A PRACTICAL FRAMEWORK FOR FINDING SOFTWARE VULNERABILITIES IN SDN CONTROLLERS

by

WALID SHARIF

B.A., Brandeis University, 2013

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirement for the degree of

Master of Science

Department of Telecommunications

2017

This thesis entitled:
A Practical Framework for Finding Software Vulnerabilities in SDN Controllers
written by Walid Sharif
has been approved for the Department of Telecommunications

Levi Perigo

Joe McManus

Charles Cook

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Sharif, Walid (M.S., Telecommunications)

A Practical Framework for Finding Software Vulnerabilities in SDN Controllers

Thesis directed by Dr. Levi Perigo.

Software-defined networking (SDN) has the potential to greatly reduce the cost and increase the manageability of large networks. However, there are multiple security concerns holding back its wide-scale adoption. While previous research has mainly examined securing the data and application planes of SDN, we argue that the controller itself is the most vulnerable component in the SDN architecture because it is both the most central and the most software-reliant component. Therefore, research into better securing the controller is central to any effort at securing the SDN architecture. This study examines the question of how to better secure the controller by developing a practical framework for finding vulnerabilities in the underlying software of the OpenDaylight controller. By finding vulnerabilities in its software, we aim to not only improve the security of the controller software, but also build a foundation to allow previous research to implement better solutions to secure other SDN components.

DEDICATION

This thesis is dedicated to my mother, Tabassum Taqi, my partner, Katharine Cohen, and my brother, Umair Sharif, whose constant support made it possible.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Dr. Levi Perigo, for the time, guidance, and feedback that made this thesis possible. His expertise in Software-defined Networking provided an excellent foundation for my research.

CONTENTS

CHAPTER I

INTRODUCTION

I. Problem Setting

Despite the obvious cost-savings and improvements in manageability, many data centers are reluctant to switch from traditional to software-defined networking (SDN). [1] One of the main reasons is concerns about security; recent surveys demonstrate that a significant percentage of Information Technology (IT) professionals both believe that SDN will make networks less secure and have major concerns about the security of open-source SDN technology.[2,3] While the benefits of SDN derive from the properties of increased software-reliance and centralization, so do the security risks. [4] Increased software-reliance leads to a higher chance of compromise because it is difficult to either guarantee that software is defect-free or identify and fix every vulnerability within a piece of software. Likewise, increased centralization leads to a higher chance of compromise because if a single network component is compromised in a centralized system, it is more likely that the entire network is compromised.

The controller is the most software-reliant and centralized component of the SDN architecture, and therefore it is the most vulnerable.[5] While previous research has addressed important issues related to the security of the data and application planes, there is a lack of research relating to the security of the controller itself. This area of study is important because it is difficult to secure the SDN architecture without first securing the controller. For example, a

[1] Mark Leary, "SDN, NFV, and open source: the operator's view," Gigaom, March 19, 2014, accessed April 1, 2017, https://gigaom.com/report/sdn-nfv-and-open-source-the-operators-view/.

[2] Jim Metzler, "Understanding Software-Defined Networks," InformationWeek, October 2012, accessed April 1, 2017, https://www.necam.com/docs/?id=28a3203e-ef17-4f0e-b193-0edc4eb065cc.

[3] Leary, "SDN, NFV, and open source: the operator's view," 2014.

[4] Diego Kreutz, Fernando M. V. Ramos, and Paulo Verissimo, "Towards secure and dependable software-defined networks," *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*, 2013.

[5] Sakir Sezer et al, "Are we ready for SDN? Implementation challenges for software-defined networks," *IEEE Communications Magazine* 51, no. 7 (2013): 36-43.

system to secure communications between switches and the controller or applications and the controller is not of much use if a software vulnerability in the controller allows an attacker to execute arbitrary code on it. It is also important because studying how to better secure the controller will lead to improvements in securing other components; when we understand how to secure the most central component in a system, it is likely that that knowledge can be used to secure more peripheral components. Therefore, research into securing the controller will enhance the body of knowledge (BoK) because 1) it will improve the security of the most vulnerable component in the SDN architecture, 2) it will reinforce solutions for securing the data and application planes yielded by previous research, and 3) it will lead to progress on securing more peripheral components.

II. Research Question

The central question we are examining is: how can we improve the software security of the controller in a SDN architecture? We propose that this can be done in measurable terms by developing a framework for discovering vulnerabilities in a widely-used controller like OpenDaylight. Our research question is broken down into three sub-problems, summarized by Table 1.

Sub-problem	Hypothesis
What is the most effective methodology for finding vulnerabilities in the controller?	Fuzzing
How can this methodology be applied?	Smart, generation-based, blackbox fuzzer created with Python modules
How does the controller respond to this methodology?	Unexpected behavior observed through CPU usage, memory usage, and log output

Table 1: Sub-problems and Hypotheses.

III. Research Sub-problems

A. What is the most effective methodology for discovering vulnerabilities in the controller?

This sub-problem is important to examine because it lays the foundation for the rest of the research. We are more likely to find vulnerabilities in the controller, and hence improve its security, by using an effective vulnerability discovery method. Our hypothesis is that fuzzing, as opposed to static analysis, which requires analysis of the target program's source code, is the most effective method for vulnerability discovery in a large and complex software system like the OpenDaylight.

Although the source code for OpenDaylight is freely available and can be analyzed for possible vulnerabilities, doing so is difficult for two reasons. First, analysis of the OpenDaylight source code requires a large time investment. Figure 1 shows that the entire OpenDaylight code base currently consists of 2222 Java classes.

```
$ wget https://github.com/opendaylight/controller/archive/master.zip
$ unzip master.zip
$ find controller-master -type f -name *.java | wc -l
    2222
```

Figure 1: Number of Java Classes in OpenDaylight's Code Base.

Analysis of these classes is bound to require a substantial amount of time.

Second, analysis of the OpenDaylight source code requires in-depth knowledge of the OpenDaylight architecture. The OpenDaylight project is open-source, which makes it easier to contribute to as a developer, but more difficult to understand all the connected subsystems; the OpenDaylight Wiki describes many aspects of the system well, but most of the information is geared towards making it easier for developers to extend the project by developing custom

modules.[6] To this end, most of the OpenDaylight Wiki describes the core controller code in high-level terms, making it difficult to understand exactly how the core functionality works at a low-level and therefore if any vulnerabilities exist.

For example, it is well-known that the java.util.Random class is not safe to use for cryptographic purposes.[7] Figure 2 shows that the java.util.Random library is imported by three Java classes in the OpenDaylight code base: DatastoreAbstractWriter.java, AbstractRaftActorBehavior.java, and LogGenerator.java.

```
$ grep -R "java.util.Random" controller-master | awk -F '/' '{print $NF}'
DatastoreAbstractWriter.java:import java.util.Random;
AbstractRaftActorBehavior.java:import java.util.Random;
LogGenerator.java:import java.util.Random;
```

Figure 2: Usage of java.util.Random in OpenDaylight's Code Base.

We know that OpenDaylight supports TLS usage.[8] However, it is not entirely clear how these three classes interact with the larger overall system and it is therefore difficult to discern whether they are used by OpenDaylight to support TLS.

B. How can this methodology be applied?

This sub-problem is important to examine because the existence of fuzzing techniques does not necessarily imply that they can be applied to or are even suited to vulnerability discovery with OpenDaylight. We will therefore examine different fuzzing techniques and determine which ones are best suited. Our hypothesis is that smart, generation-based, blackbox fuzzing through the use of various Python modules is the most applicable technique for fuzzing

[6] "OpenDaylight Project," OpenDaylight Project, accessed April 1, 2017, https://wiki.opendaylight.org.

[7] "Random (Java Platform SE 7)," Java Platform, Standard Edition 7 API Specification, accessed April 1, 2017, https://docs.oracle.com/javase/7/docs/api/java/util/Random.html.

[8] "OpenDaylight OpenFlow Plugin: TLS Support," OpenDaylight Project, accessed April 1, 2017, https://wiki.opendaylight.org/view/OpenDaylight_OpenFlow_Plugin:_TLS_Support.

various OpenDaylight features.

C. How does the controller respond to this methodology?

This sub-problem is important to examine because it is the defining sub-problem in determining if we can discover vulnerabilities in OpenDaylight and subsequently improve its security with our chosen methodology. Our hypothesis is that spikes in central processing unit (CPU) or memory utilization may indicate the presence of a vulnerability because these spikes indicate that OpenDaylight is unable to smoothly handle the fuzzy input sent to it by our chosen fuzzer. This may be because, for example, the controller is using the fuzzy input to make calculations for memory allocation, performing arithmetic operations on the fuzzy input, or is using an intensive algorithm that is not properly bounded.[9]

IV. Thesis Arrangement

The remainder of this thesis is arranged as follows. First, we review the literature in relation to 1) SDN security, in order to determine the extent of research into securing the controller and how our research can contribute to the BoK, 2) fuzzing in order determine the best methodology to use to fuzz OpenDaylight, and 3) OpenDaylight's architecture in order to understand how to apply our chosen methodology. We then apply our chosen methodology to two OpenDaylight features: odl-l2switch-all, a Northbound plugin, and odl-restconf-all, a Southbound plugin. Next, we discuss the results of our fuzzing runs. Finally, we conclude with a summary and suggestions for future research.

[9] Peter Oehlert, "Violating Assumptions with Fuzzing," *IEEE Security and Privacy Magazine* 3, no. 2 (2005): 58-62.

LITERATURE REVIEW

I. SDN Security

SDN relies on the abstraction of network functions from network hardware into software. This is achieved by separating traditional network functionality into three planes: data, control, and application. Kreutz et al. organize SDN security threats into seven categories.[10] We further reduce these threats into data, control, and application plane threats because most of the threats listed by Kreutz et al. can be classified as either a threat to a specific plane or a threat to traffic coming out of that plane. [11] This approach is summarized by Table 2.

Our Classification	Kreutz et al. Classification
Data Plane Threats	Forged or fake traffic flows (1)
	Attacks on vulnerabilities in switches (2)
	Attacks on control plane communication (3)
Control Plane Threats	Forged or fake traffic flows (1)
	Attacks on control plane communication (2)
	Attacks on and vulnerabilities in controllers (4)
Application Plane Threats	Forged or fake traffic flows (1)
	Attacks on control plane communication (3)
	Lack of mechanisms to ensure trust between the controller and management applications (5)

[10] Kreutz, "Towards secure and dependable software-defined networks," 2013.

[11] Ijaz Ahmad et al., "Security in Software Defined Networks: A Survey," *IEEE Communication Surveys and Tutorials* 17, no. 4 (2015): 2317- 2346.

	Attacks on and vulnerabilities in administrative stations (6)

Table 2: SDN Security Threats.

While there has been extensive research into data and application plane threats, the same cannot be said about control plane threats, especially as it relates to software vulnerabilities in the controller. We first examine previous research in regards to data, control, and application plane threats and then argue how our research can contribute to the BoK.

A. Data Plane Threats

An example of a data plane threat is an attacker exploiting a vulnerability in a switch's software that allows for arbitrary code execution.[12] The attacker may be able to use this vulnerability to launch a Distributed Denial of Service (DDoS) attack, exfiltrate valuable data, or stay hidden in the hopes of developing a permanent presence on the network, like many Advanced Persistent Threats (APTs).

A second example is the possibility of an attacker poisoning the network topology. An attacker can poison the Address Resolution Protocol (ARP) cache of other switches or controllers by forging packets that contain the hardware address of legitimate switches or controllers and tricking other SDN components into sending traffic its way. The attacker can then launch a Man-in-the-Middle (MitM) attack by extracting valuable data from packets or a Denial of Service (DoS) by dropping the packets, preventing them from reaching their destinations.[13] An attacker can also poison packets used by the Host Tracking Service, which

[12] Kreutz, "Towards secure and dependable software-defined networks," 2013.

[13] Li Dawei, Xiaoyan Hong, and Jason Bowman, "Evaluation of Security Vulnerabilities by Using ProtoGENI as a Launchpad," *2011 IEEE Global Telecommunications Conference - GLOBECOM 2011*, 2011.

tracks the location of hosts across the network, and the Link Discovery Service, which discovers links between switches. The former can result in the hijacking of the location of important network devices, like a server, and the latter can result in the creation of false links between switches that, in turn, can lead to a MitM or DoS attack. [14,15]

Hong et al. propose TopoGuard to combat the poisoning of the Host Tracking and Link Discovery Services by verifying the packets associated with them. Dhawan et al. propose SPHINX, which uses flow graphs to both approximate the actual network topology and validate all network updates. SPHINX first monitors all controller communication to identify the packets that are required to build a comprehensive network topology, including incoming OpenFlow packet headers, outgoing flow path setup directives, and actual flow traffic measurements over network links. SPHINX then constructs and verifies flow graphs to prevent the poisoning of the network topology.

A third example is the possibility of an attacker overwhelming the data plane. When a switch receives a new flow that does not have a match in its flow table, it must send the packet to a controller to resolve the query, which can cause high bandwidth utilization between the switch and controller, store the packet in memory until the flow table entry is returned, which can overflow switch memory, and store the resulting flow table entry in its flow table, which can become overloaded with entries. An attacker can therefore cause a DoS by producing a series of unique flow requests, perhaps through the use of distributed botclients, to saturate the data plane by either 1) causing congestion in the link between the switch and the controller, 2) causing an overflow in the switch's memory, or 3) causing an overflow in the switch's flow

[14] Sungmin Hong et al., "Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures," *Proceedings 2015 Network and Distributed System Security Symposium*, 2015. doi:10.14722/ndss.2015.23283.

[15] Mohan Dhawan et al., "SPHINX: Detecting Security Attacks in Software-Defined Networks," *Proceedings 2015 Network and Distributed System Security Symposium*, 2015. doi:10.14722/ndss.2015.23064.

table.[16,17,18,19,20,21]

Shin et al. propose a data plane extension called connection migration that can reduce the amount of interactions between the data and control plane.[22] Their solution allows the data plane to initiate a Transmission Control Protocol (TCP) handshake with a host and only proceed to send flow requests to the control plane once the handshake is completed. Benton et al. observe that although Transport Layer Security (TLS) would effectively protect against the threats of DoS and MitM attacks, the administrative overhead of implementing it has so far proven prohibitive.[23] They observe that this overhead results from the fact that TLS requires the generation of a sitewide certificate, controller certificates, switch certificates, the signing of certificates with the site-wide private key, and the installation of the correct keys and certificates on all devices. In contrast, the only configuration requirement without TLS is the controller's address. Additionally, many switch and controller vendors have either not fully implemented or skipped the TLS specification completely, further increasing the difficulty of adopting it.

B. Control Plane Threats

[16] Sezer, "Are we ready for SDN? Implementation challenges for software-defined networks," 2013.

[17] Seungwon Shin and Guofei Gu, "Attacking software-defined networks: A first feasibility study," *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*, 2013.

[18] Qai Yan and F. Richard Yu, "Distributed denial of service attacks in software-defined networking with cloud computing," *IEEE Communications Magazine* 53, (2015): 52–59.

[19] Qiao Yan, F. Richard Yu, Qingxiang Gong, Jianqiang Li, "Software-defined networking (SDN) and distributed denial of service (DDoS) attacks in cloud computing environments: A survey, some research issues, and challenges," *IEEE Communications Surveys and Tutorials*, 2015.

[20] Rowan Kloti, Vasileios Kotronis, and Paul Smith, "OpenFlow: A security analysis," *2013 21st IEEE International Conference on Network Protocols (ICNP*), 2013.

[21] Sandra Scott-Hayward, Sriram Natarajan, and Sakir Sezer, "A survey of security in software defined networks," *IEEE Communication Surveys and Tutorials*, 2015.

[22] Seungwon Shin et al., "Avant-Guard," *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security - CCS '13*, 2013.

[23] Kevin Benton, L. Jean Camp, and Chris Small, "OpenFlow vulnerability assessment," *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*, 2013.

One of the more concerning threats to the control plane is the possibility of an attacker exploiting a vulnerability in the controller's software that allows for arbitrary code execution, which would give an attacker a large amount of control over the network. [24] Although there is a notable lack of research examining solutions to such threats, Shalimov et al. present an initial method to evaluate the security of controllers.[25] They send both malformed headers and malformed payloads in OpenFlow packets to several controllers and evaluate how those packets are handled through hcprobe, a custom implementation of a controller benchmarker called cbench. For malformed headers, they manipulate the length, version, and type fields. For malformed payloads, they limit their testing to two particular OpenFlow packet types: PacketIn and PortStatus. For PacketIn packets, they indicate that the packet encapsulates an ARP packet when it actually encapsulates an IP packet. For PortStatus packets, they do not terminate the name field in the port description with a null byte, as is required. Shalimov et al. were able to crash several controllers with these techniques and their results show that sending malformed packets to the controller can be an effective method for discovering vulnerabilities.

