW_MOD message as shown in Fig. 7, which allows the controller to modify the state of a switch. 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 com-

monly 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 also be applied to the other control flows. 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) values. Our framework adopts this idea to randomize input values of a control flow.

Fuzzing Algorithm: For efficiently randomizing the fuzzing vectors, we design a graph traversal algorithm that explores the operational state diagram (Fig. 3) as shown in Algorithm 1. When we

Algorithm 1 Fuzzing Algorithm with the state diagram.

Input:
G' is a fuzzing vector (subgraph)
v_0 is an initial state of G'
1: **procedure** INITIALIZE(G', v_0)
2: $\omega \leftarrow$ an empty list
3: $\gamma \leftarrow$ an empty list
4: CONTROLFLOWFUZZER(G', v_0, ω, γ)
Input:
ω is a transition path that consists of visited v and e
γ is a list of mutated transition paths
5: **procedure** CONTROLFLOWFUZZER(G, v, ω, γ)
6: append $v \rightarrow \omega$
7: **for all** edges $e \in G.adjacentEdges(v)$ **do**
8: **if** $e.action = receive$ **then**
9: **for all** edges $e' \in E$ where $e' \neq e$ **do**
10: $result \leftarrow$ SENDTOCONTROLLER($e'.msg$)
11: **if** $result = Error$ **then**
12: $\omega' \leftarrow \omega \cup e'$
13: append $\omega' \rightarrow \gamma$
14: **if** u exists, $u \in G.adjacentVertex(v, e)$ **then**
15: append $e \rightarrow \omega$
16: $\gamma' \leftarrow$ CONTROLFLOWFUZZER(G, u, ω, γ)
17: append $\gamma' \rightarrow \gamma$
18: **return** γ

choose a fuzzing vector, it is necessary to include the ready state R as a subset of vertices and the vertices that have all incoming edges. It guarantees that a target controller reaches all selected states.

Definition 3. Fuzzing vector is a subgraph G', where selected vertices have all incoming edges and include the ready state R.

The algorithm aims to incrementally mutate state transitions by visiting each state, taking the chosen fuzzing vector G' from the operational state diagram. The list ω denotes a transition path that is composed of visited v and e, where $v \in V$ and $e \in E$. The list γ denotes a list of mutated transition paths ω', which is an output of the algorithm.

The algorithm starts to traverse from the initial state v_0 (line 4). When the procedure `ControlFlowFuzzer` visits each state v, it appends the vertex v to ω to record the traversed states (line 6). Then, from the current state, the algorithm visits all the adjacent edges (transitions) and checks the edge's *action* field (lines 7 to 8). If the action field contains *receive*, the fuzzer selects an abnormal

[2] The OpenFlow specification suggests an encryption transport (e.g., TLS) to encrypt outgoing messages. However, it is frequently disabled in favor of performance (Benton et al., 2013).

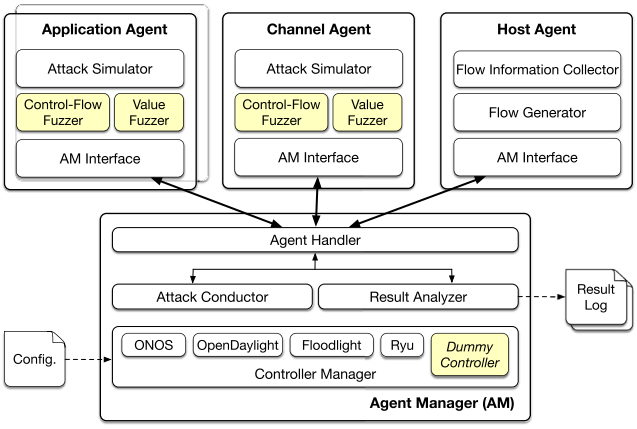


Fig. 8. Overall architecture of DELTA with four key components: (*i*) Agent Manager, (*ii*) Application Agent, (*iii*) Channel Agent, and (*iv*) Host Agent.

transition e' that is not the same with the original transition e from E. Upon the current state v, the algorithm sends the message $e'.msg$ to the controller and retrieves a result. If the sent message causes an error, the procedure makes a mutated path ω' as a union of e' and ω. Then, the mutated path ω' is stored to γ (lines 9 to 13). If there is an adjacent state u from the current state, the procedure appends the current edge e to ω and calls itself to visit all vertices recursively (lines 14 to 17).

