BEADS: Automated Attack Discovery in OpenFlow-based SDN Systems*

Samuel Jero[1], Xiangyu Bu[1], Cristina Nita-Rotaru[2], Hamed Okhravi[3], Richard Skowyra[3], and Sonia Fahmy[1]

[1] Purdue University, West Lafayette, IN, USA
{sjero,bu1,fahmy}@purdue.edu
[2] Northeastern University, Boston, MA, USA
c.nitarotaru@neu.edu
[3] MIT Lincoln Laboratory, Lexington, MA, USA
{hamed.okhravi,richard.skowyra}@ll.mit.edu

Abstract. We create BEADS, a framework to automatically generate test scenarios and find attacks in SDN systems. The scenarios capture attacks caused by malicious switches that do not obey the OpenFlow protocol and malicious hosts that do not obey the ARP protocol. We generated and tested almost 19,000 scenarios that consist of sending malformed messages or not properly delivering them, and found 831 unique bugs across four well-known SDN controllers: Ryu, POX, Floodlight, and ONOS. We classify these bugs into 28 categories based on their impact; 10 of these categories are new, not previously reported. We demonstrate how an attacker can leverage several of these bugs by manually creating 4 representative attacks that impact high-level network goals such as availability and network topology.

1 Introduction

Software-defined networking (SDN) is an attractive alternative to traditional networking, offering benefits for large enterprise and data-center networks. In SDNs, the control and management of the network (*i.e.,* the control plane) is separated from the delivery of data to the destinations (*i.e.,* the data plane). Such a separation offers enhanced manageability, flexibility, and programmability to the network administrators, enabling them to perform better resource allocation, centralized monitoring, and dynamic network reconfiguration.

SDN's benefits, however, come at a cost to security. The programmability and malleability of the network presents new attack surfaces. In addition to the network-based attacks applicable to traditional networks, new attack vectors

* DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

This material is based upon work supported by the Department of Defense under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense.

are available to an attacker to maliciously impact the network functionality by manipulating, poisoning, or abusing the malleable network logic. For example, we show that ARP spoofing attacks have broader impact in SDNs because of the centralized control. In particular, many controllers maintain a centralized ARP cache and implement Proxy ARP to resolve ARP queries, making the impact of poisoning this cache much broader than in traditional networks.

Recent efforts at the intersection of SDN and security have focused on developing new languages for SDN programming, some of which offer formally verifiable guarantees [9,15,30,43], such as flow rule consistency [16,18,40]. Some work has focused on possible attacks from the data plane to control plane and vice versa [44]. Protocol-level attacks and corresponding defenses have also been studied [7, 10, 16, 42]. Finally, the dynamism and agility offered by SDNs has been leveraged to build new defenses [11, 13, 25]. Several of these approaches have identified specific attacks in the context of SDNs [7,10,16,38,42,44]. These efforts highlight the need for systematic approaches to find attacks in SDNs.

In order to systematize OpenFlow testing, the Open Networking Foundation created conformance test documents for OpenFlow 1.0.1 [32] and 1.3.4 [34]. Following these documents, the SDN community started two projects, OFTest [8] and FLORENCE [29]. Both of them focus only on OpenFlow switches and consist of manually written tests. OFTest supports 478 manually written tests for OpenFlow 1.0-1.4, while FLORENCE supports 18 manually written tests for OpenFlow 1.3. Examples of tests performed are: `AllPortStats`, which "Verif[ies] [that] all port stats are properly retrieved" for OFTest and `Port Range test` to "Verify that the switch rejects the use of ports that are greater than OFPP_MAX and are not part of the reserved ports" for FLORENCE.

Both OFTest and FLORENCE focus on testing how well a switch conforms to the OpenFlow specification. However, OpenFlow is a configuration protocol; it specifies *how* a controller instructs a switch to do something, but not *what* the controller should tell the switch to do. As a result, many bugs and attacks on SDNs arise from incorrect assumptions in the controller software about the switches. Frameworks like OFTest and FLORENCE that exclude the controller from the testing process are unable to find such issues.

Further, conformance testing is not sufficient to detect attacks. In fact, the Open Flow Foundation conformance testing documents explicitly state: *"This document does not include requirements or test procedures to validate security, interoperability or performance."* Previous work on automated attack finding on communication protocols has been confined to distributed systems [23] and transport protocols [12] which are less complex than SDN systems.

In this work, we develop BEADS, a framework to automatically and systematically test SDN systems for attacks resulting from malicious switches and malicious hosts. Our framework automatically generates and tests thousands of scenarios involving malicious switches that do not obey the OpenFlow protocol and malicious hosts that do not obey the ARP protocol. BEADS combines known techniques such as Byzantine fault injection, semantically-aware testcase generation, and black box testing to test whole SDN systems comprising Open-

Flow switches, controllers, and hosts. As such it differs from existing SDN testing tools in the following aspects: (1) it supports malicious (Byzantine) participants – hosts and switches; (2) it does not require access to the the code of the switch or controller; (3) it targets attacks at a deeper layer than simple parsing (that can be tested using simple random fuzzers); (4) it achieves higher coverage by using message grammar and *semantically-aware* test case generation; (5) it can test controller algorithms like routing, topology detection, and flow rule management by also including the controller in its test cases; (6) it makes better use of resources by performing targeted and preferential search. BEADS is publicly available at `https://github.com/samueljero/BEADS`.

Using BEADS, we identify bugs that trigger error messages, network topology or reachability changes, or increased load. We then show that these bugs can be exploited with damaging impacts on SDN networks. Our results show the importance of malicious testing for SDNs as well as the practicality of blackbox testing for such systems. Our contributions are:

• We create BEADS, a framework to automatically find malicious switch- and host-level attacks. BEADS combines network emulation with software switches and real SDN controllers running in a virtualized environment. It takes a blackbox approach to the SDN switches and controller and does not require access to the source code of either. Attack scenarios are automatically generated based on message grammar and the protocol semantics associated with special fields (such as port). BEADS uses four criteria to detect bugs: error messages, network topology changes, reachability changes, and controller or switch load.

• We use BEADS to automatically test almost 19,000 scenarios, and find 831 unique bugs across four well-known SDN controllers: Ryu [45], POX [27], ONOS [5], and Floodlight [39]. We classify these bugs into 28 categories based on their impact; 10 of these categories have not been previously reported. Outcomes include preventing the installation of flow rules network-wide, periodic switch disconnections, inducing packet loss in the data plane, denial of service against the controller, and removing network links.

• We construct and implement 4 representative attack scenarios using several bugs we identified to break high-level network goals such as availability, reachability, and network connectivity. The scenarios are (1) TLS Man-in-the-Middle, (2) Web Server Impersonation, (3) Breaking Network Quarantine, and (4) Deniable Denial of Service. We demonstrate the feasibility of these attack scenarios on real SDN controllers.

• We have notified the SDN vendors of bugs we found. Ryu has issued a patch (CD2,CD3 in Table 2) while ONOS has confirmed that the latest version is no longer impacted (EP1 in Table 2).