C. Application Plane Threats

An example of an application plane threat is the possibility of an attacker exploiting a vulnerability in an application's software that allows for arbitrary code execution. As the SDN ecosystem expands it is likely that a market for SDN applications will evolve, similar to the mobile application stores for iOS and Android devices. If this is the case, it is likely that the SDN application market will have many vulnerable applications just like the iOS and Android

[24] Kreutz, "Towards secure and dependable software-defined networks," 2013.

[25] Alexander Shalimov et al., "Advanced study of SDN/OpenFlow controllers," *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia on - CEE-SECR '13*, 2013. doi:10.1145/2556610.2556621.

application markets, which, according to a 2015 study, contained a combined average of 9.041 vulnerabilities per application.[26]

Another example is the possibility that a malicious application masquerading as a legitimate one is installed in the application plane. There is currently no standardized way for the controller to discern trusted applications from untrusted ones and it is therefore possible for untrusted applications to make their way into the application plane. A particular concern is the ability for malicious applications to write flow rules that violate existing network security policy. For example, a malicious application disguised as a security application could try to write flow rules that allow packets from previously blacklisted domains. Such behavior would be difficult to detect with traditional methods as it is conceivable for a security application to have such privileges in the network.

A large portion of previous research in SDN security focuses on solutions to application plane threats and the majority of this research suggests that the best solution is to improve and enforce network security policy. We summarize these solutions below, including VeriFlow, FLOWGUARD, FLOVER, NICE, FortNOX, FlowChecker, FlowVisor, and Flow Security Language (FSL). VeriFlow, FortNOX, FLOWGUARD, and FLOVER mainly check for violations in security policy through the verification of flow rules. VeriFlow does this by first intercepting flow rules between the control and data planes, before they can be installed.[27] It then finds the set of network elements whose operation can be altered by a flow rule. Next, it builds forwarding graphs for every element using the current network state. Finally, it traverses the graphs to determine how the flow rule affects the status of one or more network components and

[26] "The State of Mobile Application Security 2014-2015," Checkmarx and AppSec Labs, accessed April 1, 2017, https://www.checkmarx.com/wp-content/uploads/2015/11/The-State-of-Mobile-Application-Security-2014-20151.pdf.

[27] Ahmed Khurshid et al., "Veriflow," *ACM SIGCOMM Computer Communication Review* 42, no. 4 (2012): 467.

if the flow rule causes a violation in security policy.

FortNOX is an extension for the NOX and it addresses flow rules that may result in security policy violations by first assigning each application a privilege level. [28] Every flow rule that an application attempts to install must be accompanied by a digital signature that enables FortNOX to identify the application and its privilege level. Flow rules that have no digital signature are assigned the lowest privilege level. FortNOX then intercepts new flow rules and detects if they conflict with any existing flow rules through the use of a custom algorithm. New flow rules that conflict with existing flow rules are only installed if they supersede the privilege levels of the existing flow rules.

Hu et al. note that detecting flow rules that violate security policy is difficult because flow rules support wildcards and because flows can be dynamically modified as they traverse the network. [29] Both these facts make it unclear which flow rules actually violate security policy. FLOWGUARD addresses these two problems by checking for security policy violations at the ingress switch of each flow and then tracks the flow path through the network to identify both the original source and final destination of each flow. Hu et al. also note that resolving policy violations is difficult because a flow rule may only partially violate policy and because deleting a flow rule may impact other flow rules. Both these facts make it difficult to remove flow rules. FLOWGUARD addresses these two problems by incorporating a network-wide view of flow rules to systematically resolve policy violations.

FLOVER addresses flow rules that may result in security policy violations by first translating flow rules and security policy to into an assertion set, or a series of true or false

---

[28] Philip Porras et al., "A security enforcement kernel for OpenFlow networks," *Proceedings of the first workshop on Hot topics in software defined networks - HotSDN '12*, 2012.

[29] Hongxin Hu et al., "FLOWGUARD: Building robust firewalls for software-defined networks," *Proceedings of the third ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '14*, (2014): 97–102.

statements. [30] This assertion set can then be processed and verified by a Satisfiability Modulo Theories (SMT) solver, which is a program that can solve large systems of true and false equations. The results of the SMT solver help to verify if security policy is violated by the flow rules in questions.

NICE and FlowChecker mainly check for violations in security policy through model checking, which is the process of determining whether a given model meets a given specification. NICE uses model checking in combination symbolic execution to discover security policy violations.[31] Symbolic execution is the process of determining which inputs exercise different code paths within the application in question. Canini et al. model the network through a custom algorithm, which enumerates all the possible states of the network and all the possible transitions between these states, including the transitions that cause security policy violations. They then use model checking and symbolic execution to enumerate all the possible code paths within a particular application and determine if a particular code path causes a state transition that, in turn, causes a security policy violation. In contrast to NICE, FlowChecker models the network by encoding OpenFlow configuration information using Binary Decision Diagrams, which are data structures used to represent boolean functions. [32] It then uses model checking to identify security policy violations in the network.

Sherwood et al. propose FlowVisor, which acts as a proxy between switches and multiple controllers.[33] It segments the network by re-writing flow rules so that they only affect their own

---

[30] Sooel Son et al., "Model checking invariant security properties in OpenFlow," *2013 IEEE International Conference on Communications (ICC)*, 2013.

[31] Marco Canini et al., "A NICE way to test OpenFlow applications," *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[32] Ehab Al-Shaer and Saeed Al-Haj, "FlowChecker," *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration - SafeConfig '10*, 2010.

[33] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar, "Flowvisor: A network virtualization layer," *OpenFlow Switch Consortium*, 2009.

segment. This enables multiple controllers to share a single network, making it easier to enforce security policy. Finally, Hinrichs et al. propose Flow Security Language (FSL), which serves as a replacement to Access Control Lists (ACLs), firewalls, Network Address Translators (NATs) and Virtual Local Area Networks (VLANs) in the SDN environment.[34] FSL allows for basic network access controls, directionality in communication establishment (similar to NAT), network isolation (similar to VLANs), communication paths, and rate limits.

D. Extending the BoK

While the majority of previous research has focused on the data and application planes, we have outlined it in order to illustrate how our research can contribute to the field; research into securing the controller is important not only because it is fundamental to the security of the entire SDN infrastructure, but also because it can inform previous research that aims to secure other aspects of the SDN infrastructure. It can help us narrow in on potential vulnerabilities in switches and applications, prevent us from exposing the controller to additional vulnerabilities when we implement solutions to other security problems, and inform applications that check for the correctness of flow table rules and the network structure. Discovering vulnerabilities in the controller software may even lead to important insights about the vulnerability discovery process in other software; certain methods of static analysis or fuzzing may be uniquely suited to discovering vulnerabilities in large, complex software in general. With the help of this research and previous research, we can stop implementing security haphazardly within the SDN environment and start implementing it by design, leading to a more robust and secure system as a whole.

[34] Timothy Hinrichs et al., "Expressing and enforcing flow-based network security policies," *University of Chicago*, 2008.

II. Fuzzing

Fuzzing is the process of sending specific data to a program that accepts input in an effort to cause it to behave unexpectedly.[35] The idea is that if a program behaves unexpectedly, the portion of code responsible for the abnormal behavior will likely contain an exploitable vulnerability. An effective fuzzer will create input that is valid enough so that it is not rejected at a shallow level within a program but invalid enough to trigger unexpected behavior within the program.[36] Fuzzers can be categorized in three main ways.[37] A fuzzer can be 1) mutation-based or generation-based depending on whether inputs are generated from scratch or by modifying existing inputs, 2) dumb or smart depending on whether it is aware of input structure, and 3) blackbox or whitebox depending on whether it is aware of program structure. These differences are summarized in Table 3.

Term	Definition
Mutation-based	Generation of input based on sample input
Generation-based	Generation of input based on a specific model, without requiring any sample input
Dumb fuzzing	Sending input to target program without regard to structure
Smart fuzzing	Sending input to target program within the constraints of a particular structure, such as base-64 encoding and checksums
Blackbox fuzzing	Sending input to target program without any measurement of code coverage

[35] Oehlert, "Violating Assumptions with Fuzzing," 2005.

[36] Christian Holler, Kim Herzig, and Andreas Zeller, "Fuzzing with code fragments," *Proceedings of the 21st USENIX Security Symposium*, 2012.

[37] John Neystadt, "Automated Penetration Testing with White-Box Fuzzing," Microsoft Developer Network, accessed April 1, 2017, https://msdn.microsoft.com/en-us/library/cc162782.aspx.

Whitebox fuzzing	Sending of input to target program, measuring the code coverage of input, and using that information to intelligently form new input

Table 3: Categories of Fuzzers.

A. Mutation-based Fuzzing

Mutation-based fuzzing begins with a valid sample input, which is then modified according to various strategies, like flipping, substituting, moving, or deleting random chunks of data. The input produced in mutation-based fuzzing is limited by the original valid input. For example, if an input can have up to three fields but the initial sample input only has two fields, a mutation-based fuzzer algorithm will never produce input that has all three fields.

B. Generation-based Fuzzing

Generation-based fuzzing begins with a model, which is a general outline of the input that the target program should accept. A generation-based fuzzer uses such an outline to produce fuzzed input for the target program. In general, the effectiveness of generation-based fuzzing relies on the accuracy of the initial model. If the model implies certain inputs should be accepted by the application when they actually are not, then the fuzzing process will be inefficient. However, if the model is well defined, the fuzzing process can be effective in producing novel inputs that are effective in causing exceptions.

C. Dumb Fuzzing

A dumb fuzzer is unaware of the structure of program input. It must therefore resort to either sending completely randomly generated input to the program or starting with a sample input and then apply fuzzing strategies to that sample input in order to produce new input. Since

the former is usually inefficient and unlikely to yield results, dumb fuzzers are usually mutation-based.

D. Smart Fuzzing

A smart fuzzer is aware of the structure of program input. They usually use a formal user-supplied model to identify the structure of input and then use various fuzzing strategies to produce new input. Since input is usually generated from a model, smart fuzzers are usually generation-based.

E. Blackbox Fuzzing

A blackbox fuzzer is unaware of program structure. It simply sends input to the target program and receives no feedback as to the amount of code coverage a particular input is able to produce. Blackbox fuzzing has the advantage of being relatively simple to implement. Although there has been plenty of research on whitebox fuzzing techniques, it is not readily apparent how to apply successful techniques to arbitrary target programs. Blackbox fuzzing can therefore be useful for revealing easy-to-find vulnerabilities as well as establishing a foundation for the application of more advanced whitebox fuzzing techniques.

F. Whitebox Fuzzing

There are two main problems with blackbox fuzzing that whitebox fuzzing helps to solve. First, the probability that a blackbox fuzzer will reveal a software vulnerability is small because there is no guarantee that the input generated by the dumb fuzzer will reach all parts of the target program. Figure 3 shows a sample program.

```
x = raw_input('Enter a number between 1 and 100: ')
if x == 50:
       sys.exit(1)
else:
       print x
```

Figure 3: Shortcomings of Blackbox Fuzzing.

Sending random input to the program only has a 1 percent chance of reaching the part of the code that causes an error. [38,39] While this example is simplified, it is useful in demonstrating why blackbox fuzzers inevitably fall short. The more complex a software system, the more code branches it will likely have and the less likely that random input will hit new branches in the program. This means that a dumb fuzzer is unlikely to cover a sufficient amount of the target program's code base to reveal a vulnerability. There is therefore a need for whitebox fuzzers, fuzzers that can intelligently choose input to send to the target program so as to increase the code covered by that input and therefore increase the likelihood of finding a vulnerability.

Second, blackbox fuzzers have no way of reducing the input space, which can cause problems in trying to scale the fuzzer. The Eddington Number, which is 10^{80} or the number of protons in the known universe, serves as a useful upper bound on any fuzzing process; even if we assume we have access to an infinitely powerful computer, it is not possible to store more inputs to the target application than 10^{80}. [40] There are therefore two options: 1) store fewer than 10^{80} inputs or 2) discard unsuccessful or uninteresting inputs. For example, a target application that accepts a High-definition (HD) picture as input. A HD picture consists of 1920 x 1080 pixels. Each pixel is represented by three bytes which can take on a total of 256^3 different values.

[38] Patrice Godefroid, "Random testing for security," *Proceedings of the 2nd international workshop on Random testing co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007) - RT '07*, 2007.

[39] Owen W. Redwood, "Lecture 8: Fuzzing Lecture 1," Florida State University, accessed April 1, 2017, http://www.cs.fsu.edu/~redwood/OffensiveComputerSecurity/lectures.html.

[40] Ibid.

Therefore, the total possible HD picture that can be sent to the application are $256^{3 \times 1920 \times 1080}$ which greatly exceeds 10^{80}. Therefore, in this case and many cases in general, it is unfeasible to completely exhaust the input space, as blackbox fuzzers are designed to do.

Whitebox fuzzers aim to reduce the input space through two main methods: dynamic taint analysis and forward symbolic execution. Dynamic taint analysis executes a program to observe which portions of code are affected by tainted sources like user input.[41] The idea is that the potions of code affected by user input are more likely to contain software vulnerabilities. computations There are two kinds of dynamic taint analysis: data flow dependencies and control flow dependencies. Data flow dependencies are when variables are tainted simply because they are derived from input. Figure 4 shows an example of a data flow dependency, where the variable z is tainted because it is derived from x.

```
x = raw_input('Enter an integer: ')
y = 1
z = x + y
```

Figure 4: Data Flow Dependency.

Control flow dependencies are when variables are tainted because they are part of control flow that is influenced by input. Figure 5 shows an example of a control flow dependency, where the variable z is tainted because its value depends on a conditional influenced by x.

```
x = raw_input('Enter an integer: ')
if x >= 0:
	z = 1
else:
	z = 2
```

Figure 5: Control Flow Dependency.

[41] Schwartz, Edward J., Thanassis Avgerinos, and David Brumley. "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)." *2010 IEEE Symposium on Security and Privacy*, 2010.

Forward symbolic execution builds a logical formula that describes the execution path of the target program in order to determine which inputs results in different execution.[42] The forward symbolic execution process is as follows. First, a program's logic is represented in symbols, which allows for a special focus on branches in the program. Next, all of the constraints that are required to produce a specific code path are recorded. This is repeated until all the different ways to reach all the different paths are enumerated. A constraint solver is then used in order to determine the constraints that need to be placed on program input in order to reach different code paths in the program. After enumerating all the different constraints on input, the fuzzing process can then be started with a much smaller input space, allowing for more effective fuzzing and increasing the likelihood of discovering vulnerabilities in the target program.

Research in the whitebox fuzzing domain often combines the use of dynamic taint analysis and forward symbolic execution in order to build logical formulas for those parts of a target program that depend upon tainted values.[43] In doing so, it is possible to not only determine which portions of code are influenced by tainted sources and are therefore likely to contain vulnerabilities, but also what input can be constructed to reach those potions of code. We summarize whitebox fuzzers presented by previous research below, including TaintCheck, SAGE, BuzzFuzz, TaintScope, and MAYHEM.

TaintCheck uses dynamic taint analysis with the help of Valgrind, an emulator that can help trace a program as it is runs.[44] Whenever the target program reaches a new block of code,

---

[42] Schwartz, "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," 2010.

[43] Ibid.

[44] James Newsome and Dawn Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS'05)*, 2005.

Valgrind translates and passes the block to TaintCheck, which traces the block to incorporate it in its dynamic taint analysis. TaintCheck then passes the block back to Valgrind, which translates the block back so that it may be executed. In order to make it flexible and extensible for future use, TaintCheck is broken down into three distinct parts: TaintSeed, TaintTracker, and TaintAssert. TaintSeed determines what inputs should be tainted, TaintTracker determines how the taint attribute should propagate, and TaintAssert determines the usage of tainted data should be interpreted as an exploit. TaintCheck was able to detect exploits on vulnerable versions of the ATPhttpd, cfingerd, and wu-ftpd programs.