Finally, the procedure returns a list of the mutated transition paths γ, which is a set of generated fuzz cases. The unexpected situation referred to in the fuzzing algorithm description is determined based on the seven test criteria that we have defined in Section 4.3. If the fuzz cases generated by DELTA result in any of these, the test inputs will be flagged for ex-post-facto vulnerability assessment.

5.3. System architecture

This section presents an overview of the overall architecture of DELTA and briefly explains each of its components. For a more detailed description, we point the reader to (Lee et al., 2017). As shown in Fig. 8, 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 along the path 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 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.

Remote Agents: The *application agent* is an SDN application running inside the controller. 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 API. 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. The control-

Table 2
Supported application agents for various controller versions.

	ONOS			*OpenDaylight*			*Floodlight*			*Ryu*	***Brocade Vyatta***
Version	1.6	1.9	2.1	Helium	Lithium	Carbon	0.91	1.1	1.2	4.16	***2.3.0***
Release Date	12/16/15	03/10/16	04/30/19	09/29 /14	06/29/15	05/26/17	12/30/14	04/17/15	02/07/16	08/02/17	***2016***
Supported	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

flow fuzzer and value fuzzer are used to randomize SDN control flows and their input values.

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. The modules of the channel agent are the same with those of the application agents.

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. The host agent consists of three modules: (*i*) Flow Information Collector, (*ii*) Flow Generator, and (*iii*) AM Interface. The flow information collector captures diverse flow-related information, such as latency and the number of sent and received flows. 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 set 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.

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 Section 4.3. 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.

6. Implementation

We have implemented an instance of DELTA to verify its feasibility and effectiveness. To support the design features described in Section 5, we implemented three types of agents and an AM in Java, in approximately 12,000 lines of code. DELTA has been open sourced as one of ONF's official open SDN projects (Lee et al., 2020).

DELTA currently includes application agents for four well-known open source controllers and one commercial controller, enabling it to replay attack scenarios and launch fuzzing functions as shown in Table 2. 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 supports 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. It can also collect network flow information by passively sniffing network packets. All agents have direct connections to the 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 and randomize sequences of function calls, and randomize arguments or the sequences 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.

Table 3
Unknown attack case classification: ASY (Asymmetric), SYM (Symmetric), INTRA (Intra-controller), INTER (Inter-controller), and ADMIN (admin) control flows.

Unknown Attack Name	*Flow*	*Target*
Sequence and Data-Forge	ASY	FL
Stats-Payload-Manipulation	SYM	FL, ODL
Echo-Reply-Payload-Manipulation	SYM	ODL
Service-Unregistration	INTRA	ODL
Flow-Rule-Obstruction	INTRA	ONOS
Host-Tracking-Neutralization	INTRA	ONOS
Link-Discovery-Neutralization	INTRA	FL
Heartbeat-Delay-Randomization	INTER	ONOS
Missing-Prerequisite	ADMIN	FL

7. Evaluation

We have conducted a wide range of experiments and performance evaluations involving the DELTA security assessment framework with well-known SDN controllers, ONOS (v1.9.0), OpenDaylight (Carbon), Floodlight (v1.2), Ryu (v4.16) and a commercial controller (Brocade Vyatta v2.3.0). In this section we present a range of results illustrating the penetration testing capability of DELTA across a diversity of SDN stacks, as per the objective of our framework.

7.1. 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 (FL) controllers. Table 3 summarizes 9 new attack scenarios[3] that were revealed through our evaluation. These scenarios span all five SDN control flow categories (symmetric, asymmetric, intra-controller, inter-controller, and admin control flows).

7.1.1. 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

[3] The first 7 unknown attack cases refer to (Lee et al., 2017)

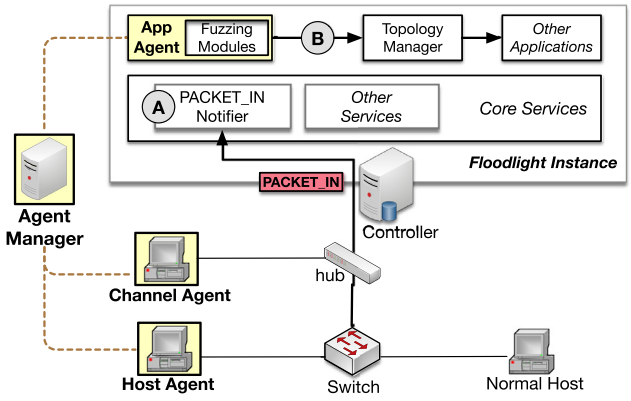


Fig. 9. Fuzz points of the Sequence and Data-Forge attacks.