Roadmap. Section 2 specifies the threat models. Section 3 describes the design of BEADS. Section 4 discusses the bugs we found and presents our attack demonstrations. Section 5 discusses some limitations of BEADS while Section 6 summarizes related work and Section 7 concludes the paper.

2 Threat Model

We consider a threat model where the attacker can control compromised SDN switches or end-hosts connected to the SDN. We consider malicious switches because prior work has shown that many SDN switches can be easily compromised due to running operating systems with poor security defaults, out of date software, and minimal updates [35, 36] and, once compromised, they can influence the entire control plane. Note that if communication is not conducted over secure channels, a man-in-the-middle attacker can control otherwise uncompromised switches and hosts. We do not consider malicious controllers.

Malicious Switches. Attackers who have compromised an OpenFlow switch can confuse SDN controllers via malicious OpenFlow messages. This ability is unique to SDNs and can confuse the controller about the network topology and the locations of target hosts [7,10]. Additionally, a malicious OpenFlow switch can mount a DoS attack against the controller by sending OpenFlow messages, spoofed or legitimate, at a very high rate. Some controllers enforce per-switch OpenFlow rate limits in an attempt to mitigate this type of attack [7]. Recent work has shown that OpenFlow switches are extremely vulnerable to attackers, running old, unsecured software versions with default/hidden administrator accounts, out of date software, and minimal updates [35, 36].

Our analysis focuses on how malicious switches can disrupt or degrade *other* parts of the network (*e.g.,* QoS on other switches or making the controller redirect distant traffic through a compromised switch) via the control-plane. Thus, we do not consider pure data-plane attacks (*e.g.,* dropping packets). We model malicious switches as having the following basic capabilities with respect to OpenFlow messages between the switch and controller:

Drop (percentage). This action drops a particular type of OpenFlow message with a given probability specified as a parameter, for example `barrier_request drop 20`. This emulates a malicious switch that does not send these messages or ignores them after receiving them.

Duplicate (times). This action duplicates a particular type of OpenFlow message a certain number of times given as a parameter. For example `barrier_reply duplicate 5` means the malicious switch duplicates this messages 5 times.

Delay (msec). This action delays a particular type of OpenFlow message by a given number of milliseconds, emulating a malicious switch that delays processing a request or taking some action; for example, `of_hello delay 1000`.

Change (field, value). This action modifies a particular field of a particular type of OpenFlow message with a particular value. Modifications supported include setting a particular value as well as adding or subtracting a constant. We select the modification values to be likely to trigger problems based on the field type. This typically includes values like 0, minimum field value, and maximum field value. This basic strategy corresponds to a malicious switch that performs a different action or returns different information than that requested by the controller. Examples of this action include `flow_add change priority 42` or `flow_removed change reason 12`.

Malicious Local Hosts. Attackers who have compromised a host that is directly connected to an SDN, like a server or a user workstation, can launch attacks to confuse the SDN controller about the network topology and the location of target hosts, in order to hijack a target host or traffic of interest [7,10]. These are primarily attacks that target the Address Resolution Protocol (ARP) [37] since ARP is one of the few protocols that hosts can use to manipulate the SDN control plane. Prior work has also pointed out that hosts can inject or tunnel LLDP packets [7, 10]. However, we need not separately consider such hosts because they appear to the network as malicious switches, which we already consider.

For ARP, SDN has brought back known vulnerabilities because, while traditional networks have deployed defenses against ARP spoofing, these defenses have not been adapted for SDNs. Unlike traditional network switches that maintain their own local ARP tables, operate on L2/L3 networks, and can be checked to prevent ARP poisoning attacks, SDN switches consist of a programmable flow table and leave the SDN controllers to check for ARP corruption. Such controllers do not currently implement ARP spoofing defenses. Moreover, some controllers (including POX and ONOS) maintain a centralized ARP cache and implement Proxy ARP to resolve ARP queries. This creates a single, centralized ARP cache for the entire network. Poisoning this cache has broader network-wide impact rather than limited subnetwork-wide impact as in traditional networks.

We model malicious or compromised local hosts as follows:

ARP-location-injection (victim-MAC, victim-IP). The malicious host injects ARP packets with the spoofed Ethernet source address of the victim to make the controller believe that the victim is at the same port as the attacker. Example: `ARP-location-injection 00:00:00:00:00:04 10.0.0.4`.

ARP-map-injection (attacker-MAC, victim-IP). The malicious host injects ARP packets that indicate a mapping between the victim's IP and the attacker's MAC. This disrupts the IP-to-MAC mapping, and leads the controller to believe that the attacker has the victim's IP address. An example of this attack would be `ARP-map-injection 00:00:00:00:00:01 10.0.0.4`.

3 BEADS Design and Implementation

We first describe the design principles behind BEADS and then provide more details about each component.

3.1 Design Goals

There are several guiding principles behind BEADS: automation of attack generation and attack search, realism by testing real-world implementations of complete SDN systems generalizable to many different implementations independent of language and operating system, reproducibility of the results, high coverage of test scenarios, efficient use of resources, and last but not least, focus on *security* (rather than conformance) by supporting malicious switches and hosts.

Our main goal is to test real-world implementations of complete SDN systems. This requires including switches, controllers, and hosts in our tests to be able to capture their interplay. Similarly, our tests would ideally include the physical hardware and operating systems running the various system nodes, the production network connecting the physical hardware, and the actual application binaries running on the operating systems. However, using physical hardware to test all possible configurations and operating systems is not scalable and prohibitively expensive. We address this challenge by choosing a virtualized environment that supports different operating systems and languages, and network emulation using Mininet [21] that enables strong control over the network, while still being close enough to a real-world installation.

Another design goal is to run actual implementations of the system of interest without discriminating based on programming language, compiler, toolkit, or target operating system, and without imposing restrictions solely for visibility into algorithmic behavior. Ideally, we should use the same implementation that will be deployed. We achieve this goal by using a *proxy* to create the behavior of the malicious switch. Malicious host behavior is injected directly into the real data-plane, similarly enabling the use of unmodified switches and controllers.

Finally, a major goal for our test case generation is to create meaningful, semantically-aware test cases that can go beyond testing parsers. One simple method to generate an attack strategy is to use random fuzzing where the entire packet or some of its fields are replaced with random strings. While random fuzzing has been used successfully to test API inputs, it would have a low success rate for OpenFlow messages. OpenFlow messages are complex data structures involving many layers of nested objects as well as other syntactic and semantic dependencies. Although any packet can be represented as a bit string, the majority of bit strings are not valid OpenFlow messages. Hence, the majority of test cases generated by random fuzzing only test the OpenFlow message parser while the attacks we are interested in lie at a much deeper layer, in the algorithms creating and processing those messages. Basic knowledge of the packet format or fields helps generate valid, meaningful messages by satisfying the syntactic requirement. However, an ideal testing tool should also consider the semantic meaning of different packet fields. For example, it should treat a field representing a switch port number differently from a field representing an IP address, and treat the switch port number differently from the length of an embedded structure. This approach enables testing on semantically meaningful, yet problematic values for a field—for example IP addresses actually in the network—and provides a means for tuning the testing to focus on particular types or locations of attacks. Similarly, our test generation creates tests with malicious hosts and switches at multiple locations in our test topology to ensure that our results are general and not tied to a specific topology.