SAGE (Scalable, Automated, Guided Execution) uses forward symbolic execution to improve the code coverage of the inputs it generates and a custom algorithm designed to find bugs in large search spaces.[45,46] SAGE has found 30 new Windows application bugs. Godefroid et al. extend SAGE by introducing a grammar, which is a set of rules that limit how input is crafted observe that the effectiveness. [47] They do this because programs that only accept highly-structured input usually reject a large number of inputs at shallow levels of code because they do not meet the basic requirements for input the program accepts. In an effort to increase the effectiveness of whitebox fuzzing with such programs, they present a method to generate grammars directly from forward symbolic execution to ensure that inputs reach beyond shallow parts of. Evaluating their algorithm against a regular whitebox fuzzing with the JavaScript interpreter of Internet Explorer 7, they are able to increase code coverage from 53% to 81% while using three times fewer tests.

BuzzFuzz uses dynamic taint analysis to identifies parts of a well-formed input that

[45] Patrice Godefroid, "Random testing for security," 2007.

[46] Patrice Godefroid, Michael Y. Levin, and David Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *Queue* 10, no. 1 (2012): 20.

[47] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin, "Grammar-based whitebox fuzzing," *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08*, 2008.

influence variables at key points in the target program.[48] BuzzFuzz automatically generates new inputs by modifying these identified parts of well-formed input. Because these new inputs typically preserve the underlying syntactic structure of the well-formed input, they tend to make it past the initial input parsing components of the program and reach code deep within the program. BuzzFuzz has found errors in two open-source applications: Swfdec, an Adobe Flash player, and MuPDF, a PDF viewer.

TaintScope also seeks to improve the effectiveness of whitebox fuzzing with programs that only accept highly-structured inputs. [49] It does this by using dynamic taint analysis to identify identify checksum-based checks on input and forward symbolic execution to bypass those checks. Additionally, it uses dynamic taint analysis to identify which parts of input are used in security-sensitive operations, such as invoking system or library calls, and then focuses on modifying those parts.

Cha et al. distinguish between an offline symbolic executor, which symbolically executes a single code path in a single run, with an online symbolic executor, which tries to execute all possible code paths in a single run.[50] They observe that offline executors are able to make progress for arbitrarily long times because each run involves executing a new code path. Offline executors are also able to use the results of previous runs for future runs as each run is executed independently. However, they repeat a lot of work because a lot of the same code needs to be executed for each run. In contrast, online symbolic executors are able to avoid re-executing the same code because they fork at branch points. However, they are unable to make progress for

[48] Vijay Ganesh, Tim Leek, and Martin Rinard, "Taint-based directed whitebox fuzzing," *2009 IEEE 31st International Conference on Software Engineering*, 2009.

[49] Tielei Wang et al., "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection," *2010 IEEE Symposium on Security and Privacy*, 2010.

[50] Sang Kil Cha et al., "Unleashing Mayhem on Binary Code," *2012 IEEE Symposium on Security and Privacy*, 2012. doi:10.1109/sp.2012.31.

arbitrarily long times because forking at branch points usually results in an exponential increase in code paths. They are also unable to use of the results of previous runs for future runs because they are not run multiple times. Cha et al. present Mayhem, which combines offline and online symbolic execution by alternating between online and offline symbolic execution runs. Mayhem has found 29 exploitable vulnerabilities in both Linux and Windows programs.

G. Practical Fuzzing Tools

While research into fuzzing has yielded some effective tools, it is not certain that these tools are applicable to a range of different software systems. Therefore, it is important to consider tools that have not necessarily been developed in an academic setting but have been proven practically effective nonetheless. Two particularly effective fuzzers are Peach and American Fuzzy Lop (AFL).

i. Peach

Peach is a smart, generation-based, blackbox fuzzer, which generates fuzzed input through XML configuration files called Peach Pits.[51] A Peach Pit consists of a Data Model, which tells Peach what format of input to expect, a State Model, which tells Peach how that input format changes as the program goes through different phases, and a Test, which brings the Data and State Models together. By decoupling the Data Model, State Model, and fuzzing engine, Peach allow the creation of custom fuzzers without having to write each fuzzer from scratch.

The Data Model tells Peach what format of input to expect. For example, to fuzz a webserver Peach would have to send one or more Hypertext Transfer Protocol (HTTP) requests

[51] "Peach Introduction," Deja vu Security, accessed April 1, 2017, http://community.peachfuzzer.com/Introduction.html.

and receive one or more HTTP responses, so both HTTP requests and responses would have to be defined in separate Data Models. The State Model tells Peach how that input format changes as the program goes through different phases. It consists of one or more States, which describe phases in the fuzzing process, and each State consists one or more Actions, which describes the steps to be taken in each phase. For example, to fuzz a webserver Peach would have to send a TCP SYN packet, receive a TCP SYN-ACK packet, send a TCP ACK packet, send one or more HTTP requests, and then receive one or more HTTP responses. Each one of these steps represent a State and its associated Actions in the State. The Test is the portion of the Peach Pit that describes how different Data and State Models interact during the length of a complete fuzzing run. It also defines a Publisher, which specifies how to send input to and received output from the target program during a fuzzing run. For example, to fuzz a webserver Peach would have to send input to the target program on TCP port 80 and receive output from the target program on a TCP user port.

ii. American Fuzzy Lop (AFL)

AFL is a dumb, whitebox, mutation-based fuzzer that has proven to be particularly effective at finding vulnerabilities. AFL was not constructed with the intention of being a proof of concept for any academic theory. Instead, the only governing principles are "speed, reliability, and ease of use."[52] The basic algorithm can be described in a few steps. The target program first has to be compiled with a special utility that allows AFL to conduct taint analysis. AFL then takes a valid sample input, executes the target program, and traces the execution path. It then attempts to reduce the size of the sample input until it no longer produces the same trace. Next,

[52] Michał Zalewski, "Technical "whitepaper" for afl-fuzz," accessed April 1, 2017, http://lcamtuf.coredump.cx/afl/technical_details.txt.

AFL applies various mutation-based fuzzing strategies to modify this reduced input, executing and tracing the execution path each time. For each modified input that results in a new execution trace, AFL repeats the process of reducing the size of that input until the reduced input no longer produces the same execution trace. If any of the inputs causes the target program to crash, that input is saved for later inspection.

AFL is particularly effective because it avoids global-scale comparisons of complete execution traces, which are particularly costly and can slow down the fuzzing process substantially. To achieve this, AFL first compiles each line of code so that it records the tuple: [ID of current code location], [ID of previously-executed code location]. Then AFL records both previously-unseen tuples in order to detect subtle changes in the control flow and the hit rate for every tuple in order to determine the code coverage of particular inputs. This ensures that both explicit conditional branches, and indirect variations in the behavior of the target program are recorded. Therefore, even if the mutations that AFL do not produce any unusual behavior, it is still a particularly useful tool in producing a set of inputs that collectively produce a substantial amount of code coverage. This set could then be utilized with other fuzzers in an effort to produce more interesting behavior from the target program.

III. OpenDaylight

The OpenDaylight controller is a framework to connect Southbound plugins, which generally communicate with network devices, to Northbound plugins, which generally communicate with applications that configure those same network devices.[53] The controller does this through a Service Abstraction Layer (SAL). Southbound plugins register themselves with

[53] Jan Medved et al., "OpenDaylight: towards a model-driven SDN controller architecture," *IEEE 15th international symposium on a world of wireless, mobile and multimedia networks (WoWMoM)*, 2014.

the SAL and the SAL then alerts Northbound plugins to the various Southbound plugins that are available. This approach resembles the way the Hardware Abstraction Layer (HAL) operates in Linux-based Operating Systems. In these systems, the HAL allows applications to use hardware connected to the system independent of the underlying hardware. With the SAL, the OpenDaylight controller allows Northbound plugins to interact with Southbound plugins, regardless of the specifics of the underlying network devices.

The Boron release of OpenDaylight uses a special SAL called a Model-driven Service Abstraction Layer (MD-SAL). MD-SAL uses YANG, which is a data modeling language used to model network device configuration so that the configuration of any network device can be abstracted out to provide extensibility and flexibility to Northbound protocols.[54] Once a YANG model is defined, the controller uses it to automatically create an interface via the RESTCONF plugin. This interface allows the consumers of the YANG model, or Southbound plugins, to be configured by producers of the YANG model, or Northbound plugins.

Take the example of an application adding a flow to an OpenFlow-enabled switch. When the controller starts up, both the Flow Programmer Service and the OpenFlow Plugin register themselves with the MD-SAL. When an application adds a flow through the controller's RESTCONF plugin, the add flow request is deserialized and a new flow is created in the Flow Service data tree. The Flow Programmer Service then receives a notification that a flow has been added as it is registered to receive updates for changes in the Flow Service data tree. Next, the Flow Programmer Service then uses the OpenFlow (OF) Plugin to generate a Remote Procedure Call (RPC), or a procedure that is executed remotely but coded as if it were executed locally, to add the flow in the appropriate switch. This RPC is then routed through the OF Plugin to the

[54] Martin Bjorklund, "RFC 6020 - YANG - A data modeling language for NETCONF," IETF Tools, accessed April 1, 2017, https://tools.ietf.org/html/rfc6020.

correct switch. Figure 6 illustrates this process.[55]

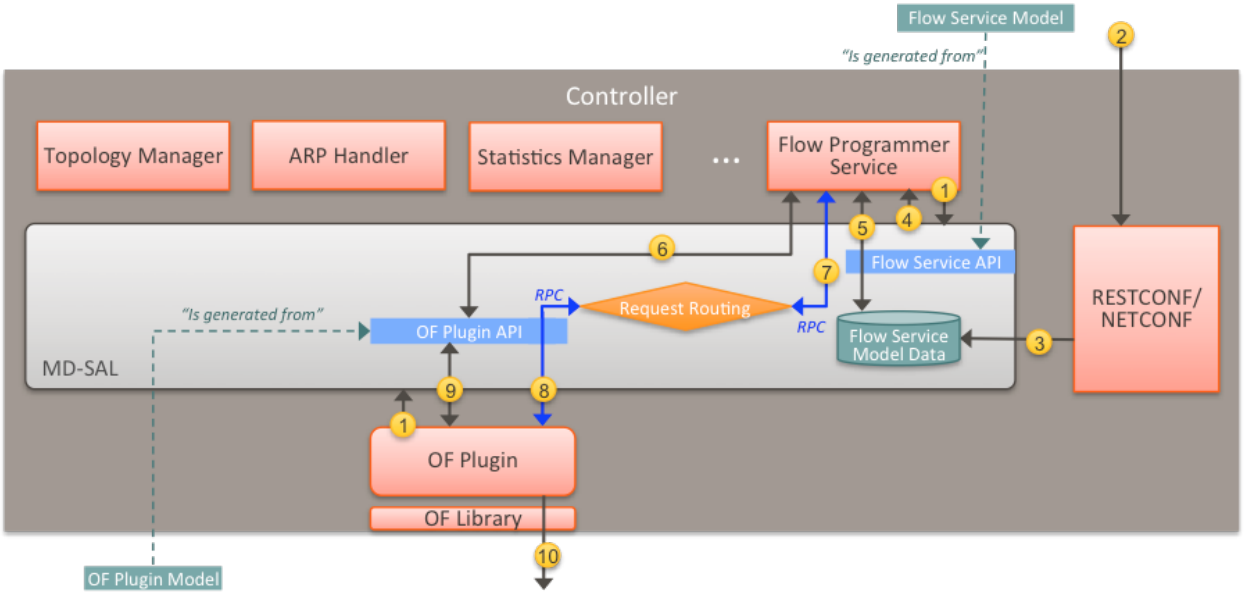

Figure 6: Adding a Flow in the OpenDaylight Controller

CHAPTER II

[55] "OpenDaylight Controller:MD-SAL:FAQ," OpenDaylight Project, accessed April 1, 2017, https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:FAQ.

METHODOLOGY

Our fuzzing framework utilizes a Virtual Machine (VM) running on VirtualBox 5.1.18 r114002 (Qt5.6.2). The specifications of the VM were chosen in accordance with benchmarks set by the OpenDaylight Project and are summarized in Table 4.[56]

Component	Specification
Operating System (OS)	Ubuntu 16.04 LTS
Memory	3.9GiB
CPU	Intel Core i7-3740QM CPU @ 2.70GHz
OS type	64-bit
Disk	14.6 GB

Table 4: VM Specifications.

First, we downloaded and extracted OpenDaylight within the VM. Figure 7 illustrates this process.

```
$ wget
https://nexus.opendaylight.org/content/repositories/opendaylight.release/org/opendayl
ight/integration/distribution-karaf/0.5.2-Boron-SR2/distribution-karaf-0.5.2-Boron-
SR2.tar.gz
$ tar -xzf distribution-karaf-0.5.2-Boron-SR2.tar.gz
```

Figure 7: Downloading and Extracting the OpenDaylight Controller.

Next, we increased the maximum size of the log file, karaf.log, so that it is not rotated before log output can be recorded later on during the fuzzing process. Figure 8 illustrates this process.

```
$ sed -i s/maxFileSize=1MB/maxFileSize=500MB/ distribution-karaf-0.5.2-Boron-
SR2/etc/org.ops4j.pax.logging.cfg
```

[56] "CrossProject:Integration Group:Performance Test:Results," OpenDaylight Project, accessed April 1, 2017, https://wiki.opendaylight.org/view/CrossProject:Integration_Group:Performance_Test:Results.

Figure 8 Changing the Maximum File Size of karaf.log.

Finally, we set the JAVA_HOME variable and ran the controller process. Figure 9 illustrates this process.

```
$ export JAVA_HOME=/usr/lib/jvm/default-java
$ distribution-karaf-0.5.2-Boron-SR2/bin/karaf
```

Figure 9: Setting the JAVA_HOME Variable and Starting the OpenDaylight Controller..

We chose to use Python modules to construct a smart, generation-based, blackbox fuzzer to fuzz the OpenDaylight controller. This fuzzing framework is available on GitHub.[57] We chose this methodology because, although we outlined several fuzzing frameworks in Chapter I, including ones that have been particularly successful in finding vulnerabilities in other programs, it is uncertain whether those frameworks are as applicable to a large and complex software system such as the OpenDaylight controller. By developing a basic fuzzing framework with Python modules, our hope is to provide a foundation for future efforts to develop more advanced frameworks. We chose to use this approach to fuzz two OpenDaylight features that form a large part of the core controller functionality: the controller's odl-l2switch-all and odl-restconf-all features. For each feature, we followed the guidelines of the Microsoft Security Development Lifecycle for network fuzzing and repeated the fuzzing process for one hundred thousand iterations.[58]

I. OpenFlow Plugin (odl-l2switch-all)

[57] Walid Sharif, "wsharif/thesis," accessed April 1, 2017, https://github.com/wsharif/thesis.

[58] "SDL Process - Phase 4: Verification," Microsoft Developer Network, accessed April 1, 2017, https://msdn.microsoft.com/en-us/library/windows/desktop/cc307418.aspx.

The odl-l2switch-all feature in the OpenDaylight controller implements OpenFlow Version 4 (v4). Figure 10 illustrates how we installed the feature.

```
feature:install odl-l2switch-switch
```

Figure 10: Installing odl-restconf-all.

The OpenFlow protocol allows controllers and switches to communicate for the purposes of forwarding packets. OpenFlow v4 has a total of thirty different packet types, which facilitate this basic purpose. Our methodology for fuzzing the OpenFlow feature of the controller includes four main steps: 1) construct each of the thirty OpenFlow packet types within the constraints of the OpenFlow v4 specification, 2) generate an OpenFlow packet randomly chosen from the thirty packet types, 3) establish a connection with the controller, 3) send the generated OpenFlow packet, and 4) record CPU usage, memory usage, and log output.

We first constructed each of the thirty OpenFlow packet types. According to the OpenFlow specification, each packet consists of several fields, some of which are constrained to certain values and some of which are not.[59] Our approach was to adhere to the general structure of the packet and only vary fields within their constraints. Our hope is that this fine-grained approach will have a greater likelihood of reaching code deeper within the controller's OpenFlow feature and that packets are not rejected as malformed by shallow, parsing portions of the feature. This will increase the likelihood that the generated packets will cause the controller to exhibit unexpected behavior. We used the os module to generate packet fields that have no constraints. Figure 11 shows how one byte of random data can be constructed with the os module.

[59] "OpenFlow Switch Specification," Open Networking Foundation, accessed April 1, 2017, https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.3.pdf.

```
from os import urandom
field = urandom(1)
```

Figure 11: Constructing One Byte of Random Data with the os Module.