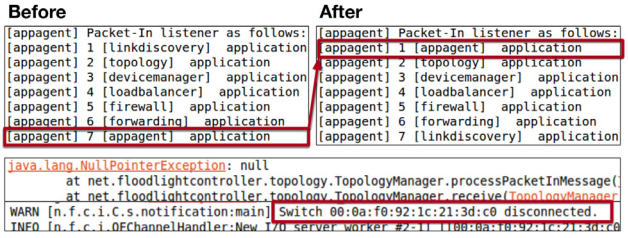


Fig. 10. Results of the Sequence and Data-Forge attack experiment.

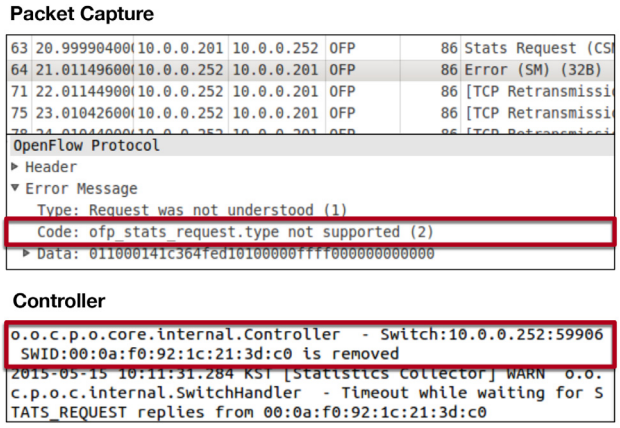


Fig. 11. Results of the Stats-Payload-Manipulation attack experiment.

`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.

Using the control-flow fuzzer and the value fuzzer in the application agent, Fig. 9 illustrates the attack scenario, highlighting the points where the fuzzing modules randomize. Specifically, the control-flow fuzzer randomizes the delivery sequence of PACKET_IN messages (A in Fig. 9), and the value fuzzer randomizes the message payloads (B in Fig. 9). 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 Fig. 9) 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 a previously unknown attack scenario.

Based on the log file generated by the result analyzer in the AM, we re-examine the *unknown* case. Fig. 10 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 Fig. 10). However, after modifying the sequence, the application agent is moved to the first entry of the 'After' column in Fig. 10.

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 payload from the message, and then the switch that sent the PACKET_IN message is subsequently disconnected (i.e., criterion (iv) switch disconnection as defined in Section 4.3).

7.1.2. Stats-Payload-Manipulation attack

As mentioned in Section 2.2, 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.

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 AM notices the switch disconnection matched to our criteria.

Fig. 11 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 Fig. 11) to the controller, and the switch is disconnected from the controller (see Controller in Fig. 11), which violates the switch-disconnection criterion.

7.1.3. Echo-Reply-Payload-Manipulation attack

In the case of the 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 (i.e., criterion (iv) switch disconnection).

From the log information, we try to reproduce this attack case. Fig. 12 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 Fig. 12), the controller causes the exception to parse the wrong length value of the message. Finally, the switch is disconnected from the controller (Controller in Fig. 12).

7.1.4. 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

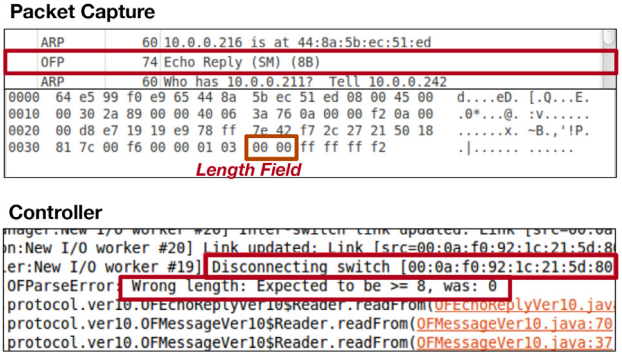


Fig. 12. Results of the Echo-Reply-Payload-Manipulation attack experiment.