3.2 Design Details

Our automated attack discovery platform, BEADS, is depicted in Figure 1. We separate the attack strategy generation functionality controlled by a *Coordinator*

from the testing of a strategy in an SDN system controlled by an *Execution Manager* (*Manager* for short). Several managers can run in parallel under the reign of one coordinator. The coordinator has three roles: generate attack strategies, assign those strategies to different managers for testing, and receive feedback about the execution of those strategies and their results. The attack strategies are generated based on the format of the messages and on the network topology in order to choose what entity (host or switch) will behave maliciously. The coordinator generates strategies for both malicious host and malicious switch behavior and decides how to interleave them. The coordinator uses feedback from the execution and testing of prior strategies for future strategy generation.

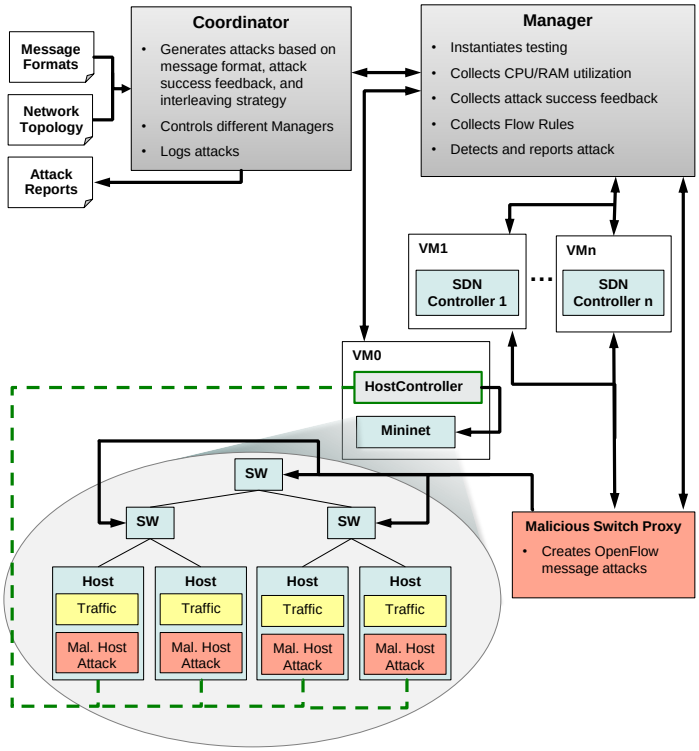

Fig. 1: BEADS Framework Design

The (execution) manager controls the execution environment for a set of attack strategy tests. This environment consists of an SDN, with a given topology, a specified placement and type of attackers (hosts and/or switches), as well as a list of attack strategies and a mechanism for interleaving host and switch strategies. BEADS combines network emulation using software switches and emulated hosts with real SDN controllers running in a virtualized environment. We select

Mininet for emulation because it offers the flexibility to test different network topologies and traffic patterns while providing attack isolation and increased reproducibility. Mininet exhibits high fidelity through the use of real network stacks, software switches, and real traffic. We leverage virtualization to run a wide range of SDN controllers independent of required operating systems, libraries, or system configurations. BEADS does not require access to the source code of the OpenFlow implementation in the switch or controller.

The manager uses Mininet to create an emulated data-plane network consisting of SDN switches and emulated hosts capable of both generating normal network traffic and injecting host-based attacks. This network is controlled using OpenFlow by one or several controllers running on a separate virtual machine, as depicted in Figure 1. A *Host Controller* running on the same virtual machine as Mininet controls the hosts connected to the testing network. Each host has a *Traffic* component that generates the traffic used during testing, and a *Malicious Host Attack* component that injects attacks that emulate a malicious host according to the strategy and timing specified by the HostController and received in turn from the manager of the execution environment. Finally, a *Malicious Switch Proxy* intercepts all messages in both directions between the SDN switches and the SDN controllers and creates malicious switch behavior according to strategies received from the manager of the execution environment.

The entire system from strategy generation to strategy testing and bug detection is automated. The user supplies the controller under test and receives a list of strategies that trigger bugs as the output. The user is then responsible to manually examine these strategies and identify the bugs triggered and any fixes required, as we do in Section 4.

3.3 Strategy Generation

Two questions must be answered to enable the coordinator to perform automatic strategy generation in BEADS: (1) what is an attack strategy and (2) when to inject an attack strategy. We discuss these aspects below.

In order to target attacks beyond simple parsing validation, we use attack strategies that represent malicious actions (by compromised switches or hosts) which target protocol packets: OpenFlow for malicious switches and ARP for malicious hosts. A detailed list of these actions is presented in Section 2. BEADS supports the testing of malicious switch actions, malicious host actions, and combinations of both. For combinations, we need some form of coordination between malicious host and malicious switch actions, as they are launched from independent components, not necessarily connected. We currently provide a basic level of coordination between these strategies based on time relative to the start of each test; *i.e.,* we specify the time at which different malicious switch and malicious host actions occur relative to the start of the test.

The injection points differ for malicious hosts and malicious switches. In the case of malicious switches, the attack is executed by a malicious proxy which has the ability to modify or affect the delivery of OpenFlow messages as discussed in Section 2. At the start of each test we define a set of one or more malicious

switches. For these switches, we then use send-based attack injection; *i.e.,* when the proxy receives an OpenFlow message to/from these switches, it takes any actions relevant to that message type, and forwards the message to its destination. To curtail state space explosion, we only consider strategies where the same action is applied to every message of a given type. These strategies manipulate the delivery or fields of OpenFlow messages based on their message type and individual message fields. For instance, a strategy may be to duplicate `features_reply` messages 10 times or to modify the `in_port` field of a `packet_out` message to 7. Our malicious proxy supports all manipulations discussed in Section 2, including dropping, duplicating, and delaying messages based on message type, as well as modifying message field values based on message type.

Our testing applies this procedure to a list of strategies that we automatically generate based on the OpenFlow message formats and semantics associated with their fields. This list of strategies includes our message delivery attacks for each message type and manipulations of each field in each message type. The field values we use in our field manipulations are based on the field type, and are chosen to be likely to cause unexpected behavior. This includes setting values to zero, and the minimum and maximum values that the field can handle. For fields representing switch ports, we also consider all real switch ports as well as OpenFlow virtual ports like `CONTROLLER`. When selecting strategies to test, we only consider strategies for message types that we observe actually occurring in communication between the switch and controller. Because controllers usually do not use all the messages detailed in the OpenFlow specification, this dramatically reduces the number of strategies we need to test.

For malicious hosts, we consider the injection of ARP packets as discussed in Section 2. We again define a list of malicious hosts at the start of each test. We use time-based attack injection with these malicious hosts, where we launch attacks for a few seconds at different times during each test. The exact time of attack, duration, and frequency of packet injection are configurable.