Figure 12 shows how one byte of random data can also be constructed with the random and struct modules.

```
from random import randint
from struct import Struct
field = Struct('! B').pack(randint(0, 255))
```

Figure 12: Constructing One Byte of Random Data with the random and struct Modules.

We favored using the os module to construct unconstrained fields because this approach is about 33% faster, as illustrated by figure 13, and this helps to speed up the fuzzing process.

```
>>> timeit.Timer("os.urandom(1)", "import os, random, struct").timeit(1) /
timeit.Timer("struct.Struct('! H').pack(random.randint(0, 255))", "import os, random,
struct").timeit(1)
0.6666666666666666
```

Figure 13: Comparing the Time to Construct One Byte of Random Data with the os Module and the random and struct Modules.

For packet fields that are constrained to particular values, we used the random and struct modules in tandem. Figure 14 shows how a one byte field that can only take on values between zero and ten can be constructed.

```
from random import randint
from struct import Struct
field = Struct('! B').pack(randint(0, 10))
```

Figure 14: Constructing a One Byte Field that Can Only Take on the Value between Zero and Ten.

We packed data in network byte order, represented by the '!' argument passed to the Struct function, because that data eventually has to be sent to the controller over the network.

Each OpenFlow packet consists of a header, which is basically the same for each packet type, and a payload, which varies from packet type to packet type. Figure 15 illustrates this structure.[60]

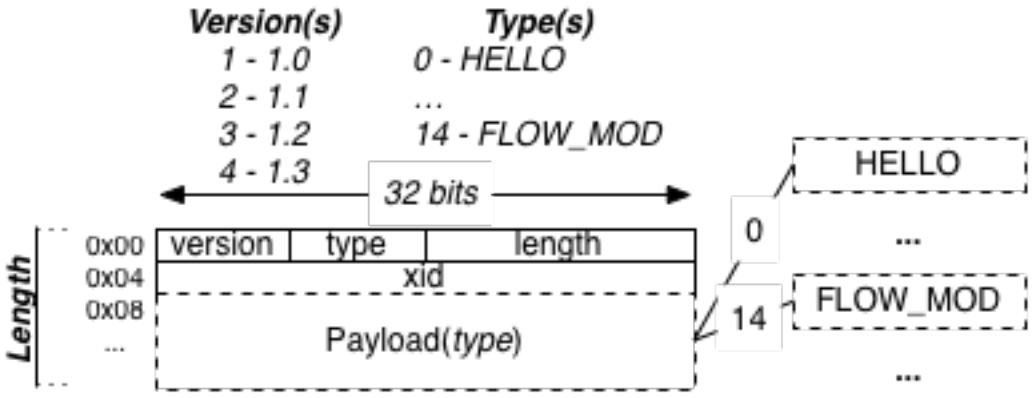

Figure 15: OpenFlow Packet Structure.

The header for each OpenFlow packet has four fields: version, type, length, and xid. Only the type field varies amongst the thirty different packet types. The Appendix describes each part of the header, its constraints, and how it can be generated by a Python expression. The version field is the value of the OpenFlow protocol version, which is 4, packed into a one-byte unsigned integer in big-endian. Figure 16 shows how the version field can be expressed in Python.

```
version = Struct('! B').pack(4)
```

Figure 16: Constructing the Version Field.

[60] "Message Layer," Flowgrammable, accessed April 1, 2017, http://flowgrammable.org/sdn/openflow/message-layer/.

The type field is the value of the OpenFlow packet type packed into a one-byte unsigned integer in big-endian. Figure 17 shows how the type field can be expressed in Python.

```
type = Struct('! B').pack(packetType)
```

Figure 17: Constructing the Type Field.

The length field is equal to the combined length of the OpenFlow packet header and payload packed into a two-byte unsigned integer in big-endian. Thus, the maximum length of any given OpenFlow packet is 65535 bytes as this is the maximum value of a two-byte unsigned integer. The length of the header for any OpenFlow packet will always be eight bytes. However, the length of the payload for an OpenFlow packet varies, depending on the packet type. Figure 18 shows how the length field can be expressed in Python.

```
length = Struct('! H').pack(8 + len(payload))
```

Figure 18: Constructing the Length Field.

The xid, or transaction ID, field is a 4-byte field that does not have any constraints. Figure 19 shows how the version field can be expressed in Python.

```
xid = urandom(4)
```

Figure 19: Constructing the Xid Field.

The payload of an OpenFlow packet varies quite drastically in both content and length depending on the packet type. The Appendix lists these payloads, their field constraints, and the Python expressions that we used to generate them. The ofptHello, ofptBarrierReq, ofptBarrierRes, ofptFeatureReq, ofptGetAsyncReq, and ofptGetConfigReq packets do not have

payloads. Many of payloads of the remaining twenty-four packet types can be constructed with simple use of the os, random, and struct modules. However, there are thirteen packet types that have more complicated payloads: ofptEchoReq, ofptEchoRes, ofptError, ofptExperimenter, ofptFlowMod, ofptFlowRemoved, ofptGroupMod, ofptMeterMod, ofptMultipartReq, ofptMultipartRes, ofptPacketIn, ofptPacketOut, and ofptQueueGetConfigRes. The payloads for these packets are complicated to construct because they have one or more fields that are variable in length.

For the ofptEchoReq, ofptEchoRes, ofptError, and ofptExperimenter payloads, the process is not too difficult because the fields that are variable in length are not constrained in terms of content. For these payloads we used the os module to generate a field with random data of random length. For example, the payload for the ofptEchoReq packet consists of a data field that is variable in length but is not constrained in terms of content. Figure 20 shows how we constructed the payload for this packet with the os module.

```
data = urandom(randint(0, maxLength - 8))
```

Figure 20: Constructing Random Data of Variable Length.

For the remaining nine packet types, the process is slightly more difficult because the fields that are variable in length are sometimes constrained in terms of content. This poses a problem because we had to generate the content for these fields and do so for a randomly assigned length. Some of these fields have subfields that are of variable length as well, further complicating the process. In order to effectively construct these payloads, we defined several helper functions: generateActions, generateBuckets, generateGroupDescriptions, generateGroupStats, generateInstructions, generateMatch, generateMeterBands, generateMeterConfigs, generateMeterStats, generateOxm, generateQueues, generatePorts, and

generateTableFeatures.

All the helper functions have the same basic structure. First, they are passed an integer which represents the maximum length of the string they return. They then declare a list, which holds the generated strings. Next, they enter a while loop and strings are generated according to their constraints under the OpenFlow v4 specifications. Each generated string is added to the declared list if its addition does not make the combined length of the list's entries larger than the maximum length the function was passed. If the generated string is added to the list and the current combined length of all the strings in the list plus the maximum possible length of a generated string is less than the maximum length, the while loop is executed for an additional iteration. If not, the while loop is exited. A random number of list elements are then combined and returned as a string.

Take, for example, the generateActions function. The function is passed a maximum length as an argument and declares a list of actions. The function then enters a while loop and a single action is generated according to the action constraints under the OpenFlow v4 specifications.[61] The generated action is added to the list if its addition does not make the combined length of the list's entries larger than the maximum length the function was passed. If the generated action is added to the list and the combined length of the list's entries plus sixteen is greater than the maximum length the function was passed, the while loop is exited. Sixteen represents the largest possible action that can be generated by the while loop and ensures that it is possible for any of the actions to be added to the list if randomly selected when the while loop is executed for an additional iteration. A random number of list elements are then combined together and returned.

After constructing all thirty packet types, we established a connection with the controller

[61] "Actions," Flowgrammable, accessed April 1, 2017, http://flowgrammable.org/sdn/openflow/actions/.

before sending it a generated packet. This is a required step because the controller will simply drop the sent packet unless a connection is established first. To establish a connection, the controller first sends a ofptHello packet to a switch it has not seen before, or vice versa, in order to negotiate the version of OpenFlow they will use to communicate. Next, either the switch or controller sends a ofptHello packet in response, depending on whether the controller or switch sent the original ofptHello packet. Then, the controller sends a ofptFeaturesReq packet in order to determine what features the switch it is communicating with supports. The switch replies with a ofptFeaturesRes packet to describes the features it supports. When this process is finished, the controller and switch can proceed with further communication. Figure 21 illustrates the initial negotiation process.[62]

[62] "State Machine," Flowgrammable, accessed April 1, 2017, http://flowgrammable.org/sdn/openflow/state-machine/.

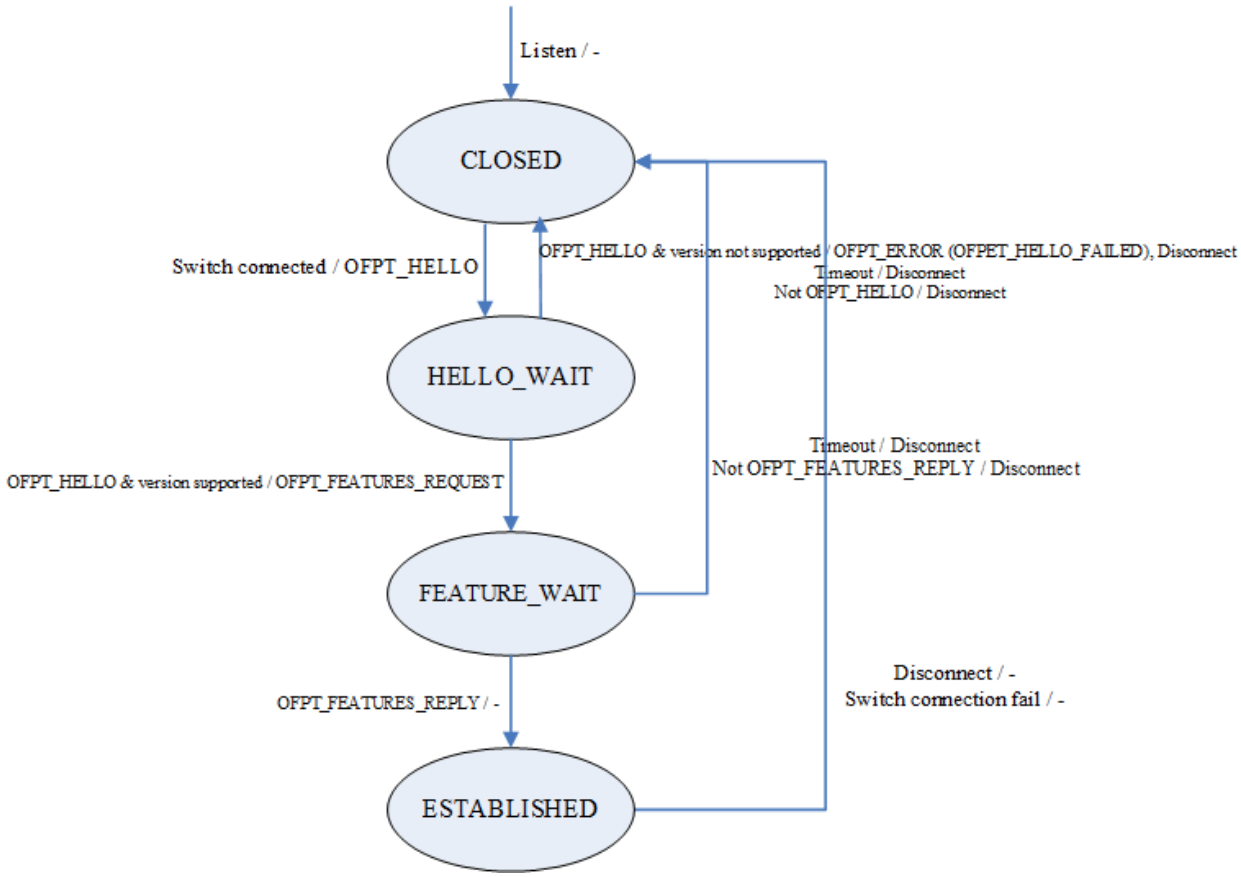

Figure 21: Controller-Switch Connection Process.

We established a connection with the controller by sending an ofptHello and ofptFeatureRes packet and used the time module to wait half a second each time for the controller to respond with acknowledgements. We then generated a packet at random and sent it to the controller, waiting another half a second for an acknowledgement. We then wrote the packet that was sent to a file for later reference , checked the CPU and memory usage of the controller process, and recorded any log output. We obtained the CPU and memory usage of the controller process by running and getting the output of the ps command as a subprocess. We obtained any log information written to the controller log file, karaf.log, by running and obtaining the output of the wc and tail commands as subprocesses.

II. RESTCONF Plugin (odl-restconf-all)

The odl-restconf-all feature in OpenDaylight implements the RESTCONF protocol. Figure 22 illustrates how we installed the feature.

```
feature:install odl-restconf-all
```

Figure 22: Installing odl-restconf-all.

After installation of the feature, a REST interface is created that listens on port 8181 and can be accessed through HTTP requests. Our methodology for fuzzing the OpenFlow feature of the controller follows four main steps: 1) enumerate all valid HTTP requests that can be sent to the RESTCONF feature, 2) generate a fuzzy request chosen randomly from the list of valid requests, 3) send the request, and 4) record CPU usage, memory usage, and log output.

We first enumerated all valid HTTP requests that could be sent to the RESTCONF feature. Unlike with the OpenFlow feature, there is no standardized list of HTTP requests that all RESTCONF implementations must support. RESTCONF implementations have different purposes and so the exact HTTP requests accepted by any given implementation varies. Additionally, there does not seem to be a comprehensive list of valid requests within the OpenDaylight documentation. However, there is an OpenDaylight feature, known as the apidocs explorer, that provides such a list. Its purpose is to document the Application Programming Interfaces (APIs) provided by the RESTCONF feature. Figure 23 illustrates how we installed the feature.

```
feature:install odl-mdsal-apidocs
```

Figure 23: Installing odl-mdsal-apidocs.

After we installed the apidocs explorer, we accessed a complete list of valid HTTP requests accepted by the RESTCONF feature at http://127.0.0.1:8181/apidoc/explorer/index.html with the default username and password of ‘admin’ and ‘admin’ respectively. This apidocs explorer is illustrated by Figure 24.

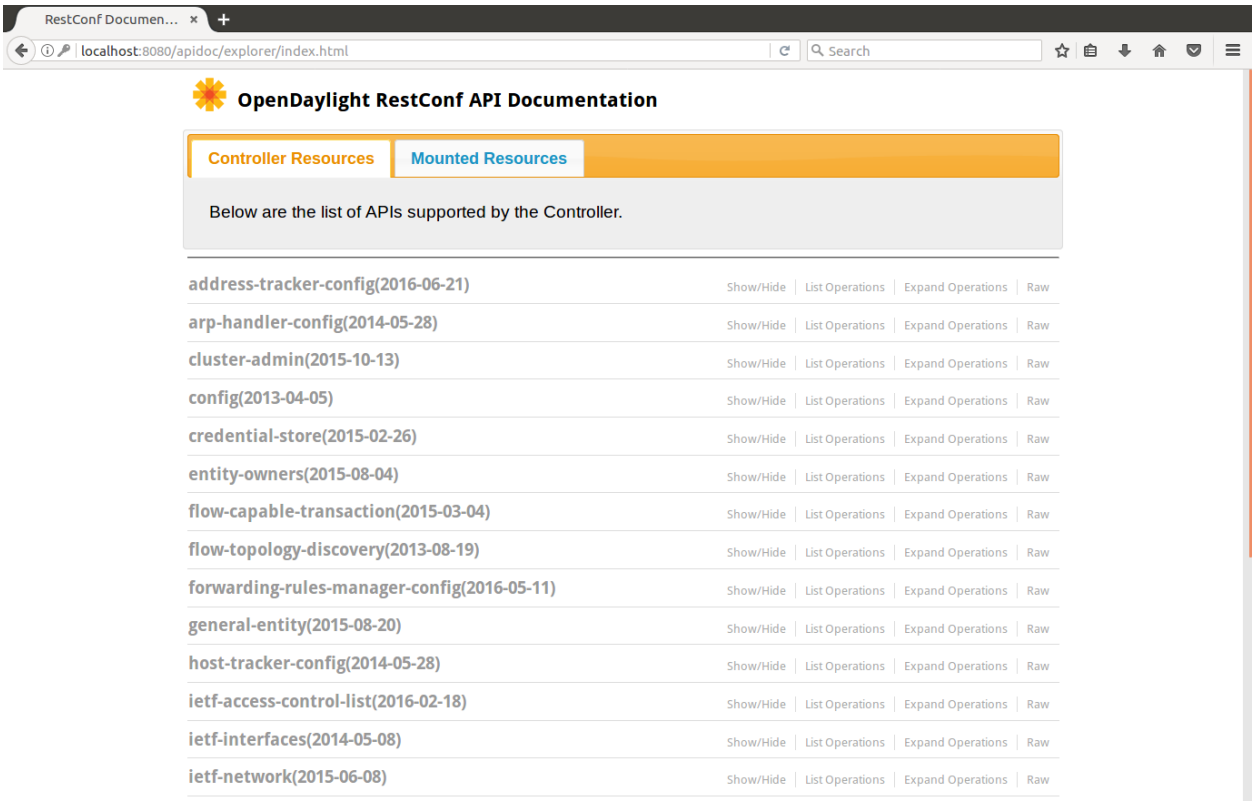

Figure 24: Apidocs Explorer.