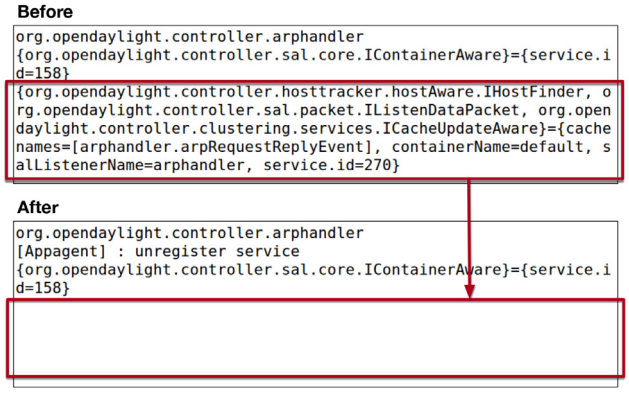


Fig. 13. Results of the Service-Unregistration attack experiment.

register for 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.

Based on the log file, we can backtrack this attack scenario. As shown in Fig. 13, the `ArpHandler` initially registered three kinds of services: IHostFinder, IListenDataPacket, and ICacheUpdateAware (Before in Fig. 13). After the fuzzing modules unregister the services, the network loses its functionality, since ARP packets play a critical role during the initiation of network communications (After in Fig. 13). Therefore, two hosts that are connected to the switch cannot communicate with each other (i.e., criterion (vi): inter-host communication disconnection).

7.1.5. 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.

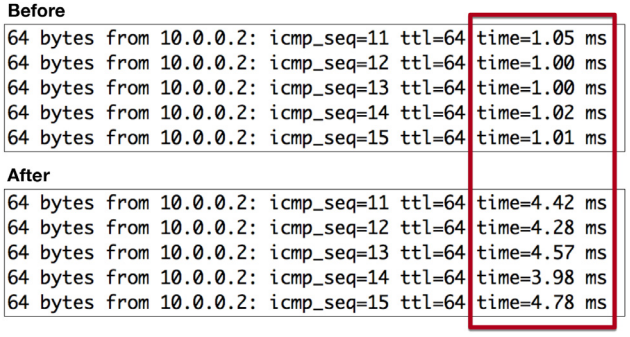


Fig. 14. Results of the Flow-Rule-Obstruction attack experiment.

This attack scenario was discovered by targeting DELTA to the intra-controller control flows. The value fuzzer in the application agent chooses the 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.

With the log file, we can verify the feasibility of this attack. Fig. 14 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 Fig. 14 bottom) than the average before the attack (about 1 ms in Fig. 14 top) as the workload of the controller increases (i.e., criterion (v): switch performance downgrade).

7.1.6. 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, and a connected port). For example, if an end-host attaches to a switch, the service identifies this 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 is selected by the value fuzzer in the application agent for input-value randomization. While the value fuzzer runs, 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. Fig. 15 shows the outputs from a packet capture tool (Orebaugh et al., 2006) 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

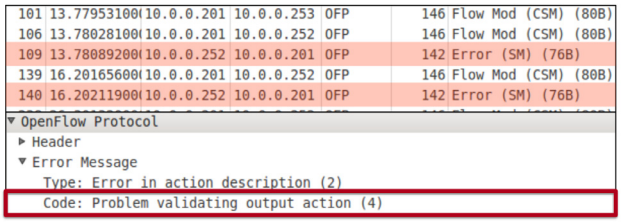


Fig. 15. Results of the Host-Tracking-Neutralization attack experiment.

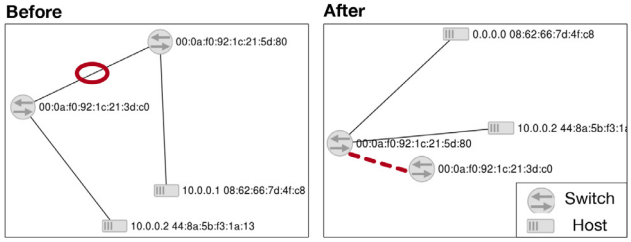


Fig. 16. Results of the Link-Discovery-Neutralization attack experiment. A circle (before) represents a live link between two switches, and a dotted line (after) represents a failed link.