This automatic strategy generation enables us to quickly and easily generate tens of thousands of strategies in a manner that considers both message structure and protocol semantics. In contrast, DELTA [24] uses blind fuzzing, supplemented with a few manual tests. It is able to generate an unbounded number of strategies, but considers neither message structure nor protocol semantics. OFTest and FLORENCE make no attempt at strategy generation and rely on a manually developed set of tests. As a result, FLORENCE only tests 18 different scenarios and OFTest only covers a few hundred. BEADS is able to generate several orders of magnitude more tests in a fraction of the time. While number of test cases does not perfectly correspond with amount of search space covered, this does strongly suggest that BEADS can cover a much larger portion of the search space, especially in combination with the new attacks that we find.

3.4 Impact Assessment

Once we have executed a strategy, we automatically determine the impact based on a variety of system and network characteristics. Our framework collects sev-

eral outputs and, at the end of each test, checks them for conditions indicating a deviation from normal behavior and a possible vulnerability. If such conditions are detected, we automatically schedule a re-test of the strategy to make sure that the failure is repeatable. If it is, we declare this strategy to be a vulnerability.

We use four methods to determine if a tested strategy leads to unexpected behavior: (1) OpenFlow error messages, (2) network configuration changes, (3) network reachability failures, and (4) controller or switch resource usage.

Since BEADS aims to identify actions that are the most damaging to the network, we gradually filter actions based on their impact. First, we consider the error messages, then the static network state, then the network connectivity, and finally the controller and switch resource usage. Below we provide more details about each of these and the rationale for considering them.

OpenFlow Error Messages. One mechanism we use to observe protocol deviation is monitoring the OpenFlow connections for error messages. Error messages are sent when an OpenFlow device (switch or controller) fails to parse an OpenFlow message or the message indicates an invalid or unsupported option. These messages indicate an anomalous condition in the OpenFlow connection, and that some desired change to the network was not performed.

Network State. One of the most powerful indicators of undesirable changes in the network are changes in the network state, including changes to routing, access control lists (ACLs), and priorities of flows in the network. We define *network state* as the state of the flow rules at each switch. The manager collects the flow rules from all switches at the end of each test, canonicalizes them, and compares them to reference flow rules from a benign run.

Unfortunately, the network state is not completely deterministic. Part of our canonicalization process filters out known non-deterministic elements (timestamps, etc). Additionally, we use multiple benign test runs to detect other non-deterministic elements and filter them out. Note that without detailed knowledge of the application algorithms the SDN controller has implemented we cannot decide whether a given non-deterministic rule is correct.

Reachability. Another protocol deviation indicator we use is pair-wise connectivity tests. In particular, our test system uses pair-wise ICMP pings and `iperf` to verify that all hosts are reachable from all other hosts. This is extremely effective in detecting spoofing attacks and attacks on connectivity. It also detects many manipulations of flow rules that our network state detection mechanism cannot detect due to non-determinism.

Controller or Switch Resource Usage. The final protocol deviation indicator we use is monitoring the RAM and CPU time used by the SDN controller and switches. Excessive usage, compared to a benign baseline, indicates an opportunity for denial of service, where the the ability of the switches or controller to process messages and packets in a timely fashion is impaired.

3.5 Implementation

We use KVM for virtualization and Mininet for the emulated network. The hosts were written in Python and use `iperf` and `ping` for traffic generation, and the

scapy library[4] for malicious host attack injection. The hosts communicate with a *HostController* Python script to execute malicious host attacks, generate traffic in the network, and conduct reachability tests on the network.

We insert our malicious proxy into the path between the Open vSwitch[5] software switches started by Mininet and our SDN controllers by simply having the proxy listen for TCP connections on a specified port and address, and supplying this port and address to Mininet as the address of the controller. When a switch connects, the proxy opens a second TCP connection to the controller and passes messages back and forth, modifying the message as required by the strategy. The proxy is implemented in C++ and leverages the C version of the Loxigen[6] library to parse and modify OpenFlow messages.

4 Experimental Results

We present the results obtained by applying BEADS to four SDN controllers: ONOS, POX, Ryu, and Floodlight. We then demonstrate the impact of these bugs with 4 real attacks.

4.1 Methodology

We applied BEADS to ONOS 1.2.1, POX version eel[7], Ryu 3.27, and Floodlight 1.2. For ONOS we used its default forwarding, which uses topology detection and shortest path routing along with proxy ARP, and a flow rule idle time of 30 seconds; for POX, we used the `proto.arp_responder`, `openflow.discovery`, `openflow.spanning_tree`, and `forwarding.l2_multi` modules to enable topology detection and shortest path routing along with proxy ARP; for Ryu, we used the `simple_switch` module, which emulates a network of learning switches; for Floodlight, we used its default forwarding, which uses topology detection and shortest path routing, and a flow rule idle time of 90 seconds.

The emulated network was created using Mininet 2.2.1 and Open vSwitch 2.0.2. While BEADS supports OpenFlow versions 1.0-1.5, our testing was done with OpenFlow 1.0 because it was the default negotiated by Open vSwitch and none of the SDN controllers we tested make use of the additional features introduced in later versions of OpenFlow. We configured Mininet with a simple two-tier tree topology of three switches and four hosts. The malicious switches and hosts vary depending on the test being run.

Our testing was done on a hyper-threaded 20 core Intel Xeon 2.4 GHz system with 125GB of RAM. Each test takes about 60 seconds. We parallelize the tests by running between 2 and 6 managers simultaneously. Testing required around 200 hours of total computation per tested SDN controller.

[4] `http://www.secdev.org/projects/scapy/`
[5] `http://openvswitch.org/`
[6] `https://github.com/floodlight/loxigen`
[7] Commit `4ebb69446515d9d9a0d5a002243cdca3c411520b` from 9/24/2015

Table 1 presents a summary of tested scenarios and bugs found. We tested 6,996 strategies for ONOS, 4,286 for POX, 3,228 for Ryu, and 4,330 for Floodlight. Not all the controllers take advantage of the complete functionality of OpenFlow, and as we test only the messages that are actually used by the tested system, the number of testing scenarios for each controller depended on the implemented and used OpenFlow functionality. As a result, we tested significantly more strategies for ONOS because ONOS automatically polls every switch for statistics about flow rules and ports periodically using the OpenFlow `flow_stats_*` and `port_stats_*` messages. The other controllers do not poll for statistics and so have no need to use these message types, effectively utilizing a much smaller portion of the OpenFlow protocol. Similarly Ryu's learning switch behavior requires no topology detection which reduces the number of messages it uses. We found a total of 831 unique bugs, with 178 common to all four controllers and a further 134 common to two or three controllers. Table 1 also shows the detection criteria (Section 3.4) for each bug.

Table 1: Summary of tested scenarios and bugs.

SDN Controller	**Total Tested**	**Bugs Found**	**Error Msg.**	**Net. State**	**Reach-ability**	**Res. Usage**
ONOS	6,996	578	104	372	102	0
POX	4,286	487	121	335	29	2
Ryu	3,228	251	48	168	32	3
Floodlight	4,330	577	95	478	4	0
Total	18,840	1,893	368	1,353	167	5

4.2 Detailed Results

We analyze all 831 unique bugs, based on their outcome, and present a summary in Table 2.

