2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN): SNS2018 - International Workshop on Security in NFV-SDN @ IEEE NFV-SDN 2018

RE-CHECKER: Towards Secure RESTful Service in Software-Defined Networking

Seungwon Woo, Seungsoo Lee, Jinwoo Kim and Seungwon Shin
KAIST, Daejeon, Korea
Email: {seungwonwoo, lss365, jinwoo.kim, claude}@kaist.ac.kr

***Abstract*—Over the years, Software-Defined Networking (SDN) has grown aggressively, and many SDN controller products have been released to date as not only open source projects but also commercial ones. Considering the adoption of SDN, the security of SDN components is an essential aspect that needs to be thoroughly investigated, so research in this area has been getting attention. However, despite growing interest in SDN security, SDN controllers are vulnerable to security vulnerabilities that have not yet been disclosed. Among them, we focus on RESTful services provided by SDN controllers because those services help users to implement useful network functions in a programmable way, so it can be a critical attack point to an adversary. Therefore, in this work, we try to find out vulnerabilities and bugs of the RESTful service implementation, which are powerful enough to jeopardize the entire network. To more efficiently detect those vulnerabilities and bugs, we introduce a framework called RE-CHECKER that can find the security holes of RESTful services in SDN controller. As a result, using RE-CHECKER, we found four bug types against three open source controllers: ONOS, Floodlight, and Ryu. To prove the feasibility and examine the potential impact of each vulnerability and bug, we demonstrate some vulnerable scenarios in the real SDN environments.**

***Index Terms*—SDN, Software-Defined Networking, REST APIs, SDN Security, RESTful services**

I. Introduction

In a traditional network, a network device consists of a control plane, which determines how to control network flows, and a data plane, which serves to forward or drop network packets by the policy of the control plane. However, since they are tightly coupled with each other, it is very difficult to add new functionalities to the devices. Thus, to overcome the shortcomings, Software-Defined Networking has been proposed and its key idea is the separation and centralization of the control plane to an SDN controller. These separated control plane and the data plane communicate with each other using an SDN protocol called OpenFlow [1]. With its separation and centralization, the SDN controller provides useful network services through northbound interfaces and exposes open APIs that allow developers to implement innovative SDN applications. For that reasons, it is far easier to flexibly and dynamically manage the entire networks in SDN than the traditional networks.

In addition, in order to allow an easy access to the network functionalities, today's SDN controllers provide RESTful services as an external web service. So, network administrators can employ the core services and various network functionalities in the SDN controller by calling REST APIs (i.e., HTTP protocol) instead of programming complex logic in the controller. Technically, the complicated and time-consuming programming task is simplified to CRUD operations of HTTP methods (i.e., GET, POST, etc.), so the administrators can easily query and manipulate network states. Inspired by these benefits, most popular SDN controllers provide the administrators with RESTful services to facilitate network management in a flexible way [2], [3].

Meanwhile, since the SDN controller is a key component that has powerful capabilities over the entire network infrastructure, there have been many concerns about a security aspect [4], [5]. In particular, the efforts to design a secure control layer within the controller have been proposed [6], [7]. One of the problems come from that uncontrolled SDN applications can call core services through the northbound interface without any constraints, therefore, the invocation of the applications can manipulate network states and even kill the controller itself. With these motivations, an SDN-specific penetration tool has also been presented [8], in order to investigate all possible scenarios that misuse the northbound interfaces. However, to the best of our knowledge, no one systemically explored the vulnerability of the RESTful services, despite the fact that the services can be easily exposed to an external adversary.

This paper introduces a security assessment framework for the RESTful services in SDN, called RE-CHECKER, which can automatically find out security holes related to the RESTful services in the SDN controller. Using RE-CHECKER, we specifically attempt to reveal the design flaws in the SDN controller, which are potentially powerful enough to put the entire network at risk. To prove the feasibility and examine the potential impact of each RESTful service vulnerability, we demonstrate some vulnerable scenarios in SDN. In addition, with the help of RE-CHECKER, we disclosed some security vulnerabilities or bugs in the implementation of RESTful services in three different SDN controllers: ONOS [9], Floodlight [10], and Ryu [11].