We used the apidocs explorer to enumerate a list of valid requests through the use of the Python modules selenium, a web browsing automation module, and bs4, a HTML parsing module. The apidocs explorer is mainly rendered through JavaScript and so simply passing the loaded page to the bs4 module did not allow us to enumerate all the HTTP requests. Instead, we used the selenium module to automate the loading of all dynamically rendered JavaScript content

before passing the HTML document to the bs4 module.

First, we downloaded a webdriver that the selenium module could use. We chose to use the FireFox webdriver. Figure 25 shows how we downloaded and installed the FireFox webdriver.

```
$ wget https://github.com/mozilla/geckodriver/releases/download/v0.15.0/geckodriver-
v0.15.0-linux64.tar.gz
$ tar -xzf geckodriver-v0.15.0-linux64.tar.gz
$ sudo mv geckodriver /usr/bin
```

Figure 25: Downloading and Installing the FireFox webdriver.

Next, we started the web browser and passed it the URL of the apidocs explorer as an argument. We included the username and password in the URL to avoid having to instruct the selenium module to enter the credentials through additional actions. We then dismissed the popup dialog that resulted. This popup dialog is illustrated by figure 26.

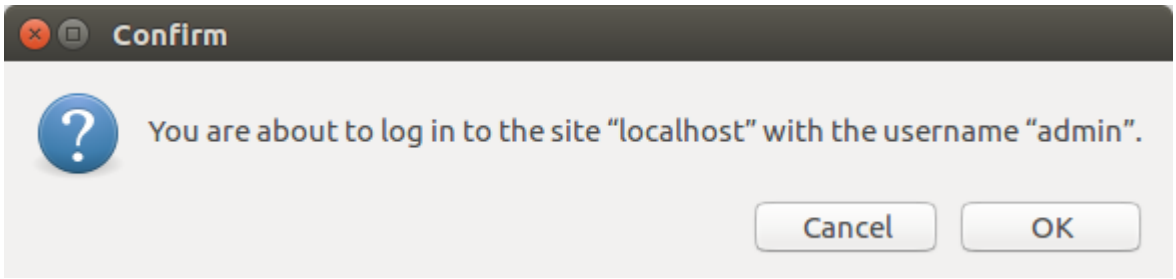

Figure 26: FireFox Popup Dialog.

Next, we waited until the page was fully. In particular, we waited until a specific HTML element, the ‘Show/Hide’ link, was loaded because clicking this element exposes the HTML content that corresponds to the HTTP requests we wished to enumerate. We used the FireFox Inspector feature to manually inspect the page and find the HTML tag associated with the

‘Show/Hide’ link. Figure 27 shows the FireFox Inspector with a ‘Show/Hide’ HTML element highlighted.

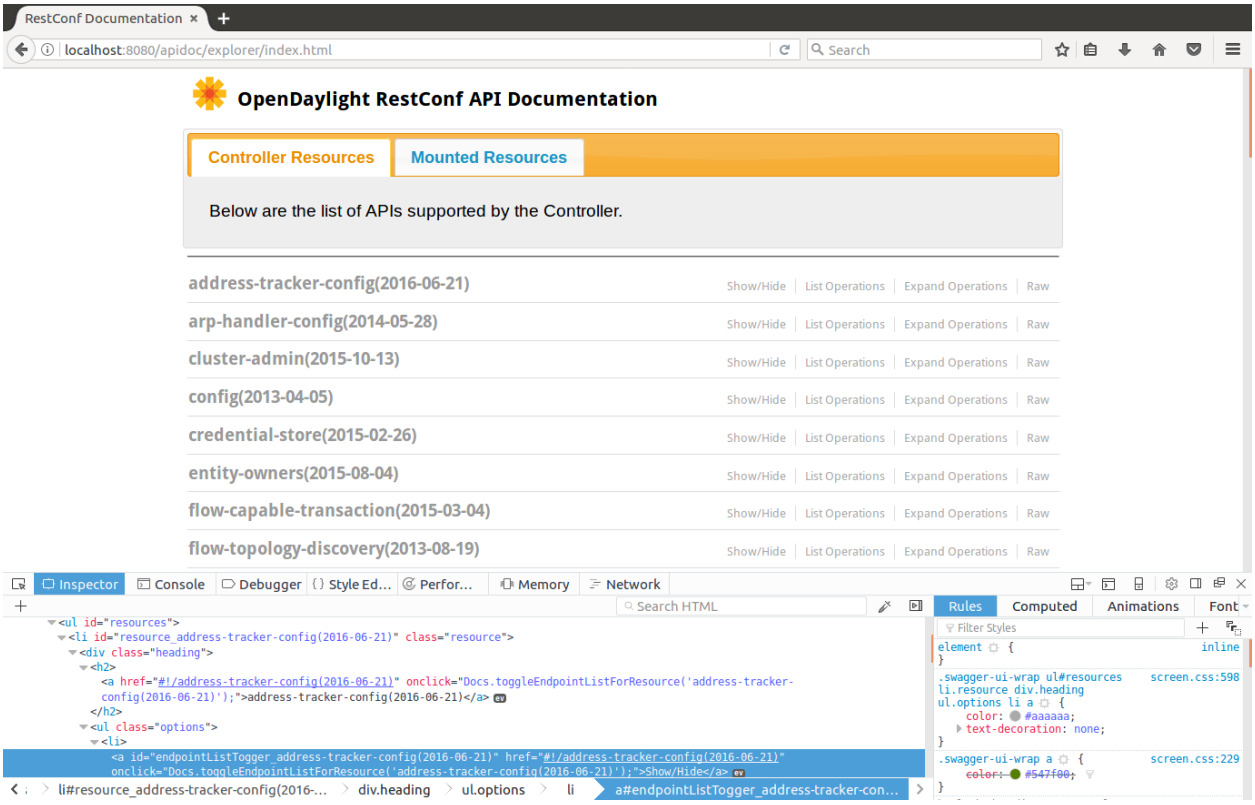

Figure 27: FireFox Inspector.

After identifying the tag that belongs to the ‘Show/Hide’ link, we used the selenium module to wait until those HTML elements were clickable. If they took more than thirty seconds to load, we assumed there was an error loading the page and restarted our program. We then used the selenium module to click on all of the 'Show/Hide' links and passed the rendered page to the bs4 module. Figure 28 shows the apidocs explorer with all of the 'Show/Hide' links clicked.

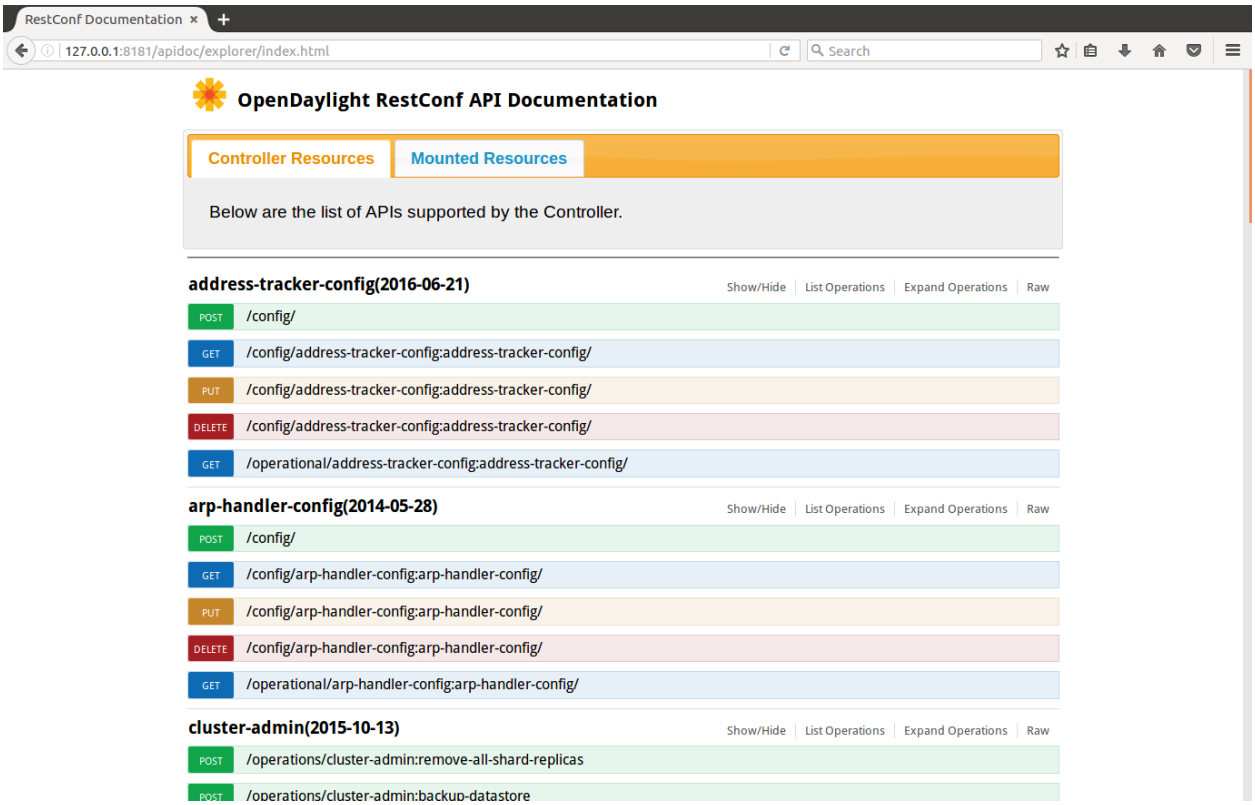

Figure 28: Apidocs Explorer with All HTTP Requests Listed.

Finally, we parsed the rendered page for all the listed HTTP requests and saved them to a file. In particular, we saved the HTTP method, URL, and any accompanying payload.

After enumerating all valid HTTP requests, we used those requests to generate fuzzy requests. Again, our approach was to adhere to the general structure of the requests and only vary fields within their constraints and we hope is that this approach will have a greater likelihood of reaching code deeper within the controller's RESTCONF feature. First, we modified the request that contained variable fields, which were demarcated by '{' and '}'. These variable fields are meant to represent specific instances within MD-SAL. We modified these fields while ensuring that 1) the maximum URL length was not exceeded and 2) the URL did not contain any invalid characters. We verified the maximum length of a URL that the RESTCONF feature accepts is

5909 bytes. We verified this length with the program shown in Figure 29.

```
from requests import post, get, put, delete
from requests.auth import HTTPBasicAuth

requests = [post, get, put, delete]
maxLength = []

for i in requests:
	url = '/restconf/'
	while True:
		response = i('http://127.0.0.1:8181' + url, auth =
HTTPBasicAuth('admin', 'admin'), headers = {'Content-Type': 'application/json'})
		if response.status_code == 413:
				maxLength.append(len(url))
				break
		else:
			url += 'a'

print min(maxLength)
```

Figure 29: Program to Verify the Maximum Length of a URL in a HTTP Request to the RESTCONF Feature.

The RESTCONF feature accepts all characters specified by RFC 3986 except the '%', '[', and ']' characters.[63] We verified this with the program shown in Figure 30.

```
from string import letters, digits, punctuation
from requests import get
from requests.auth import HTTPBasicAuth

urlCharacters = [i for i in letters + digits + punctuation] + ['%' +
hex(i)[2:].zfill(2) for i in range(256)]

for i in urlCharacters:
	response = get('http://127.0.0.1:8181/restconf/' + i, auth =
HTTPBasicAuth('admin', 'admin'))
	if response.status_code not in (204, 404):
		print i
```

Figure 30: Program to Verify the URL Characters Accepted in the URL in a HTTP Request to the RESTCONF Feature.

[63] Tim Berners-Lee, Roy Fielding, Larry Masinter, "RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax," IETF Tools, accessed April 1,2017, https://tools.ietf.org/html/rfc3986.

After we verified these two constraints, we modified the URLs of requests that contained variable fields with the modifyUrl function, which reserves a minimum length for each field that must be generated, generates a random combination of valid URL characters of random length, and then inserts those characters into the URL.

Next, we modified the request payloads that had variable fields, which were demarcated by the placeholder strings "string", "integer", "number", "boolean", and "object". We replaced these placeholder strings with randomly generated data within specific constraints.[64,65] These constraints are specified by JavaScript Object Notation (JSON), which states that a string can be any unicode string, an integer can be any integer that is less than nineteen digits, a number can either be integer less than nineteen digits or a float, a boolean can be any boolean, and an object can be any dictionary where all the keys are strings and all values are either be strings, integers, numbers, or booleans.[66] We generated random data for each type with the generateString, generateInteger, generateNumber, generateBoolean, and generateObject functions respectively. These functions return strings because the generated request iseventually converted to a string by the requests module before it is sent over the network to the RESTCONF feature.

The functions basically operate in the same way; they choose a random length for the string that is returned and then generate a string of that length one iteration at a time. For example, the generateString function chooses a random length between six, which is the

[64] The RESTCONF feature accepts payloads in both JSON and Extensible Markup Language (XML) format. While the apidocs explorer provide outlines for the structure of JSON payloads, it provides no such outlines for the structure of the same payloads in XML. Given this difficulty and the fact that the documentation suggests that JSON and XML payloads get processed in identical fashion, we have chosen to only generate payloads in JSON.

[65] "Overview for programmers," OpenDaylight Project, accessed April 1, 2017, https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Restconf:Overview_for_programmers.

[66] We exclude generating objects that include objects because of the difficulties associated with doing so randomly.

minimum length of a valid unicode string, and the maximum length it is passed. It then generates a unicode string of that chosen length one iteration at a time.

We then replaced the placeholders in the payload with the generated data and saved the modified requests to a text file for later inspection so that if a certain request is observed to cause unexpected behavior, there is a way to reference that particular request. Next, we randomly selected a request and sent it to the RESTCONF feature by reading from the modifiedRequests.txt file and sending the request through either the post, get, put, or delete functions of the requests module as necessary. Finally, we recorded CPU usage, memory usage, and log output after each request was sent. We obtained the CPU and memory usage of the controller process by running and getting the output of the ps command as a subprocess. We obtained any log information written to the controller log file, karaf.log, by running and obtaining the output of the wc and tail commands as subprocesses. We then recorded the response status code and requests error, if any, through the requests module.

CHAPTER III

RESULTS

I. OpenFlow Plugin (odl-l2switch-all)

Figures 31 and 32 illustrate CPU and memory usage throughout the fuzzing run for the OpenFlow plugin of the OpenDaylight controller.