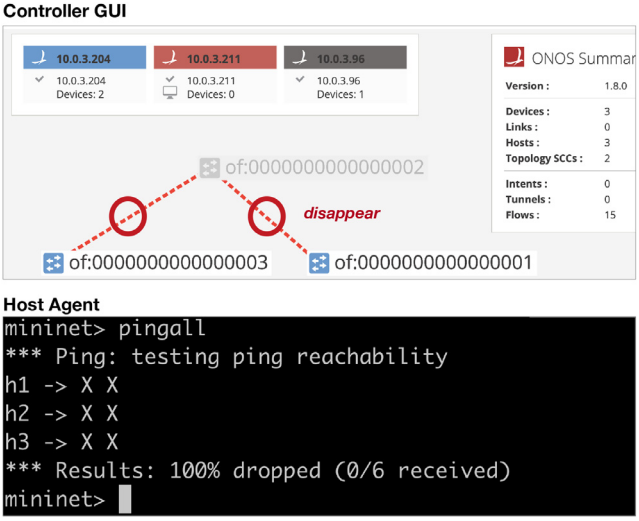


Fig. 17. Results of the Heartbeat-Delay-Randomization attack experiment.

available due to the invalid host. However, although the communication ends, error messages are sent to the controller every 10 s until the controller shuts down (i.e., criterion (vii): error-packet generation).

7.1.7. 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 packets 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 LinkDiscoveryService, which suppresses the sending of LLDP packets.

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 Fig. 16, the controller web UI displays the correct network topology information (Before in Fig. 16). However, after the attack is conducted, the topology information is changed, although the real topology has not been altered (After in Fig. 16).

7.1.8. Heartbeat-Delay-Randomization attack

To implement synchronization among distributed ONOS controllers, they leverage a RAFT algorithm (Ongaro and Ousterhout, 2014), which is a consensus algorithm that achieves a leader selection from those instances. By synchronizing the different state transitions with the eventual consistency concept, the RAFT enables the instances to decide a single leader per a switch. Thus, the selected leader has an authority of the switch. Also, for each instance, receiving *heartbeat* messages from their neighbors through the east-west interface is important to know whether the neighbors are alive or not. If an instance does not receive the heartbeat message, it thinks that the neighbor is dead and tries to elect a new leader.

Paying attention to this, using DELTA, we were able to find out that there exists a vulnerability of the heartbeat mechanism operated in the distributed ONOS controllers, and its scenarios is as follows: the channel agent in DELTA conducts the port scanning to one of the instances from the cluster first. If the agent detects the port 9876 (i.e., the default port for ONOS inter-controller communication) of the instance is opened, it assumes that the port is used for exchanging the heartbeat messages among the instances, so the agent starts to sniff the packets going through it. Here, when the fuzzing module in the channel agent drops and delays the messages arbitrarily, the network state becomes unstable as shown in Fig. 17. Specifically, each instance cannot keep the link states for the entire network, resulting in that all the links disappear from the ONOS cluster database (Controller in Fig. 17), which violates one of our criteria (i.e., criterion (iii) internal-storage poisoning). Thus, this unstable link information can affect other applications to make a wrong decision. More seriously, the host agent cannot communicate each other again although the channel agent stopped dropping and delaying the heartbeat messages (Host Agent in Fig. 17).

7.1.9. Missing-Prerequisite attack

According to the OpenFlow specification (OpenFlow, 2011), If a user wants to use the TCP/UDP port numbers in match fields of a flow rule, she should specify which IP protocol will be used together, which is called a prerequisite. If they do not comply with this, the SDN controller should deny such flow rule request from the users, and then notify them of an error.

In this instance, the DELTA operator targets the admin control flows first. Then, the value fuzzer in the application agent randomly generates the flow rule request that includes the source IP address and IP protocol as the match fields through the RESTful services. At this time, the flow rule requests are remotely pushed in the same way that the administrator manually configures. After receiving the requests from the RESTful services, the controller builds the FLOW_MOD messages, but here the fuzzer disrupts the prerequisite by removing the IP protocol from the match fields. Finally, when processing the flow rule, the controller disconnects the switch, the AM notices this disconnection event (i.e., criterion (iv) switch disconnection).