Roadmap. The remainder of this paper is structured as follows. In Section II, we present the motivation with a simple example and related works. In Section III, we introduce several considerations for finding vulnerabilities and present the overall design and implementation of RE-CHECKER. Then, we show some use cases of the bugs we found in Section IV. Finally, we discuss future works and conclude our paper in Section V.

978-1-5386-8281-4/18/$31.00 ©2018 IEEE

II. Motivation and Related Work

A. Motivation

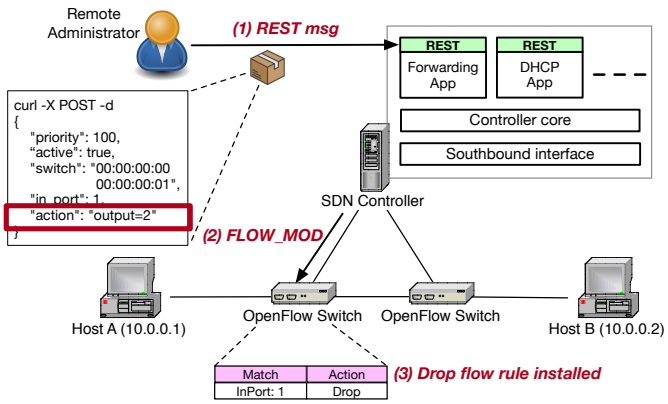

Fig. 1: A REST API misuse scenario that a network administrator misunderstands the `actions` parameter of installing a flow rule

Figure 1 shows a motivating example of how a simple typo made by a network administrator can affect network flows in SDN environments. In this motivating example, there is a simple network topology that consists of a Floodlight SDN controller and two OpenFlow switches. And, assuming that the administrator installs a flow rule on the switch by using a RESTful service provided by the forwarding application running on the controller so that Host A can communicate with Host B, he should build a syntax-appropriate REST message and then send it to the controller through the HTTP protocol.

The process of installing the flow rule on the switch through the RESTful service is as follows. First, the administrator makes a REST message that is composed of some resources (URI, method, etc.) in the HTTP header and the flow rule information in the data field according to the format required by the SDN controller. Next, after he sends the message to the RESTful service, the service verifies if the message corresponds with its specification. If the REST message is legal, the controller creates a FLOW_MOD message[1] and then sends it to the switch. After that, the RESTful service returns a message with the status code `200` (success). Otherwise, the service returns an error code `404` without installing the flow rule (fail).

In Figure 1, the network administrator makes the REST message with an `action` parameter and sends it to the RESTful service of the forwarding application (1). However, the action parameter is undefined in the Floodlight specification [3]. In fact, he should have made the message with `actions` parameter (not action) according to the specification. The problem here is that the Floodlight controller does not verify the undefined parameter, and instead sends a flow rule with an empty value of actions parameter to the switch (2), so the drop flow rule is installed on the switch (3). This is not intended one by the administrator, and more seriously, the controller returns a result message indicating that the requested flow rule was successfully installed (i.e., code `200`), so the administrator believes that the requested flow rule (i.e., the forwarding rule) was properly installed. However, the actually installed flow rule is the drop flow rule and it disrupts the communication between the two hosts by dropping all the packets from Host A.

[1]It is one of the OpenFlow message types [1] and used to manage flow rules on the OpenFlow switches

B. Related Work

As the attention to SDN has been growing, there have been some studies on the security of SDN. Hong et al. found security holes pertaining to the topology fabrication attacks in SDN and provided the defense mechanism [12]. Lee et al. proposed a penetration testing framework for the overall components in SDN [8]. BEADS, on the other hand, developed a framework that automatically generates various test scenarios specific to OpenFlow messages for SDN [13]. However, these studies do not consider the RESTful services on the SDN controller.

In the case of the security issues related to RESTful services in SDN controllers, Xiao et al. found various XSS vulnerabilities in web services provided by not only open source controllers (ONOS and Floodlight) but also commercial controllers (HPE VAN, etc.) [14]. CONGUARD showed a new attack that leverages harmful race conditions in the Floodlight and OpenDaylight by using REST APIs [15]. However, while they found the vulnerabilities and attacks in an ad-hoc manner, we suggest a framework that can automatically discover the security holes of the RESTful services in the SDN controller.