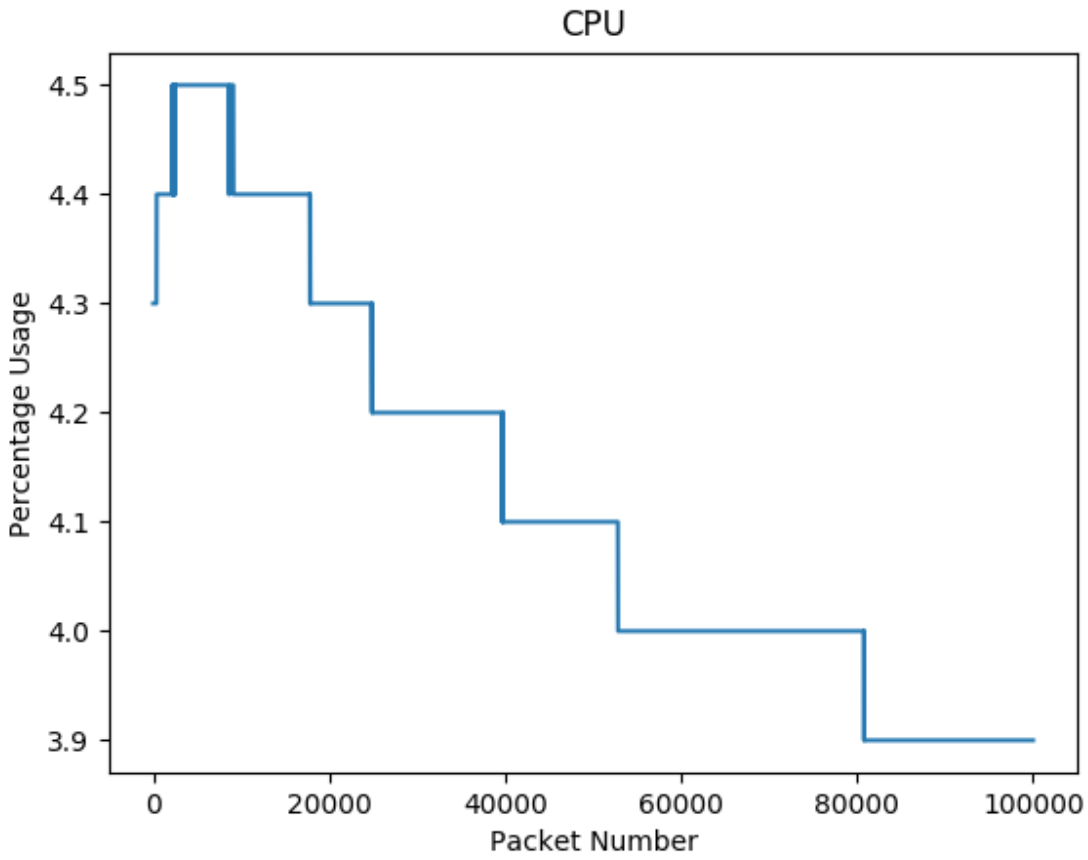

Figure 31: OpenFlow CPU Usage Results.

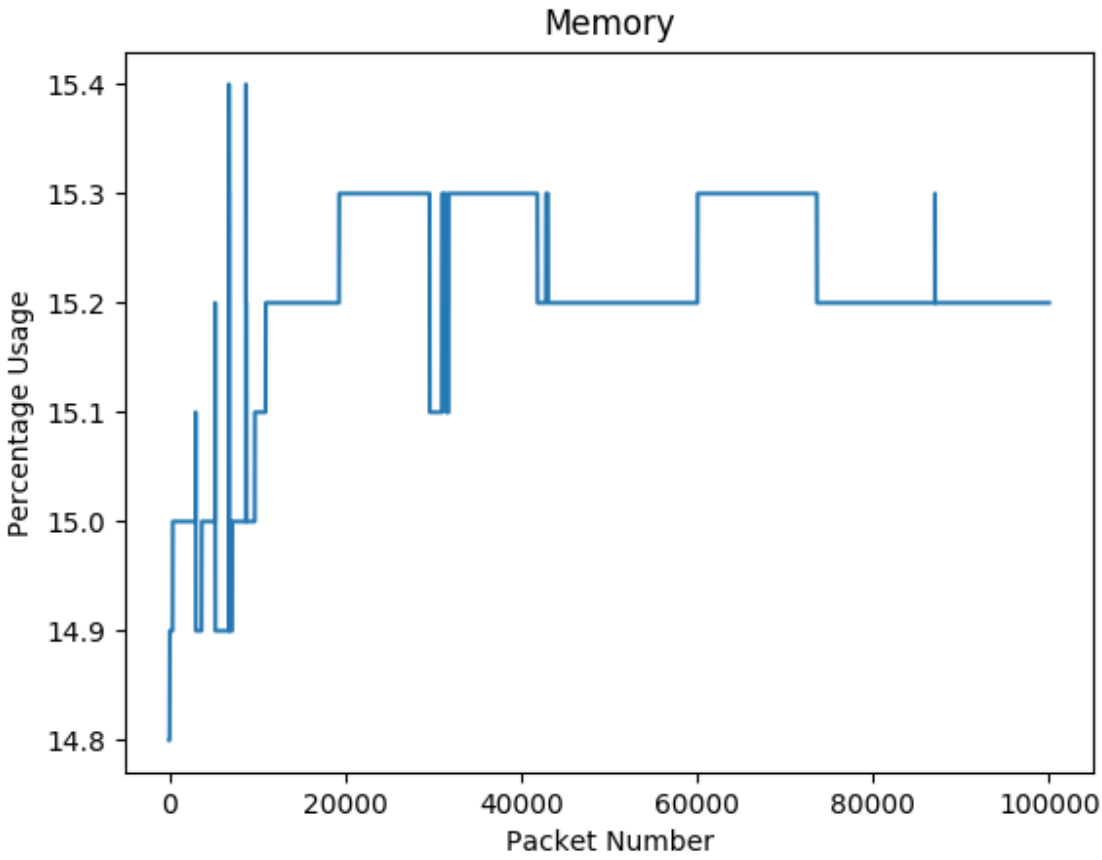

Figure 32: OpenFlow Memory Usage Results.

This single run of the fuzzing process took about seven hours to complete as illustrated by Figure 33.

```
$ time openFlowFuzzer.py distribution-karaf-0.5.2-Boron-SR2/bin/karaf

real   414m57.283s
user   90m59.552s
sys    45m53.476s
```

Figure 33: Time to Complete OpenFlow Fuzzing Run.

The variance in CPU usage is not particularly interesting; there seems to be a general downward trend in usage with no significant spikes. There does, however, seem to be several spikes in

memory usage throughout the fuzzing run. The four spikes that occur before the twenty-thousandth packet is sent are particularly interesting because they run opposed to the more gradual increases in memory usage of 0.1 percentage point increments. Inspecting the results.csv file reveals that these readings were taken after packets 2995, 5216, 6781, and 8717 were sent. We can reference these packets in the packets.txt file with the code shown in Figure 34, where x is the packet number referenced.

```
with open('packets.txt', 'r') as f:
       data = f.read().split('#packet')[1:]
print data[x]
```

Figure 34: Code to Reference Requests with OpenFlow Plugin.

Referencing these packets in the packets.txt file allows us to determine that packets 2995, 5216, 6781, 8716 are ofptSetAsync, ofptGetConfigReq, ofptGroupMod, and ofptMultipartRes packets respectively. Table 5 summarizes these packets.

Packet number	Packet type	Bytes
2995	ofptSetAsync	32
5216	ofptGetConfigReq	8
6781	ofptGroupMod	26552
8716	ofptMultipartRes	58008

Table 5: Packets that Caused Spikes in CPU Usage.

Further inspection of the packets does not provide any compelling evidence as to why they would cause such sudden spikes in memory usage. The log output for the packets does not seem to give any clues either. However, we believe repeating the fuzzing process multiple times exclusively using the packets in question may reveal patterns that can help to answer this

question.

II. RESTCONF Plugin (odl-restconf-all)

Figures 35 and 36 illustrate CPU and memory usage throughout the fuzzing run for the RESTCONF plugin of the OpenDaylight controller.

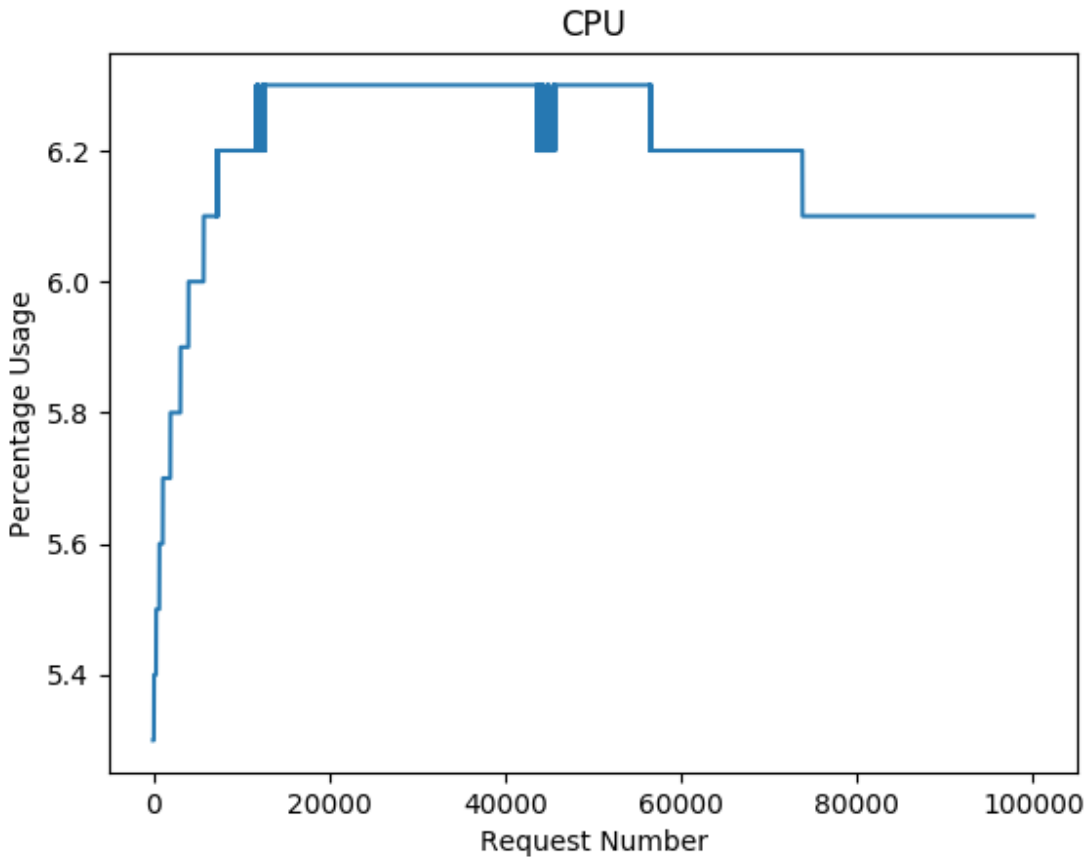

Figure 35: RESTCONF CPU Usage Results.

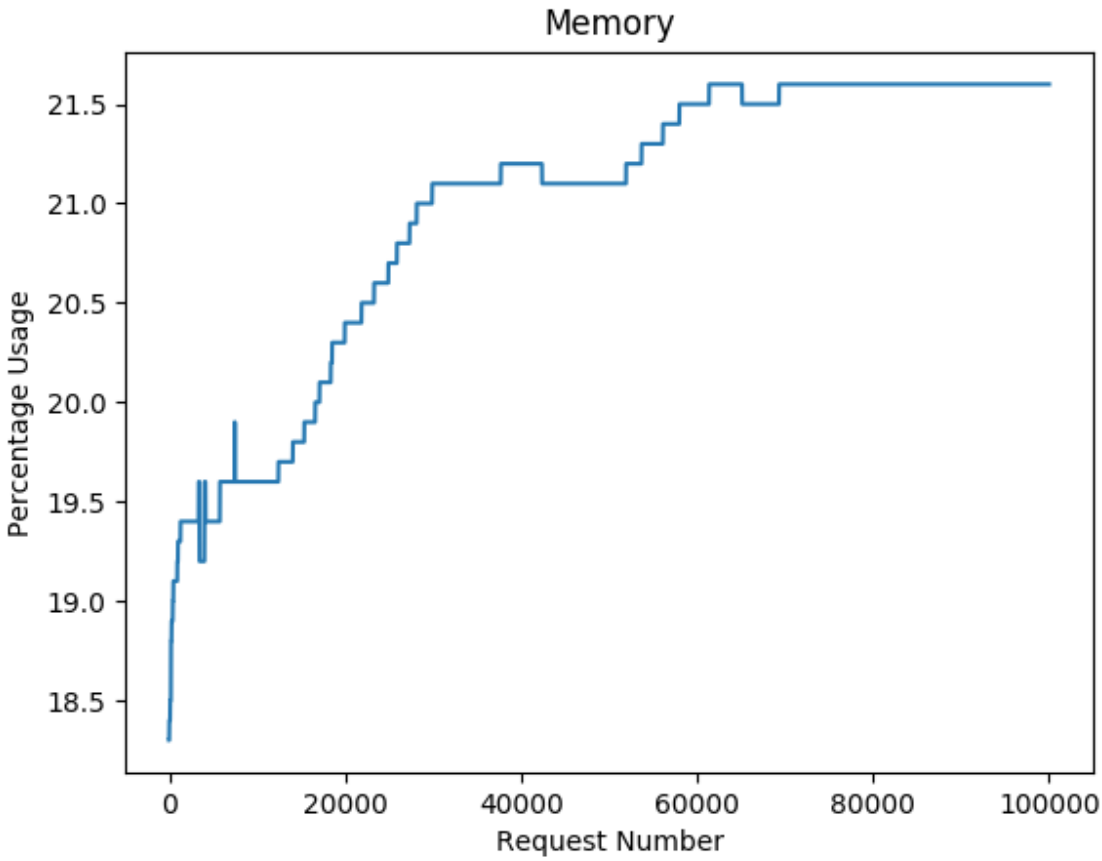

Figure 36: RESTCONF Memory Usage Results.

This single run of the fuzzing process took about an hour and fifteen minutes to complete as illustrated by Figure 37.

```
$ time restconfFuzzer.py distribution-karaf-0.5.2-Boron-SR2/bin/karaf

real   74m57.786s
user   35m30.912s
sys    23m14.260s
```

Figure 37: Time to Complete RESTCONF Fuzzing Run.

The trends in CPU and memory usage for the RESTCONF fuzzing run are quite similar to those of the OpenFlow fuzzing run; the variance in CPU usage is not particularly interesting but there

are three spikes in memory usage that occur before the twenty-thousandth request is sent. Inspecting the results.csv file reveals that these readings were taken after requests 3347, 5126, and 7431. We can reference these requests in the modifiedRequests.txt file with the code shown in Figure 38, where x is the request number referenced.

```
with open('modifiedRequests.txt', 'r') as f:
      data = f.read().split('#request')[1:]
print data[x]
```

Figure 38: Code to Reference Requests with RESTCONF Plugin.

Referencing these requests in the modifiedRequests.txt file allows us to determine that requests 3347, 5216, and 7431 are GET, POST, and DELETE requests respectively. Table 6 summarizes these requests.

Request number	Request method	Bytes
3347	GET	3304
5126	POST	59563
7431	DELETE	916

Table 6: Requests that Caused Spikes in Memory Usage.

Like with the OpenFlow fuzzing run, further inspection of the requests does not provide any compelling evidence as to why they would cause spikes in memory usage but we believe repeating the fuzzing process exclusively with the requests in question may be able to help answer this question.

CHAPTER IV

CONCLUSION

I. Summary

Our goal with this project was to provide a practical framework for finding vulnerabilities in SDN controllers. We successfully achieved this objective by developing a fuzzing tool with standard Python modules to fuzz the OpenFlow and RESTCONF plugins of the OpenDaylight controller. To develop this framework, we chose to use the OpenDaylight controller, because it is well-documented, and fuzzing, because it is an effective method for finding vulnerabilities in large software systems. We chose two controller features, odl-l2switch-all, a Southbound plugin and odl-restconf-all, a Northbound plugin, to fuzz with a smart, generation-based, blackbox fuzzing approach. This approach consisted of determining the structure of valid packets or requests to the controller, determining the fields within those packets or requests that could be varied, and varying those fields randomly.

Our hypothesis was that fuzzing these two OpenDaylight plugins would cause unexpected behavior within the controller that could be measured through careful observation of CPU usage, memory usage, and log output. There is some evidence to suggest that this was the case; we observed spikes in memory usage while testing both the odl-l2switch-all and odl-restconf-all features. It is not immediately clear why the packets or requests in question caused the observed spikes in memory usage. However, we believe the next step is to conduct multiple fuzzing runs exclusively with the packets or requests that caused spikes in memory usage as it may reveal patterns that can help to explain the spikes in memory usage.

II. Future Research

As mentioned, repeating the fuzzing process multiple times exclusively with the packets or requests that caused spikes in memory usage is a promising place to start for future research. Besides this, though, we believe our framework provides a good foundation for future research because it can be both extended and expanded. First, our methodology can be extended to more OpenDaylight features and more SDN controllers. There are a total of forty features listed by the OpenDaylight Wiki.[67] These features can be fuzzed in a similar manner to our methodology by determining the normal structure of input sent to those features, determining what parts of that input can be varied, and varying those parts randomly. Similarly, other SDN controllers such as Floodlight, OpenContrail, ONOS, and RYU, have their own implementations of Northbound and Southbound plugins, similar to OpenDaylight's OpenFlow and RESTCONF features, that may reveal vulnerabilities when subjected to the fuzzing framework outlined in our methodology.