Fig. 18 shows the results of the Missing-Prerequisite attack. When the value fuzzer manipulates the prerequisite of FLOW_MOD

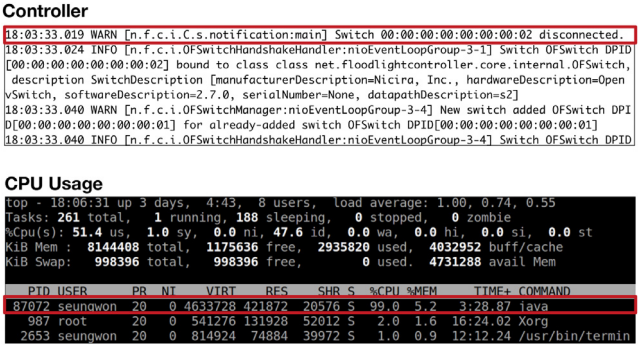


Fig. 18. Results of the Missing-Prerequisite attack experiment.

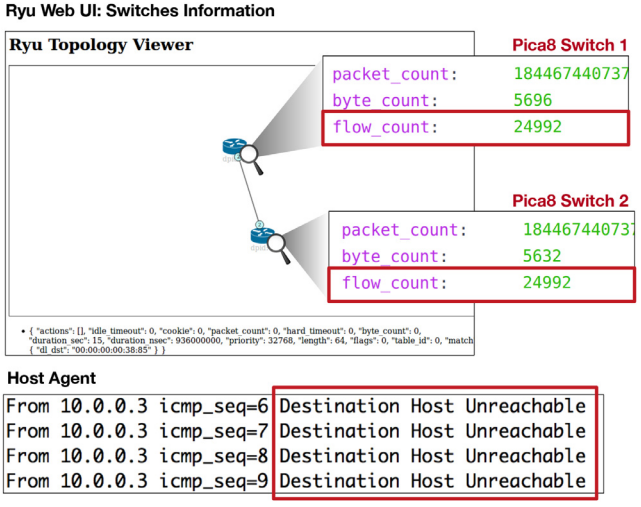


Fig. 19. Results of the Flow Rule Flooding attack experiment.

message by removing the IP protocol field, it causes a CPU burst (CPU Usage in Fig. 18) due to an infinite loop within the controller, so the switch is disconnected from the controller (Controller in Fig. 18).

7.2. 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: Flow-Rule-Flooding Attack and Application Eviction attack.

7.2.1. Flow rule flooding attack

To issue flow rules on the switches, SDN applications can employ useful APIs provided by SDN controllers. However, the problem is that there are no restrictions on issuing flow rules. Thus, a malicious application can keep generating flow rules to fill up the flow tables of SDN enabled switches to mislead the switches into performance degradation or unexpected status.

Fig. 19 shows an example of conducting this attack with the Ryu controller. It denotes the Ryu web UI (Fig. 19 top) and ping results from the host agent (Fig. 19 bottom). When reproducing this attack, the number of flow rules in the switch significantly increases within seconds (Fig. 19 top), and thus it exhausts the flow table of the switch. Then, the host agent cannot communicate with other hosts (Fig. 19 bottom), because there is no space to add new flow rules corresponding to the connections of the host agent (i.e., criterion (vi) inter-host communication disconnection).

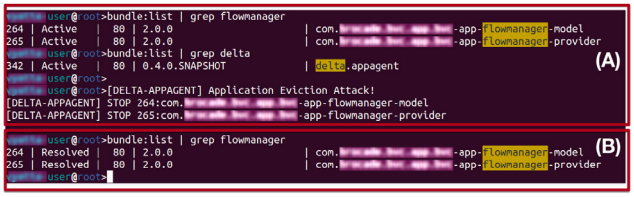


Fig. 20. Results of the Application Eviction attack experiment.

Table 4
Finding unknown attack microbenchmark.

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
Inter-controller control flow	89.2 sec
Admin control flow	76.5 sec

7.2.2. Application eviction attack

Most controllers adopt a mechanism that can allow users to dynamically load and unload an application running on the controller. However, due to no restriction on using this mechanism, an application can arbitrarily unload other applications. Here, we demonstrate an attack against the commercial Brocade Vyatta SDN controller (Brocade, 2016), which is based on OpenDaylight.

Once the target controller has been initialized, as shown in Fig. 20 (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 (Fig. 20 (B)).

The demonstration of this range of attack cases (both known and unknown) across the diversity of commercial and open source controllers illustrates the flexibility of DELTA design, and the potential for its use in security testing across an even broader range of controllers.