OpenFlow Operation Stall (OS)–No Known Mitigations. Several bugs have the common outcome of preventing or delaying OpenFlow operations that may affect multiple switches. By ignoring or dropping `barrier_request` and `barrier_reply` messages or changing their transaction IDs, a malicious switch can stall the installation of flow rules forming a path through that switch as we discovered in POX. A similar operation stall occurs when dropping or ignoring `flow_add` messages; the flow will eventually be inserted, but it will take extra messages and controller processing. These bugs are due to the design of OpenFlow and there are no known mitigations.

Periodic Switch Disconnect (SD)–Some Mitigations. We found many bugs that cause the malicious switch to periodically disconnect. This causes topology churn and prevents the installation of flow rules or the delivery of `packet_in`/`packet_out` messages. It takes about 3 seconds for the network to fully recover from one of these events, although the TCP level disconnection is only about half a second.

While most of these bugs are unavoidable and due to the reception of invalid OpenFlow messages, we did identify two subcategories of these bugs that can

Table 2: Discovered bugs, each line corresponds to several bugs grouped by message and action. Note that some bugs may occur multiple times, in different categories for different controllers. FL=Floodlight.

Outcome	**Name**	**Strategy**	**Num**	**Controllers**	**New**
OpenFlow operation stall	OS1	Drop barrier messages	4	POX	No
	OS2	Change xid in barrier messages	12	POX	No
	OS3	Drop flow_add	3	ALL	No
Periodic switch disconnect	SD1	Change version,type,length fields of handshake messages	197	ALL	No
	SD2	Duplicate handshake messages	20	ONOS	Yes
	SD3	Change version,type,length of barrier_request/barrier_reply	36	ONOS/POX/FL	No
	SD4	Change version,type,length in flow_add/flow_delete/flow_removed	48	ALL	No
	SD5	Change version,type,length in packet_in/packet_out	46	ALL	No
	SD6	Change version,type,length in port_mod/echo_reply/echo_request	42	POX/RYU/FL	No
	SD7	Change version,type,length in of_*_stats_reply/of_*_stats_request	68	ONOS	No
	SD8	Change role in of_nicira_controller_role_*	12	ONOS/FL	No
	SD9	Add CONTROLLER port to features_reply/port_status	15	ONOS/POX/FL	Yes
Data-plane loss	DP1	Delay/drop packet_in/packet_out	17	ALL	No
	DP2	Mod buffer_id in pkt_in/flow_add	8	ALL	No
Flow rule modification	FM	Change flow rule match, actions, etc in flow_add	162	ALL	No
Port config modification	PC	Change port_mod to change port configuration	39	POX	No
Packet location hijacking	LH1	Change port where packet was received in packet_in	14	ALL	No
	LH2	Change port for packet_out	14	ALL	No
Empty packet_ins	EP1	Change inner packet length to 0 in packet_in	1	ONOS	Yes
	EP2	Set packet_in length=0	2	POX/RYU/FL	No
Controller DoS	CD1	Delay flow_add	2	POX	No
	CD2	Change length to 0 on any message	8	RYU	Yes
	CD3	Change inner packet length to 0 in packet_in	2	RYU	Yes
Link detection failure	LD	Change port lists in features_reply/port_status	33	ONOS/POX/FL	Yes
Broken ARP broadcast	BB	Change port lists in features_reply/port_status	20	ONOS/FL	Yes
Unexpected flowrule removal	FR1	Change flow_stats_reply such that flow rule entry does not match	8	ONOS	Yes
	FR2	Change flow_stats_reply such that packet_count is constant	4	ONOS	Yes
Unexpected broadcast	UB	Change port_in field of the packet_out message	6	POX/RYU	Yes

be easily fixed. The first of these, consists of duplication of ONOS handshake messages. The state machine ONOS uses to control its handshake with a switch is not tolerant to message duplication. As a result, duplicating these messages results in a connection reset. This could be avoided by designing the handshake state machine to tolerate duplication.

The second subcategory of these bugs operates by modifying the `features_reply` message sent during the initial handshake to ONOS or POX to include a port with number 0xFFFD. This triggers a disconnection by the malicious switch the next time an ARP flood occurs, which might be hours later. The disconnection occurs because this port number (in OpenFlow 1.0) indicates the controller and results in an invalid `packet_in` being sent to the controller. These bugs can be mitigated by modifying the controller to sanity check the list of ports received from the switch.

Data-Plane Loss (DP)–No Known Mitigations. While we do not explicitly consider data-plane level attacks, we found several bugs which can trigger data-plane packet loss. All the controllers we tested are vulnerable to dropping occasional data-plane packets as a result of malicious switches discarding `packet_in` or `packet_out` messages. A different method to induce data loss is to target the buffering of packets at malicious OpenFlow switches by corrupting the buffers indicated in `packet_in` or `packet_out` messages. This causes the buffered packet to eventually be dropped. These bugs can have particularly large impacts on small flows like ARP and DNS where installing flow rules makes little sense. We are not aware of any known mitigations against these bugs.

Flow Rule Modification (FM)–No Known Mitigations. Another class of bugs disrupts flow rules from the controller by modifying `flow_add` messages. This enables the attacker to affect the timeout, priority, and match fields and masks of flow rules in malicious switches as well as the actions performed on a match. Our testing found a number of modifications that cause network-wide denial of service, but specific changes to small sets of flows are also possible. We are not aware of any known mitigations against these bugs.

Port Config Modification (PC)–No Known Mitigations. Similar to the flow rule modification, a compromised switch can mislead a controller as to the configuration of its ports by modifying `port_mod` messages. This configuration primarily consists of the port's enabled or disabled state and whether it has broadcast enabled. Our testing found a number of specific modifications that cause broad, network-wide denial of service, but these bugs could also be used for specific modifications targeting specific topology changes in networks. We are not aware of any known mitigations against these bugs.

Packet Location Hijacking (LH)–No Known Mitigations. Several bugs allow a malicious switch to change the apparent source port of a packet sent to the controller and the apparent destination port of packets send by the controller. This hijacking of packet locations has dramatic and wide spread effects across the network, including topology detection, MAC learning, and reactive forwarding. Note that the topology poisoning attacks identified in prior efforts [7, 10] apply these bugs to LLDP traffic on particular ports to carefully

forge specific links without breaking the entire network. While attacks forging LLDP packets can be mitigated using cryptographic techniques, the more general bugs are more difficult to address, and we are not aware of any known mitigations.

Empty `packet_in`'s (EP)–Some Mitigations. We identified a bug in the ONOS controller where sending a `packet_in` message with a zero-length payload packet triggers a NULL pointer exception in the processing thread. ONOS's design separates the processing of messages from different switches into different threads. As a result, this exception causes this switch's to terminate, disconnecting the malicious switch, but allows the controller to continue running. We reported this bug to the ONOS project, which confirmed it and verified that it was no longer present in their most recent release.