Second, our methodology can be expanded to incorporate more advanced fuzzing tools, like Peach or AFL. Our modeled OpenFlow packets and RESTCONF requests can be leveraged to create Peach Pits that model those packets and requests in a similar way. Figure 39 and figure 40 illustrate Peach Pits that model an OpenFlow packet and a RESTCONF HTTP request respectively.

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://peachfuzzer.com/2012/Peach"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://peachfuzzer.com/2012/Peach ../peach.xsd">

  <DataModel name="DataModel">
    <Number name="version" size="8" value="4" endian="big" mutable="false"/>
    <Number name="type" size="8" value="0" endian="big" mutable="false"/>
    <Number name="length" size="16" value="64" endian="big" mutable="false"/>
    <Number name="xid" size="32" endian="big"/>
  </DataModel>
```

[67] "OpenDaylight Features List," OpenDaylight Project, accessed April 1, 2017, https://www.opendaylight.org/opendaylight-features-list.

```
  <StateModel name="StateModel" initialState="InitialState">
    <State name="InitialState">
      <Action type="output"><DataModel ref="DataModel"/></Action>
    </State>
  </StateModel>

  <Test name="Default">
    <StateModel ref="StateModel"/>
    <Publisher class="TcpClient">
      <Param name="Host" value="127.0.0.1"/>
      <Param name="Port" value="6653"/>
    </Publisher>
  </Test>

</Peach>
```

Figure 39: Peach Pit for a OpenFlow Packet.

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://peachfuzzer.com/2012/Peach"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://peachfuzzer.com/2012/Peach ../peach.xsd">

  <DataModel name="DataModel">
    <String value="POST /restconf/config/ HTTP/1.1\r\n" mutable="false"/>
    <String value="Host: 127.0.0.1:8181\r\n" mutable="false"/>
    <String value="Connection: keep-alive\r\n" mutable="false"/>
    <String value="Accept-Encoding: gzip, deflate\r\n" mutable="false"/>
    <String value="Accept: */*\r\n" mutable="false"/>
    <String value="User-Agent: python-requests/2.13.0\r\n" mutable="false"/>
    <String value="Content-Type: application/json\r\n" mutable="false"/>
    <String value="Content-Length: " mutable="false"/>
    <String>
      <Relation type="size" of="Payload"/>
    </String>
    <String value="\r\n" mutable="false"/>
    <String value="Authorization: Basic YWRtaW46YWRtaW4=\r\n" mutable="false"/>
    <Block name="Payload">
      <String value='{"address-tracker-config":{"observe-addresses-from":"'
mutable="false"/>
      <String/>
      <String value='","timestamp-update-interval":"' mutable="false"/>
      <Number name="timestamp-update-interval" size="64"/>
      <String value='"}}' mutable="false"/>
    </Block>
  </DataModel>

  <StateModel name="StateModel" initialState="InitialState">
    <State name="InitialState">
      <Action type="output">
        <DataModel ref="DataModel"/>
      </Action>
```

```
    </State>
  </StateModel>

  <Test name="Default">
    <StateModel ref="StateModel"/>
    <Publisher class="TcpClient">
      <Param name="Host" value="127.0.0.1" />
      <Param name="Port" value="8181" />
    </Publisher>
  </Test>

</Peach>
```

Figure 41: Peach Pit for a RESTCONF HTTP Request.

Although AFL is generally better suited to binary programs, it may be possible to incorporate AFL into our fuzzing process. Figure 41 is a simple Java class, figure 42 is a makefile to compile that Java class, and figure 43 is a set of commands to start the fuzzing process with AFL. Collectively they demonstrate how AFL can be used to fuzz a simple Java class. [68]

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
public class Testcase{
    public native void nativeCrash();
    public static void main(String args[]){
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try{
            String s = in.readLine();
            if(s == null || s.length() == 0){
                System.out.println("Hum?");
                System.exit(1);
            }
            if(s.charAt(0) == '0'){
                System.out.println("Looks like a zero to me!");
                    //System.exit(134); //Doesn't work, although it looks "the same"
in a bash $?, it's not the same, therefore:
                        new Testcase().nativeCrash();
            }
            else{
                System.out.println("A non-zero value? How quaint!");
```

[68] Floyd-fuh, "Floyd-fuh/AFL_GCJ_Fuzzing_Simple," GitHub, December 05, 2016, accessed April 1, 2017, https://github.com/floyd-fuh/AFL_GCJ_Fuzzing_Simple.

```
            }
            System.exit(0);
        }
        catch(IOException ioe){
            System.out.println("Hum?");
            System.exit(1);
        }
    }
}
```

Figure 40: A Simple Java Class.

```
Testcase:  Testcase.o crash.o
	afl-gcj -o Testcase \
	      Testcase.o crash.o -lstdc++ --main=Testcase

Testcase.o:  Testcase.class
	afl-gcj -c Testcase.class

Testcase.class: Testcase.java
	afl-gcj -C Testcase.java

Testcase.h: Testcase.class
	gcjh -cp . Testcase

crash.o: Testcase.h crash.cc
	g++ -c crash.cc

clean:
	rm -f Testcase Testcase.o crash.o Testcase.class Testcase.h
```

Figure 41: A Makefile to Compile the Java Class.