7.3. Performance

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 4 shows the amount of time taken to complete one fuzz test cycle. Actually, it can be dependent on the scale of the testbed and the number of the fuzz points. But, the results in Table 4 take the simple test topology as shown in Fig. 9 and one fuzz point per each cycle.

8. 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, reproducing 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.

9. 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 9 previously unknown attack scenarios that were detected by our tool.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2018-0-00254, SDN security technology development).

References

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

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

Big Switch Networks, Floodlight. http://www.projectfloodlight.org/floodlight/.

BLACK-HAT-USA-2016, Delta: Sdn security evaluation framework. https://www.blackhat.com/us-16/briefings/schedule/index.html.

BLACK-HAT-USA-2017, Attacking sdn infrastructure: are we ready for the next-gen networking?https://www.blackhat.com/us-17/arsenal/schedule/index.html.

BLACK-HAT-USA-2018, The finest penetration testing framework for software-defined networks. https://www.blackhat.com/us-18/briefings/schedule/index.html.

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

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

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

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

Dhawan, M., Poddar, R., Mahajan, K., Mann, V., 2015. Sphinx: detecting security attacks in software-defined networks.. NDSS.

Dover, J. M., 2013. A denial of service attack against the Open Floodlight SDN controller. Dover Networks, Tech. Rep.

Dover, J. M., 2017. A switch table vulnerability in the Open Floodlight SDN controller. url: http://dovernetworks.com/wpcontent/uploads/2014/03/OpenFloodlight-03052014.pdf.

Fyodor, Nmap security scanner. http://www.nmap.org.

Godefroid, P., Levin, M.Y., Molnar, D.A., 2008. Automated whitebox fuzz testing.. In: NDSS, 8, pp. 151–166.

Hizver, J., Taxonomic modeling of security threats in software Defined. Blackhat 2015. https://www.blackhat.com/us-15/briefings/schedule/index.html.

Hocevar, S., zzuf. https://github.com/samhocevar/zzuf.

Hong, C.-Y., Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M., Wattenhofer, R., 2013. Achieving high utilization with software-driven wan. In: ACM SIGCOMM Computer Communication Review, 43. ACM, pp. 15–26.

Hong, K., Xu, L., Wang, H., Gu, G., 2015. 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).

HP, HP SDN App Store. https://marketplace.saas.hpe.com/sdn.

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

Jero, S., Bu, X., Nita-Rotaru, C., Okhravi, H., Skowyra, R., Fahmy, S., Beads: automated attack discovery in openflow-based sdn systems.

Kotani, D., Okabe, Y., 2014. 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. ACM, New York, NY, USA, pp. 29–40. doi:10.1145/2658260.2658276.

Kreutz, D., Ramos, F.M., Verissimo, P., Rothenberg, C.E., Azodolmolky, S., Uhlig, S., 2015. Software-defined networking: a comprehensive survey. Proc. IEEE 103 (1), 14–76.

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

Lee, S., Yoon, C., Lee, C., Shin, S., Yegneswaran, V., Porras, P.A., 2017. Delta: A security assessment framework for software-defined networks.. NDSS.

Maynor, D., 2011. Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research. Elsevier.

Medved, J., Varga, R., Tkacik, A., Gray, K., 2014. Opendaylight: Towards a model–driven sdn controller architecture. In: 2014 IEEE 15th International Symposium on. IEEE, pp. 1–6.

Miller, B.P., Fredriksen, L., So, B., 1990. An empirical study of the reliability of unix utilities. Commun. ACM.

NTT Communications, Ryu. http://osrg.github.io/ryu/.

Oktian, Y.E., Lee, S., Lee, H., Lam, J., 2017. Distributed sdn controller system: a survey on design choice. Comput. Networks 121, 100–111.

Ongaro, D., Ousterhout, J., 2014. In search of an understandable consensus algorithm. In: 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14), pp. 305–319.

ONOS, Cord: Reinventing central offices for efficiency & agility. https://opencord.org.

Open Networking Foundation, https://www.opennetworking.org/.

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

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

OpenFlow, 2014. OpenFlow Swtch Specification version 1.5.0. Technical Report. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf

Orebaugh, A., Ramirez, G., Beale, J., 2006. Wireshark & Ethereal network protocol analyzer toolkit. Elsevier.