However, a second bug exists which effectively prevents all topology detection and useful reactive forwarding through a compromised switch on any controller. The bug is exploited by configuring the compromised switch to send `packet_in` messages with a payload length of at most zero bytes. This means that no packet headers will be sent to the controller, which can then do nothing useful with the message, preventing topology detection, MAC learning, and reactive forwarding. Preventing these bugs would require an update to the OpenFlow specification to disallow very small payload lengths.

Controller DoS (CD)–Some Mitigations. We identified several possible bugs that can overload and DoS the controller. One unavoidable way to do this is simply to delay the installation of flow rules in malicious switches, causing a flood of `packet_in` messages. This bug has been identified by several other studies, including [7, 42, 44]. Note that ONOS and Floodlight partially mitigate this bug by tracking flow rules to prevent repeated insertion attempts. The only complete mitigation is to proactively insert all needed flow rules and never send packets to the controller.

We also identified two new bugs that crash the Ryu controller. The first of these causes an infinite loop when receiving an OpenFlow message with a zero-length header while the second terminates the controller with an uncaught exception when a `packet_in` message with a zero length payload is received. We reported these bugs to the Ryu project, which has patched both.

Link Detection Failure (LD)–Some Mitigations. This bug works against implementations of the LLDP protocol to prevent a correct global topology from being constructed by a vulnerable controller. It exists in ONOS, Floodlight, and POX; Ryu is not vulnerable only because it does not attempt to construct a global view of the topology, but simply emulates a set of learning switches. Link detection is typically implemented by having the controller send LLDP packets out of each port on each switch that it knows about and observing where the `packet_in` messages containing those packets arrive. From the `packet_in` message, the controller knows what port the packet was received on, allowing it to identify a unidirectional link between the port where this packet was sent and the port where it was received.

This bug tampers with the list of ports sent by a malicious switch in the the `features_reply` and `port_status` messages that the controller uses to enumerate available switch ports. If ports are omitted in these messages, no LLDP packets will be sent on them, which means no links can form from those ports. Without knowledge of these links, the controller is limited in its ability to route packets and may be unable to reach certain destinations.

These bugs can be substantially mitigated by monitoring received `packet_in` messages and looking for previously unknown ports. If such ports are observed, the controller can begin to send LLDP packets on those ports and emit an alert about a malicious or buggy switch sending inconsistent information.

Broken ARP Broadcast (BB)–Some Mitigations. This bug is conceptually similar to the link detection failure bug except that it applies to the network edge ports of a malicious switch that are directly connected to hosts instead of to other switches. It enables an attacker to render target hosts unreachable in a network running ONOS or Floodlight. Both controllers identify edge ports as those that have not received LLDP packets and are thus not connected to other switches and only broadcast ARP requests on these ports. However, by relying solely on the port lists from `features_reply` and `port_status` messages, certain ports may be omitted from those messages and hidden from the controller, preventing ARP broadcasts on those ports. This is despite other traffic from those ports. This causes hosts behind these omitted ports of malicious switches to be effectively unreachable. This lasts until each target host sends an ARP request of its own, at which point the controller receives the ARP request and learns the location of the target host. Much like link detection failure bugs, monitoring received `packet_in` messages can substantially mitigate these bugs.

Unexpected Flow Rule Removal (FR)–Complete Mitigations. These bugs confuse the ONOS controller into removing flows that it installed on a malicious switch, complicating debugging and directing suspicion away from the malicious switch. This bug occurs because ONOS manages the flow rules in switches with a very heavy hand. In particular, it will remove any flow rule in the switch that it did not insert and will track the usage of flow rules and request removal of flows rules that have been idle for some amount of time. As a result, by modifying the flow rule information returned to ONOS in the `flow_stats_reply` message, a malicious switch can make a flow rule appear idle or appear sufficiently different that ONOS does not recognize it and orders its removal. These bugs can be mitigated by relying on the ability of OpenFlow switches to automatically remove flow rules based on idle timeouts [31, 33] and ensuring that all expected rules are accounted for before beginning removal.

Unexpected Broadcast Behavior (UB)–Partial Mitigations. OpenFlow `packet_out` messages include a special broadcast option that asks a switch to broadcast the included packet out of all ports with broadcast enabled that are not the port on which this packet was received. However, this mechanism is vulnerable to subtle changes in behavior that cause unexpected packet forwarding and cripples learning-switch type routing. This bug occurs when the `packet_out` message is modified by a malicious switch to change the `in_port`, which results

in the packet being broadcast by the malicious switch out of the port on which it was received. This has impact on learning switch routing because broadcasting packets in this manner causes switches to learn incorrect locations for hosts resulting in connectivity losses. These bugs can be detected by linking packets sent at one switch with those received by other switches.

4.3 Attack Demonstrations

We demonstrate that one can weaponize the bugs in Table 2 into powerful attacks with potentially disastrous consequences. We manually develop exploits for a few of the bugs we discover and present these weaponized examples below. All attacks were manually implemented and tested using BEADS. The network topology was a simple tree with three switches and four hosts.

TLS Man-in-the-Middle. The security of TLS against man-in-the-middle attacks relies on a correctly implemented certificate-based PKI and active user involvement. Unfortunately, attackers can leverage maliciously obtained certificates [22] or tools like SSLStrip [26] to observe (and potentially modify) confidential information exchanged between client and server.

We implemented this scenario using the Ryu controller, which provides learning switch routing. We assume that the attacker has access to a compromised switch on path as well as a host that is not currently on the path between client and server. We use the FM bug to alter the flow table of the attacker-controlled switch to insert his host, potentially performing an SSL man-in-the-middle attack, into the path between the target client and server. Additional rules must be inserted using the FM vulnerability to ensure that each switch only sees packets with addresses that conform to the network topology.

Web Server Impersonation. In this scenario, an attacker wishes to impersonate an internal web server. We use the ONOS controller (we believe POX is vulnerable to a similar attack) and a malicious host at an arbitrary location in the network. We used the ARP-location-injection bug to confuse the controller into believing that the target webserver is now located on the same port as the attacker. All future connections from new or idle hosts are then sent to the attacker. Since ONOS uses a global Proxy ARP cache, the attacker can be anywhere on the network. This effect lasts until the target server starts a new connection with a host that causes a `packet_in` to the controller. This will reset the target server's location and end the attack.

If the switch to which the target server is connected is compromised, the attacker can increase the duration of this attack by also using the DP1 vulnerability to drop all `packet_in` messages from the target to the controller. This prevents the target server from ever re-asserting its old location and causes the attack to last indefinitely.

Break Network Quarantine. This scenario considers an attacker who has found useful information (*e.g.,* PII, credit card data, intellectual property, *etc.*) but induced a network quarantine in the process, and must transfer that data to an external server despite the imposed isolation.