```
set -x
mkdir input
echo "AAA" > input/A
export AFL_DONT_OPTIMIZE=TRUE
make clean
make
./Testcase < input/A
echo "last command exit code: $?"
afl-fuzz -i input/ -o output -m 100 ./Testcase
```

Figure 42: Commands to Start AFL Fuzzing Process.

Two major challenges need to be overcome to incorporate AFL into our fuzzing process. The first challenge is to compile the OpenDaylight controller code with the GNU Compiler for Java

(GCJ). The second challenge is to redirect the OpenDaylight controller to read input from files instead of network sockets. Both challenges may be difficult to overcome because of the complexity of the controller code. If they are overcome, our modeled OpenFlow packets and RESTCONF requests can be leveraged to construct sample input for AFL. AFL can then use these as starting sample inputs to fuzz the OpenDaylight controller.

BIBLIOGRAPHY

Ahmad, Ijaz, Suneth Namal, Mika Ylianttila, and Andrei Gurtov. "Security in Software Defined Networks: A Survey." *IEEE Communication Surveys & Tutorials*, Vol. 17, No. 4, Fourth Quarter, 2015.

Al-Shaer, Ehab, and Saeed Al-Haj. "FlowChecker." *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration - SafeConfig '10*, 2010. doi:10.1145/1866898.1866905.

Benton, Kevin, L. Jean Camp, and Chris Small. "OpenFlow vulnerability assessment." *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*, 2013. doi:10.1145/2491185.2491222.

Berners-Lee, Tim, Roy Fielding, Larry Masinter. "RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax." IETF Tools. Accessed April 1, 2017. https://tools.ietf.org/html/rfc3986.

Bjorklund, Martin. "RFC 6020 - YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)." IETF Tools. Accessed April 1, 2017. https://tools.ietf.org/html/rfc6020.

Canini, Marco, Daniele Venzano, Peter Peresini, Dejan Kostic, and Jennifer Rexford. "A NICE way to test OpenFlow applications." *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

Cha, Sang Kil, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. "Unleashing Mayhem on Binary Code." *2012 IEEE Symposium on Security and Privacy*, 2012. doi:10.1109/sp.2012.31.

Dhawan, Mohan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. "SPHINX: Detecting Security Attacks in Software-Defined Networks." *Proceedings 2015 Network and Distributed System Security Symposium*, 2015. doi:10.14722/ndss.2015.23064.

"Flowgrammable." Flowgrammable. Accessed April 1, 2017. http://flowgrammable.org/.

Floyd-fuh. "Floyd-fuh/AFL_GCJ_Fuzzing_Simple." GitHub. December 05, 2016. Accessed April 1, 2017. https://github.com/floyd-fuh/AFL_GCJ_Fuzzing_Simple.

Ganesh, Vijay, Tim Leek, and Martin Rinard. "Taint-based directed whitebox fuzzing." *2009 IEEE 31st International Conference on Software Engineering*, 2009. doi:10.1109/icse.2009.5070546.

Godefroid, Patrice. "Random testing for security." *Proceedings of the 2nd international workshop on Random testing co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007) - RT '07*, 2007. doi:10.1145/1292414.1292416.

Godefroid, Patrice, Michael Y. Levin, and David Molnar. "SAGE: Whitebox Fuzzing for Security Testing." *Queue* 10, no. 1 (2012): 20. doi:10.1145/2090147.2094081.

Godefroid, Patrice, Adam Kiezun, and Michael Y. Levin. "Grammar-based whitebox fuzzing." *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08*, 2008. doi:10.1145/1375581.1375607.

Hinrichs, Timothy, Natasha Gude, Martin Casado, John Mitchell, and Scott Shenker. "Expressing and enforcing flow-based network security policies." *University of Chicago*, 2008.

Holler, Christian, Kim Herzig, and Andreas Zeller. "Fuzzing with code fragments." *Proceedings of the 21st USENIX Security Symposium*, 2012.

Hong, Sungmin, Lei Xu, Haopei Wang, and Guofei Gu. "Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures." *Proceedings 2015 Network and Distributed System Security Symposium*, 2015. doi:10.14722/ndss.2015.23283.

Hu, Hongxin, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. "FLOWGUARD: Building robust firewalls for software-defined networks." *Proceedings of the third ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '14*, (2014): 97–102.

Khurshid, Ahmed, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. "Veriflow." *ACM SIGCOMM Computer Communication Review* 42, no. 4 (2012): 467. doi:10.1145/2377677.2377766.

Kloti, Rowan, Vasileios Kotronis, and Paul Smith. "OpenFlow: A security analysis." *2013 21st IEEE International Conference on Network Protocols (ICNP)*, 2013. doi:10.1109/icnp.2013.6733671.

Kreutz, Diego, Fernando M. V. Ramos, and Paulo Verissimo. "Towards secure and dependable software-defined networks." *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*, 2013. doi:10.1145/2491185.2491199.

Leary, Mark. "SDN, NFV, and open source: the operator's view." Gigaom. March 19, 2014. Accessed April 1, 2017. https://gigaom.com/report/sdn-nfv-and-open-source-the-operators-view/.

Li, Dawei, Xiaoyan Hong, and Jason Bowman. "Evaluation of Security Vulnerabilities by Using ProtoGENI as a Launchpad." *2011 IEEE Global Telecommunications Conference - GLOBECOM 2011*, 2011. doi:10.1109/glocom.2011.6134465.

Jan Medved, Robert Varga, Anton Tkacik, Ken Gray. "OpenDaylight: towards a model-driven SDN controller architecture." *IEEE 15th international symposium on a world of wireless, mobile and multimedia networks (WoWMoM)*, 2014.

Metzler, Jim. "Understanding Software-Defined Networks." InformationWeek Reports. October 2012. Accessed April 1, 2017. http://reports.informationweek.com/abstract/6/9044/DataCenter/research-understanding-software-defined-networks.html.

Newsome, James and Dawn Song. "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software." *Proceedings of the 12th Network and Distributed System Security Symposium* (NDSS'05), 2005.

Neystadt, John. "Automated Penetration Testing with White-Box Fuzzing." Microsoft Developer Network. Accessed April 1, 2017. https://msdn.microsoft.com/en-us/library/cc162782.aspx.

Oehlert, Peter "Violating Assumptions with Fuzzing." *IEEE Security and Privacy Magazine 3*, no. 2 (2005): 58-62. doi:10.1109/msp.2005.55.

"OpenDaylight Wiki." OpenDaylight Project. Accessed April 1, 2017. https://wiki.opendaylight.org/view/Main_Page.

"OpenFlow Switch Specification." Open Networking Foundation. Accessed April 1, 2017. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.3.pdf.

"Peach Introduction." Deja vu Security. Accessed April 1, 2017. http://community.peachfuzzer.com/Introduction.html.

Porras, Philip, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. "A security enforcement kernel for OpenFlow networks." *Proceedings of the first workshop on Hot topics in software defined networks - HotSDN '12*, 2012.

"Random (Java Platform SE 7)." Java Platform, Standard Edition 7 API Specification. Accessed April 1, 2017. https://docs.oracle.com/javase/7/docs/api/java/util/Random.html.

Redwood, Owen W. "Lecture 8: Fuzzing Lecture 1." Florida State University. Accessed April 1, 2017. http://www.cs.fsu.edu/~redwood/OffensiveComputerSecurity/lectures.html.

Schwartz, Edward J., Thanassis Avgerinos, and David Brumley. "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been

Afraid to Ask)." *2010 IEEE Symposium on Security and Privacy*, 2010. doi:10.1109/sp.2010.26.

Scott-Hayward, Sandra, Sriram Natarajan, and Sakir Sezer. "A survey of security in software defined networks." *IEEE Communications Surveys & Tutorials*, 2015.

"SDL Process - Phase 4: Verification." Microsoft Developer Network. Accessed April 1, 2017. https://msdn.microsoft.com/en-us/library/windows/desktop/cc307418.aspx.

Sezer, Sakir, Sandra Scott-Hayward, Pushpinder Chouhan, Barbara Fraser, David Lake, Jim Finnegan, Niel Viljoen, Marc Miller, and Navneet Rao. "Are we ready for SDN? Implementation challenges for software-defined networks." *IEEE Communications Magazine 51*, no. 7 (2013): 36-43. doi:10.1109/mcom.2013.6553676.

Shalimov, Alexander, Dmitry Zuikov, Daria Zimarina, Vasily Pashkov, and Ruslan Smeliansky. "Advanced study of SDN/OpenFlow controllers." *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia on - CEE-SECR '13*, 2013. doi:10.1145/2556610.2556621.

Sherwood, Rob, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. "Flowvisor: A network virtualization layer." *OpenFlow Switch Consortium, Tech.Rep*, 2009.

Seungwon Shin and Guofei Gu. ‘‘Attacking software-defined networks: A first feasibility study.’’ *Proceedings of the 2nd Workshop Hot Topics Software Defined Networking*, 2013. pp. 1–2.

Shin, Seungwon, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. "Avant-Guard." *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*, 2013. doi:10.1145/2508859.2516684.

Son, Sooel, Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. "Model checking invariant security properties in OpenFlow." *2013 IEEE International Conference on Communications (ICC)*, 2013. doi:10.1109/icc.2013.6654813.

"The State of Mobile Application Security 2014-2015." Checkmarx and AppSec Labs. Accessed April 1, 2017. https://www.checkmarx.com/wp-content/uploads/2015/11/The-State-of-Mobile-Application-Security-2014-20151.pdf.

Wang, Tielei, Tao Wei, Guofei Gu, and Wei Zou. "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection." *2010 IEEE Symposium on Security and Privacy*, 2010. doi:10.1109/sp.2010.37.

Yan, Qai, F. Richard Yu, "Distributed denial of service attacks in software-defined networking with cloud computing." *IEEE Communications Magazine 53*, April 2015. pp. 52–59.

Yan, Qiao, F. Richard Yu, Qingxiang Gong, Jianqiang Li. "Software-defined networking (SDN) and distributed denial of service (DDoS) attacks in cloud computing environments: A survey, some research issues, and challenges." *IEEE Communications Surveys & Tutorials*, 2015.

Zalewski, Michal. "Technical "whitepaper" for afl-fuzz." Accessed April 1, 2017. http://lcamtuf.coredump.cx/afl/technical_details.txt

APPENDIX

OPENFLOW HEADER

Field	Bytes	Constraints	Python expression
Version	1	4	`Struct('! B').pack(4)`
Type	1	0-29	`Struct('! B').pack(type)`
Length	2	≥ 8	`Struct('! H').pack(8 + len(payload))`
Xid	4	None	`urandom(4)`

OPENFLOW PAYLOADS

I. ofptHello: No payload

II. ofptError

Field	Bytes	Constraints	Python expression
errorType	2	0-13	`Struct('! H').pack(randint(0, 13))`
errorCode	2	Depends on errorType	`Struct('! H').pack(errorCode)`
data	8 ≤ data ≤ Maximum length of packet - 8 -4	None	`urandom(randint(8, maxLength - 8 - 4))`

III. ofptEchoReq and ofptEchoRes

Field	Bytes	Constraints	Python expression
data (optional)	≤ Maximum length of packet - 8	None	`urandom(randint(0, maxLength - 8))`

IV. ofptExperimenter

Field	Bytes	Constraints	Python expression
experimenterId	4	None	`urandom(4)`
experimenterType	4	None	`urandom(4)`
data (optional)	≤ Maximum length of packet - 8 - 8	None	`urandom(randint(0, maxLength - 8 - 8))`

V. ofptFeatureReq: No payload

VI. ofptFeatureRes

Field	Bytes	Constraints	Python expression
dataPathId	8	None	`urandom(8)`
nBuffers	4	None	`urandom(4)`
nTables	1	None	`urandom(1)`
auxiliaryId	1	None	`urandom(1)`
pad	2	None	`urandom(2)`
capabilities	4	Only bits 0, 1, 2, 3, 5, 6, and 8 are variable	`Struct('! I').pack(generateBitFlags([0, 1, 2, 3, 5, 6, 8]))`
reserved	4	None	`urandom(4)`

VII. ofptGetConfigRes and ofptSetConfig

Field	Bytes	Constraints	Python expression
flags	2	0-3	`Struct('! H').pack(randint(0, 3))`
missSendLen	2	None	`urandom(2)`

VIII. ofptPacketIn

Field	Bytes	Constraints	Python expression
bufferId	4	None	`urandom(4)`
totalLen	2	None	`urandom(2)`
reason	1	0-2	`Struct('! B').pack(randint(0, 2))`
tableId	1	None	`urandom(1)`
cookie	8	None	`urandom(8)`
pad	2	None	`urandom(2)`
match	≤ Maximum length of payload - 18	Match constraints	`generateMatch(payloadLength - 18)`
data (optional)	14 ≤ data ≤ Maximum length of payload - 18	None	`urandom(randint(14, payloadLength - 18 - len(match)))`

IX. ofptFlowRemoved

Field	Bytes	Constraints	Python expression
cookie	8	None	`urandom(8)`
priority	2	None	`urandom(2)`
reason	1	0-3	`Struct('! B').pack(randint(0, 3))`
tableId	1	None	`urandom(1)`
durationSec	4	None	`urandom(4)`
durationNSec	4	None	`urandom(4)`
idleTimeout	2	None	`urandom(2)`
hardTimeout	2	None	`urandom(2)`
packetCount	8	None	`urandom(8)`

byteCount	8	None	`urandom(8)`
match	≤ Maximum length of payload - 40	Match constraints	`generateMatch(payloadLength - 40)`

X. ofptPortStatus

Field	Bytes	Constraints	Python expression
reason	1	0-2	`Struct('!B').pack(randint(0, 2))`
pad	7	None	`urandom(7)`
portID	4	4294967040, 4294967288-4294967295	`Struct('!I').pack(choice([4294967040] + range(4294967288, 4294967296)))`
pad2	4	None	`urandom(4)`
hwAddr	6	None	`urandom(6)`
pad3	2	None	`urandom(2)`
name	16	None	`urandom(16)`
config	4	Only bits 0, 2, 5, and 6 are variable	`Struct('!I').pack(generateBitFlags([0, 2, 5, 6]))`
state	4	0-15	`Struct('!I').pack(randint(0, 15))`
current	4	0-65535	`Struct('!I').pack(randint(0, 65535))`
advertised	4	0-65535	`Struct('!I').pack(randint(0, 65535))`
supported	4	0-65535	`Struct('!I').pack(randint(0, 65535))`

peer	4	0-65535	`Struct('! I').pack(randint(0, 65535))`
curSpeed	4	None	`urandom(4)`
maxSpeed	4	None	`urandom(4)`

XI. ofptPacketOut

Field	Bytes	Constraints	Python expression
bufferID	4	None	`urandom(4)`
inPort	4	0-4294967040	`Struct('! I').pack(randint(0, 4294967040))`
pad	6	None	`urandom(6)`
actionsLen	2	Length of actions field	`Struct('! H').pack(len(actions))`
actions (optional)	≤ Maximum length of payload - 16	Action contraints	`generateActions(payloadLength - 16 )`
data (optional)	14 ≤ data ≤ Maximum length of payload - 16 - length of actions	None	`urandom(randint(14, payloadLength - 16 - len(actions)))`

XII. ofptFlowMod

Field	Bytes	Constraints	Python expression
cookie	8	0-18446744073709551614 (the max value of a long long)	`Struct('! Q').pack(randint(0, 18446744073709551614))`
cookieMask	8	None	`urandom(8)`
tableId	1	0-254	`Struct('! B').pack(randint(0, 254))`

command	1	0-4	`Struct('!B').pack(randint(0,4))`
idleTimeout	2	None	`urandom(2)`
hardTimeout	2	None	`urandom(2)`
priority	2	None	`urandom(2)`
bufferID	4	None	`urandom(4)`
outPort	4	None	`urandom(4)`
outGroup	4	None	`urandom(4)`
flags	2	0-31	`Struct('!H').pack(randint(0,31))`
pad	2	None	`urandom(2)`
match	≤ Maximum length of payload - 40	Match constraints	`generateMatch(payloadLength - 40)`
instructions (optional)	≤ Maximum length of payload - 40 - length of match	Instruction constraints	`generateInstructions(payloadLength - 40 - len(match))`

XIII. ofptGroupMod

Field	Bytes	Constraints	Python expression
command	2	0 - 2	`Struct('!H').pack(randint(0,2))`
type	1	0-3	`Struct('!B').pack(randint(0,3))`
pad	1	None	`urandom(1)`
groupId	4	None	`urandom(4)`
buckets (optional)	≤ Maximum length of payload - 8	Bucket constraints	`generateBuckets(payloadLength - 8)`

XIV. ofptPortMod

Field	Bytes	Constraints	Python expression
port	4	None	`urandom(4)`
pad	4	None	`urandom(4)`
hwAddr	6	None	`urandom(6)`
pad2	2	None	`urandom(2)`
config	4	Only bits 0, 2, 5, and 6 are variable	`Struct('! I').pack(generateBitFlags([0, 2, 5, 6]))`
mask	4	Only bits 0, 2, 5, and 6 are variable	`Struct('! I').pack(generateBitFlags([0, 2, 5, 6]))`
advertise	4	0-65535	`Struct('! I').pack(randint(0, 65535))`
pad3	4	None	`urandom(4)`

XV. ofptTableMod

Field	Bytes	Constraints	Python expression
tableId	1	0-254	`Struct('! B').pack(randint(0, 254))`
pad	3	None	`urandom(3)`
config	4	3	`Struct('! I').pack(3)`

XVI. ofptMultipartReq

Types 0, 3, 7, 8, 11, and 13 only consist of type, flags, and pad fields.

Field	Bytes	Constraints	Python expression

type	2	0-13, 65535	`Struct('! B').pack(choice(range(14) + [65535]))`
flags	2	0-1	`Struct('! H').pack(randint(0, 1))`
pad	4	None	`urandom(4)`

A. Type = 1, 2

Field	Bytes	Constraints	Python expression
tableId	1	None	`urandom(1)`
pad	3	None	`urandom(3)`
outPort	4	0-4294967279	`Struct('! I').pack(randint(0, 4294967279))`
outGroup	4	None	`urandom(4)`
pad2	4	None	`urandom(4)`
cookie	8	None	`urandom(8)`
cookieMask	8	None	`urandom(8)`
match	≤ Maximum length of payload - 8 - 32	Match constraints	`generateMatch(payloadLength - 8 - 32)`

B. Type = 4

Field	Bytes	Constraints	Python expression
portNumber	4	0-4294967279	`Struct('! I').pack(randint(0, 4294967279))`
pad	4	None	`urandom(4)`

C. Type = 5

Field	Bytes	Constraints	Python expression
portNumber	4	0 - 4294967279	`Struct('! I').pack(randint(0, 4294967279))`
queueId	4	None	`urandom(4)`

D. Type = 6

Field	Bytes	Constraints	Python expression
groupId	4	None	`urandom(4)`
pad	4	None	`urandom(4)`

E. Type = 9, 10

Field	Bytes	Constraints	Python expression
meterId	4	None	`urandom(4)`
pad	4	None	`urandom(4)`

F. Type = 12

Field	Bytes	Constraints	Python expression
tableFeatures (optional)	≤ Maximum length of payload - 8	Table Feature constraints	`generateTableFeatures(payloadLength - 8)`

G. Type = 65535

Field	Bytes	Constraints	Python expression
experimenterId	4	None	`urandom(4)`
experimenterType	4	None	`urandom(4)`

XVII. ofptMultipartRes

Field	Bytes	Constraints	Python expression
type	2	0-13, 65535	`Struct('! B').pack(choice(range(14) + [65535]))`
flags	2	0-1	`Struct('! H').pack(randint(0, 1))`
pad	4	None	`urandom(4)`

A. Type = 0

Field	Bytes	Constraints	Python expression
mfrDesc	256	None	`urandom(256)`
hwDesc	256	None	`urandom(256)`
swDesc	256	None	`urandom(256)`
serialNum	32	None	`urandom(32)`
dpDesc	256	None	`urandom(256)`

B. Type = 1

Field	Bytes	Constraints	Python expression
tableId	1	None	`urandom(1)`
pad	1	None	`urandom(1)`
durationSec	4	None	`urandom(4)`
durationNSec	4	None	`urandom(4)`

priority	2	None	`urandom(2)`
idleTimeout	2	None	`urandom(2)`
hardTimeout	2	None	`urandom(2)`
pad2	6	None	`urandom(6)`
cookie	8	None	`urandom(8)`
packetCount	8	None	`urandom(8)`
byteCount	8	None	`urandom(8)`
match	≤ Maximum length of payload - 8 - 48	Match constraints	`generateMatch(payloadLength - 8 - 48)`
instructions (optional)	≤ Maximum length of payload - 8 - 48 - length of match	Instruction constraints	`generateInstructions(payloadLength - 8 - 48 - len(match))`

C. Type = 2

Field	Bytes	Constraints	Python expression
packetCount	8	None	`urandom(8)`
byteCount	8	None	`urandom(8)`
flowCount	4	None	`urandom(4)`

D. Type = 3

Field	Bytes	Constraints	Python expression
tableId	1	None	`urandom(1)`
pad	3	None	`urandom(3)`
activeCount	4	None	`urandom(4)`
lookupCount	8	None	`urandom(8)`
matchedCount	8	None	`urandom(8)`

E. Type = 4

Field	Bytes	Constraints	Python expression
portNumber	4	0-4294967279	`Struct('! I').pack(randint(0, 4294967279))`
pad	4	None	`urandom(8)`
rxPackets	8	None	`urandom(8)`
txPackets	8	None	`urandom(8)`
rxBytes	8	None	`urandom(8)`
txBytes	8	None	`urandom(8)`
rxDropped	8	None	`urandom(8)`
txDropped	8	None	`urandom(8)`
rxErrors	8	None	`urandom(8)`
txErrors	8	None	`urandom(8)`
rxFrameErr	8	None	`urandom(8)`
rxOverErr	8	None	`urandom(8)`
rxCrcErr	8	None	`urandom(8)`
collisions	8	None	`urandom(8)`
durationSec	4	None	`urandom(4)`
durationNSec	4	None	`urandom(4)`

F. Type = 5

Field	Bytes	Constraints	Python expression
portNumber	4	0-4294967279	`Struct('! I').pack(randint(0, 4294967279))`
queueId	4	None	`urandom(4)`

txBytes	8	None	`urandom(8)`
txPackets	8	None	`urandom(8)`
txErrors	8	None	`urandom(8)`
durationSec	4	None	`urandom(4)`
durationNSec	4	None	`urandom(4)`

G. Type = 6

Field	Bytes	Constraints	Python expression
groupStats (optional)	≤ Maximum length of payload - 8	Group Stats constraints	`generateGroupStats( payloadLength - 8)`

H. Type = 7

Field	Bytes	Constraints	Python expression
groupDescriptions (optional)	≤ Maximum length of payload - 8	Group Descriptions constraints	`generateGroupDescri ptions(payloadLengt h - 8)`

I. Type = 8

Field	Bytes	Constraints	Python expression
types	4	0-15	`Struct('! I').pack(randint(0, 15))`
capabilities	4	0-15	`Struct('! I').pack(randint(0, 15))`
maxGroups	4	None	`urandom(4)`
maxGroups2	4	None	`urandom(4)`
maxGroups3	4	None	`urandom(4)`
maxGroups4	4	None	`urandom(4)`

actions	4	Only bits 0, 11, 12, and 15-28 are variable	`Struct('! I').pack(generateBitFlags([0, 11, 12] + range(15, 28)))`
actions2	4	Only bits 0, 11, 12, and 15-28 are variable	`Struct('! I').pack(generateBitFlags([0, 11, 12] + range(15, 28)))`
actions3	4	Only bits 0, 11, 12, and 15-28 are variable	`Struct('! I').pack(generateBitFlags([0, 11, 12] + range(15, 28)))`
actions4	4	Only bits 0, 11, 12, and 15-28 are variable	`Struct('! I').pack(generateBitFlags([0, 11, 12] + range(15, 28)))`

J. Type = 9

Field	Bytes	Constraints	Python expression
meterStats (optional)	≤ Maximum length of payload - 8	Meter Stats constraints	`generateMeterStats(payloadLength - 8)`

K. Type = 10

Field	Bytes	Constraints	Python expression
meterConfigs (optional)	≤ Maximum length of payload - 8	Meter Config constraints	`generateMeterConfigs(payloadLength - 8)`

L. Type = 11

Field	Bytes	Constraints	Python expression
maxMeter	4	None	`urandom(4)`
bandType	4	None	`urandom(4)`
capabilities	4	None	`urandom(4)`

maxBands	1	None	`urandom(1)`
maxColor	1	None	`urandom(1)`
pad	2	None	`urandom(2)`

M. Type = 12

Field	Bytes	Constraints	Python expression
tableFeatures (optional)	≤ Maximum length of payload - 8	Table Features Constraints	`generateTableFeatures(payloadLength - 8)`

N. Type = 13

Field	Bytes	Constraints	Python expression
ports (optional)	≤ Maximum length of payload - 8	Port constraints	`generatePorts(payloadLength - 8)`

O. Type = 65535

Field	Bytes	Constraints	Python expression
experimenterId	4	None	`urandom(4)`
experimenterType	4	None	`urandom(4)`

XVIII. ofptBarrierReq and ofptBarrierReq: No payload

XIX. ofptQueueGetConfigReq

Field	Bytes	Constraints	Python expression
port	4	0-4294967039	`Struct('!I').pack(randint(0, 4294967039))`
pad	4	None	`urandom(4)`

XX. ofptQueueGetConfigRes

Field	Bytes	Constraints	Python expression
port	4	0-4294967039	`Struct('! I').pack(randint(0, 4294967039))`
pad	4	None	`urandom(4)`
queues (optional)	≤ Maximum length of payload - 8	Queue constraints	`generateQueues(payloadLength - 8)`

XXI. ofptRoleReq and ofptRoleRes

Field	Bytes	Constraints	Python expression
role	4	0-3	`Struct('! I').pack(randint(0, 3))`
pad	4	None	`urandom(4)`
generationId	8	None	`urandom(8)`

XXII. ofptGetAsyncReq: No payload

XXIII. ofptGetAsyncRes and ofptSetAsync

Field	Bytes	Constraints	Python expression
packetInMask	4	0-7	`Struct('! I').pack(randint(0, 7))`
packetInMask	4	0-7	`Struct('! I').pack(randint(0, 7))`
packetStatusMask	4	0-7	`Struct('! I').pack(randint(0, 7))`
packetStatusMask2	4	0-7	`Struct('! I').pack(randint(0, 7))`

flowRemovedMask	4	0-15	`Struct('!I').pack(randint(0,15))`
flowRemovedMask2	4	0-15	`Struct('!I').pack(randint(0,15))`

XXIV. ofptMeterMod

Field	Bytes	Constraints	Python expression
command	2	0-2	`Struct('!H').pack(randint(0,2))`
flags	2	0-15	`Struct('!H').pack(randint(0,15))`
meterId	4	0, 4294901760, 4294967293-4294967295	`Struct('!I').pack(choice([0,4294901760] + range(4294967293,4294967296)))`
meterBands (optional)	≤ Maximum length of payload - 8	Meter band constraints	`generateMeterBands(payloadLength - 8)`