Porras, P., Shin, S., Yegneswaran, V., Fong, M., Tyson, M., Gu, G., 2012. A security enforcement kernel for openflow networks. In: Proceedings of the first workshop on Hot topics in software defined networks (HotSDN'12).

Röpke, C., Holz, T., 2015. Sdn Rootkits: subverting Network Operating Systems of Software-defined Networks. In: Research in Attacks, Intrusions, and Defenses. Springer, pp. 339–356.

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

Secci, S., Attou, K., Phung, D., Scott-Hayward, S., Smyth, D., Vemuri, S., Wang, Y., 2017. ONOS Security and Performance Analysis: Report No. 1. https://www-phare.lip6.fr/~secci/papers/ONOS-security-and-performance-analysis-brigade-report-no1.pdf.

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

SDNSecurity.org, SDN Security Vulnerabilities Genome Project. http://edisonchicken.cafe24.com/vulnerability/attacks/.

Security, T. N., Nessus. http://www.tenable.com/products/nessus-vulnerability-scanner.html.

Shin, S., Gu, G., 2013. 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).

Shin, S., Song, Y., Lee, T., Lee, S., Chung, J., Porras, P., Yegneswaran, V., Noh, J., Kang, B.B., 2014. 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).
Shin, S., Yegneswaran, V., Porras, P., Gu, G., 2013. 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).
Takanen, A., Demott, J. D., Miller, C., Fuzzing for Software Security Testing and Quality Assurance. http://www.mcafee.com/us/products/network-security-platform.aspx.
Ujcich, B.E., Thakore, U., Sanders, W.H., 2017. Attain: An attack injection framework for software-defined networking. In: Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on. IEEE, pp. 567–578.
Yao, J., Wang, Z., Yin, X., Shiyz, X., Wu, J., 2014. Formal modeling and systematic black-box testing of sdn data plane. In: Network Protocols (ICNP), 2014 IEEE 22nd International Conference on. IEEE, pp. 179–190.

Seungsoo Lee is a Ph.D. student in Graduate School of Information Security at KAIST working with Dr. Seungwon Shin in NSS Lab. He received his B.S. degree in Computer Science from Soongsil University in Korea. He received his M.S. degree in Information Security from KAIST. His research interests include secure and robust SDN controller, and protecting SDN environments from threats.

Jinwoo Kim is a Ph.D studnet in the School of Electrical Engineering at KAIST. He received his M.S degree in Graduate School of Information Security from KAIST, and his B.S degree in Computer Science and Engineering from Chungnam National University. His research topic mainly focus on Software Defined Networking (SDN) security, designing a network security system, and an applied network theory.

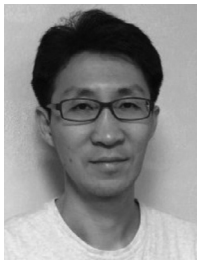
Seungwon Woo is a researcher at ETRI. He received his M.S degree in Graduate School of Information Security from KAIST, and his B.S degree in Computer Science and Engineering from Chungnam National University. He is interested in SDN security and blockchain area.

Changhoon Yoon is a director of research in S2W Lab. He received his B.S degree in Computer Engineering from the University of Michigan, Ann Arbor in 2010 and his M.S degree in Information Security from KAIST in 2014. He received his PhD degree in Information Security from KAIST in 2019.

Dr. Sandra Scott-Hayward, CEng CISSP CEH, is a Lecturer (Assistant Professor) in Network Security at Queen's University Belfast. In the Centre for Secure Information Technologies at QUB, Sandra leads research and development of network security architectures and security functions for software-defined networks (SDN) and network functions virtualization (NFV). She has presented her research globally and received Outstanding Technical Contributor and Outstanding Leadership awards from the Open Networking Foundation (ONF) in 2015 and 2016, respectively.

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

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



Seungwon Shin is an associate professor in the School of Electrical Engineering at KAIST. He received his Ph.D. degree in Computer Engineering from the Electrical and Computer Engineering Department, Texas A&M University, and his M.S degree and B.S degree from KAIST, both in Electrical and Computer Engineering. He is currently a Research Associate of Open Networking Foundation (ONF), and a member of security working group at ONF. His research interests span the areas of Software Defined Networking (SDN) security, IoT security, and Botnet analysis/detection.