In our demonstration of this attack, the Ryu controller is implementing a firewall and attempting to quarantine a target host from the rest of the network by dropping all packets from its port. The attacker is assumed to control an arbitrary switch in the network and is trying to send traffic from the target host to elsewhere in the network to exfiltrate discovered data. We use the CD2 bug for this attack, which causes the controller to enter an infinite loop and become unresponsive. Eventually, the switches in the network detect the failed connection and enter standalone mode, at which point they fall back to conventional Layer-2 Ethernet learning switches. This purges the flow table and enables all-to-all connectivity, allowing the attacker to exfiltrate the data. We were able to successfully demonstrate this attack against Ryu.

Deniable Denial of Service. In this scenario an adversary wishes to degrade network performance while remaining undetected for as long as possible. Whole-network effects such as controller crashes are thus undesirable, as are any actions that are easily traceable to attacker-controlled entities.

We implemented this attack scenario using ONOS. To stealthily disrupt network service, we use an infinite sequence of SD9 bugs. This bug uses a malformed `features_reply` message to cause disconnection of the malicious switch on the next ARP flood, which may be a long time after the message was sent. Blame for the disconnection will be placed on the controller because of the invalid `packet_out` message that triggers the disconnection, thereby directing suspicion away from the malicious switch. Using this attack, each ARP flood caused the malicious switch to disconnect from the controller, resulting in about 3 seconds of impaired service. ARP floods occurred 4 times in our tests, but an attacker could use normal ARP requests for non-existent hosts to increase that by a factor of 10. We successfully tested this attack against ONOS.

5 Discussion and Limitations

BEADS does not find fully weaponized attacks ready to launch against a target. Instead, it identifies strategies that cause significant impact on the network stemming from one or more bugs much like stack-overflow vulnerabilities, there is still manual effort needed to write an exploit that uses the bug in a targeted way. This includes fixing the malicious host or switch locations, as the bugs themselves exist irrespective of network location.

Many of the bugs found by BEADS allow a malicious switch to impact other switches or hosts indirectly. While a malicious switch always has the ability to impact such devices directly, there are two reasons it might want to use indirect methods instead. First, it makes it difficult to identify the malicious party by making the controller appear responsible for the undesirable behavior. Second, if a switch does not protect its connection with the controller using TLS, these bugs allow a Man-In-The-Middle attacker to maliciously control the switch using OpenFlow alone. Prior work has established that a significant number of SDN switches are not using TLS to protect their communication with the controller, making this a promising attack avenue [35, 36].

Because BEADS is designed to detect bugs in the SDN control plane, we do not include metrics like latency, throughput, and packet drop rate in our detection. These are important data plane metrics, but provide little to no information about the control plane, and thus for our testing.

Our malicious proxy is stateless, and thus cannot coordinate modifications of particular requests or responses. Instead, it applies actions based on the type of each message. This maps well to OpenFlow's use of separate types for most requests, responses, and commands and reduces the attack generation search space. Adding additional state to this proxy could enable the discovery of more complex attacks, but at the cost of an exponential increase in the search space.

6 Related Work

Network testing and debugging. The work that is closest to ours is DELTA [24]. DELTA also evaluates the whole SDN system, including both controller and switches. However, it focuses on the SDN controller's northbound interface and uses only blind fuzzing without regard for message structure or probable vulnerabilities on the OpenFlow southbound interface. BEADS focuses on the southbound interface and uses message format and semantic information to provide much better test coverage, especially against controller algorithms like routing and topology detection. As a result, BEADS finds all the malicious switch attacks that DELTA finds and several that DELTA does not.

Other closely related efforts are OFTest [8] and FLORENCE [29]. Both of these tools test OpenFlow switches using manually written tests focusing on conformance to the OpenFlow specification. Since these tools do not consider the controller, they are unable to find bugs and attacks in the controller software based on incorrect assumptions about the switches.

Another work related to ours is NICE [6], which uses model checking and symbolic execution to test SDN controller applications using network invariants. NICE differs from our work in that it focuses on non-malicious SDN testing, while we focus on malicious attacks. NICE was only shown to scale to simple first-generation SDN controllers (*e.g.,* NOX). The second generation of SDN controllers we test, like ONOS and Ryu, include orders of magnitude more code, which would substantially complicate the symbolic execution. In particular, topology generation requires the controller to send messages to the switches on a timer which is not supported in NICE. BEADS successfully tests ONOS and other large, second-generation SDN controllers. Finally, while NICE models switches and hosts, our approach uses real (software) switches and real applications. OFTEN [20] (an extension of NICE) adds real switches, but it cannot test for performance attacks. Further, neither NICE nor OFTEN consider sending the switches malformed messages and both are dependent on difficult-to-design network state invariants for bug detection.

STS [41] is another work looking at network debugging. This work develops a method to minimize network execution traces containing bugs for OpenFlow networks. To test their trace minimizing technique, they develop a network event

fuzzer that randomly injects events like link failures or packets into a network and use it to find seven new bugs in five SDN controllers. Unlike the STS fuzzer, our work focuses on manipulating the OpenFlow messages themselves and identifies which of these messages are likely to lead to attacks.

Attacks and defenses in SDN. Work that studies SDN attacks includes exploration of protocol attacks [10], saturation attacks [42, 44], and controller-switch communication attacks [4]. Several defense and verification techniques have been proposed to ensure that flow rules do not violate invariants [1,2,16–18]. These verification approaches focus on logic errors in rules, as opposed to malicious manipulation of the SDN. The work by Mekky, *et al.* [28] allows efficient inspection and filtering of higher network layers in SDNs. Kotani and Okabe [19] filter `packet_in` messages according to predefined rules to protect the controller. LineSwitch [3] mitigates control plane saturation DoS attacks by applying probabilistic black-listing. Recently, Spiffy [14] was proposed to detect link-flooding DDoS attacks in SDNs by applying rate changes to saturated links. None of these approaches considers the problem of automatic attack identification.

7 Conclusion

We have developed a framework, BEADS, to automatically find attacks in SDN systems. BEADS considers attacks caused by malicious hosts or switches by using semantically-aware test case generation and considering the whole SDN system (switches, controllers, and hosts). We used BEADS to automatically test almost 19,000 scenarios on four controllers and found 831 unique bugs. We classified these into 28 categories based on their impact; 10 of which are new. We demonstrated through 4 attacks how an attacker can use these bugs to impact high-level network goals such as availability, network topology, and reachability.

Acknowledgements

We thank William Streilein and James Landry for their support of this work as well as our shepherd, Guofei Gu, and anonymous reviewers for their helpful comments on this paper. This material is based in part upon work supported by the National Science Foundation under Grant Numbers CNS-1654137 and CNS-1319924.

References

1. Al-Shaer, E., Al-Haj, S.: FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In: Proc. of ACM SafeConfig. pp. 37–44 (2010)
2. Al-Shaer, E., Marrero, W., El-Atawy, A., Elbadawi, K.: Network configuration in a box: Towards end-to-end verification of network reachability and security. In: Proc. of ICNP. pp. 123–132 (2009)

3. Ambrosin, M., Conti, M., De Gaspari, F., Poovendran, R.: LineSwitch: efficiently managing switch flow in software-defined networking while effectively tackling DoS attacks. In: Proc. of ASIA CCS. pp. 639–644 (2015)
4. Benton, K., Camp, L.J., Small, C.: OpenFlow vulnerability assessment. In: Proc. of HotSDN. pp. 151–152 (2013)
5. Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W., et al.: ONOS: towards an open, distributed SDN OS. In: Proc. of HotSDN. pp. 1–6 (2014)
6. Canini, M., Venzano, D., Peresini, P., Kostic, D., Rexford, J.: A NICE way to test OpenFlow applications. In: Proc. of NSDI (2012)
7. Dhawan, M., Poddar, R., Mahajan, K., Mann, V.: SPHINX: detecting security attacks in software-defined networks. In: Proc. of NDSS (2015)
8. Floodlight Project: Github - floodlight/oftest: Openflow switch test framework (2016), `https://github.com/floodlight/oftest`
9. Foster, N., Harrison, R., Freedman, M.J., Monsanto, C., Rexford, J., Story, A., Walker, D.: Frenetic: A network programming language. In: ACM SIGPLAN Notices. vol. 46, pp. 279–291 (2011)
10. Hong, S., Xu, L., Wang, H., Gu, G.: Poisoning network visibility in software-defined networks: New attacks and countermeasures. In: Proc. of NDSS. pp. 8–11 (2015)
11. Jafarian, J.H., Al-Shaer, E., Duan, Q.: OpenFlow random host mutation: transparent moving target defense using software defined networking. In: Proc. of HotSDN. pp. 127–132 (2012)
12. Jero, S., Lee, H., Nita-Rotaru, C.: Leveraging state information for automated attack discovery in transport protocol implementations. In: 45th IEEE/IFIPDSN. pp. 1–12. IEEE Computer Society (2015)
13. Kampanakis, P., Perros, H., Beyene, T.: SDN-based solutions for moving target defense network protection. In: Proc. of WoWMoM (2014)
14. Kang, M.S., Gligor, V.D., Sekar, V.: SPIFFY: Inducing cost-detectability tradeoffs for persistent link-flooding attacks. In: Proc. of NDSS (2016)
15. Katta, N.P., Rexford, J., Walker, D.: Logic programming for software-defined networks. In: Workshop on Cross-Model Design and Validation (XLDI). vol. 412 (2012)
16. Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., Whyte, S.: Real time network policy checking using header space analysis. In: Proc. of NSDI. pp. 99–111 (2013)
17. Kazemian, P., Varghese, G., McKeown, N.: Header space analysis: Static checking for networks. In: Proc. of NSDI. pp. 113–126 (2012)
18. Khurshid, A., Zhou, W., Caesar, M., Godfrey, P.B.: Veriflow: Verifying network-wide invariants in real time. In: Proc. of NSDI (2013)
19. Kotani, D., Okabe, Y.: A packet-in message filtering mechanism for protection of control plane in OpenFlow networks. In: Proc. of ANCS. pp. 29–40 (2014)
20. Kuzniar, M., Canini, M., Kostic, D.: OFTEN testing OpenFlow networks. In: European Workshop on Software Defined Networking (EWSDN). pp. 54–60 (2012)
21. Lantz, B., Heller, B., McKeown, N.: A network in a laptop: Rapid prototyping for software-defined networks. In: Proc. of HotNets (2010)
22. Leavitt, N.: Internet security under attack: The undermining of digital certificates. Computer 44(12), 17–20 (2011)
23. Lee, H., Seibert, J., Hoque, E., Killian, C., Nita-Rotaru, C.: Turret: A platform for finding attacks in unmodified implementations of intrusion tolerant systems. In: IEEE ICDCS (2014)

24. Lee, S., Yoon, C., Lee, C., Shin, S., Yegneswaran, V., Porras, P.: DELTA: a security assessment framework for software-defined networks. In: Network and Distributed System Security Symposium. Internet Society (2017)
25. Lim, S., Ha, J.I., Kim, H., Kim, Y., Yang, S.: A SDN-oriented DDoS blocking scheme for botnet-based attacks. In: Proc. of ICUFN. pp. 63–68 (2014)
26. Marlinspike, M.: New tricks for defeating SSL in practice. BlackHat DC, February (2009)
27. McCauley, M.: About POX (2013), http://www.noxrepo.org/pox/about-pox/
28. Mekky, H., Hao, F., Mukherjee, S., Zhang, Z.L., Lakshman, T.: Application-aware data plane processing in SDN. In: Proc. of HotSDN. pp. 13–18 (2014)
29. Natarajan, S.: Github - snrism/florence-dev: Sdn security test framework (2016), https://github.com/snrism/florence-dev
30. Nelson, T., Ferguson, A.D., Scheer, M.J., Krishnamurthi, S.: Tierless programming and reasoning for software-defined networks. In: Proc. of NSDI. pp. 519–531 (2014)
31. Open Networking Foundation: OpenFlow switch specification (1.0) (2009)
32. Open Networking Foundation: Conformance test specification for OpenFlow switch specification 1.0.1 (2013), https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-test/conformance-test-spec-openflow-1.0.1.pdf
33. Open Networking Foundation: OpenFlow switch specification (1.5.0) (2014)
34. Open Networking Foundation: Conformance test specification for OpenFlow switch specification 1.3.4 - basic single table conformance test profile (2015), https://www.opennetworking.org/images/stories/downloads/working-groups/OpenFlow1.3.4TestSpecification-Basic.pdf
35. Pickett, G.: Abusing software defined networks. In: Defcon (2014)
36. Pickett, G.: Staying persistent in software defined networks. In: BlackHat (2015)
37. Plummer, D.: Ethernet address resolution protocol: Or converting network protocol addresses to 48.bit ethernet address for transmission on ethernet hardware. RFC 826 (1982)
38. Porras, P., Cheung, S., Fong, M., Skinner, K., Yegneswaran, V.: Securing the software-defined network control layer. In: Proc. of NDSS (2015)
39. Project Floodlight: Floodlight OpenFlow Controller (2016)
40. Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., Walker, D.: Abstractions for network update. In: Proc. of ACM SIGCOMM. pp. 323–334 (2012)
41. Scott, C., Wundsam, A., Raghavan, B., Panda, A., Or, A., Lai, J., Huang, E., Liu, Z., El-Hassany, A., Whitlock, S., Acharya, H., Zarifis, K., Shenker, S.: Troubleshooting blackbox SDN control software with minimal causal sequences. In: Proc. of SIGCOMM. pp. 395–406. ACM (2014)
42. Shin, S., Gu, G.: Attacking software-defined networks: A first feasibility study. In: Proc. of HotSDN. pp. 165–166 (2013)
43. Shin, S., Porras, P., Yegneswaran, V., Gu, G.: A framework for integrating security services into software-defined networks. In: Proc. of open networking summit (2013)
44. Shin, S., Yegneswaran, V., Porras, P., Gu, G.: Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In: Proc. of CCS. pp. 413–424 (2013)
45. The Ryu Project: Ryu SDN framework using OpenFlow 1.3. Website (2014), https://osrg.github.io/ryu/