III. RE-CHECKER

In this section, we present the design of RE-CHECKER to effectively find out the security vulnerabilities or implementation bugs in the RESTful services. Thus, we first describe the design considerations and the major components of RE-CHECKER, and then, we briefly explain the implementation.

A. Design Considerations

We assume that two misuse scenarios are possible: (1) a network administrator misconfigures the RESTful services by accident as shown in Section II, and (2) an external adversary may access the RESTful services and tries to exploit them by injecting malicious inputs. Since one of the common causes of numerous network problems is the misconfiguration by human factors [16], it is possible that the administrator may send an abnormal message (e.g., erroneous syntax or range of valid input) to the SDN controller. Also, the RESTful services can be exploited by an attacker, given that the services typically are launched on 8181 or 8080 ports in most SDN controllers. We observed that ONOS uses basic access authentication [17] when receiving the inputs, while Floodlight and Ryu do not. But here, we argue that a lot of network systems surprisingly employ default username/password [18], so the ONOS's authentication can also be broken.

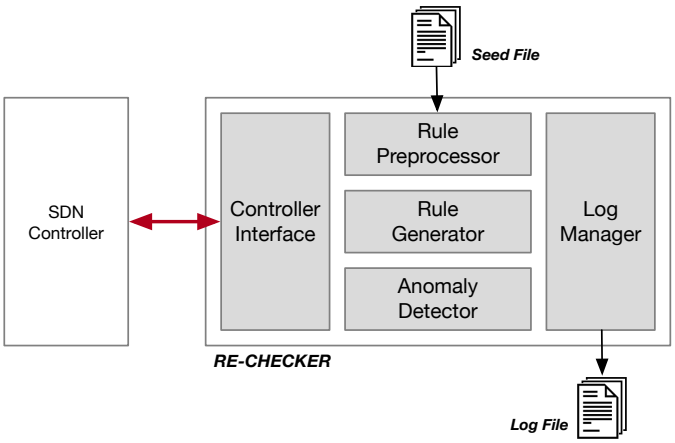

Fig. 2: The overall design of RE-CHECKER

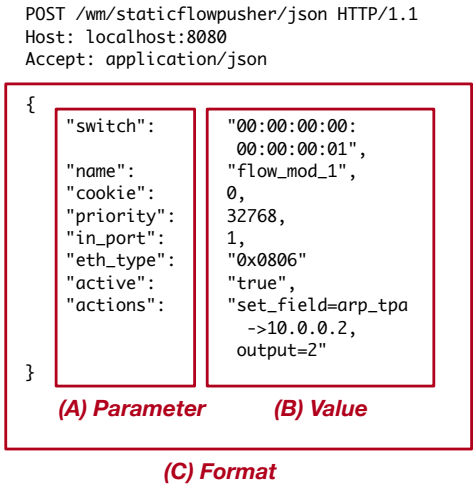

Fig. 3: The REST message example of installing a flow rule in Floodlight controller

To more efficiently find these security issues of misusing the REST APIs, we describe several design considerations as follows. First, *(i) it should be aimed at the critical RESTful services.* For example, most SDN controllers provide the RESTful services for managing the flow rules on the switches (e.g., Floodlight), or for configuring SDN applications as well (e.g., ONOS). Among them, we selected the RESTful services related to *flow rule* as a target RESTful service, which is a significant network function in SDN. Next, *(ii) it should be highly automated to efficiently make REST messages.* Since manually making and sending all the malformed messages is an impractical and time-consuming job, we adopt a black-box fuzzing technique. By automatically putting arbitrary values in JSON format into various fields (*parameter*, *value* and JSON *format*), we can efficiently and randomly create malformed REST messages, which may lead the SDN controllers to unexpected behavior such as controller crash or installing undesired flow rule.

Unexpected behavior: Since analyzing all the result information after each test is inefficient, we need to define the unexpected behavior, which can detect the test case as an abnormal one. In general, after requesting the RESTful services, we can see whether the REST message is processed properly by looking at its response message. For example, when an abnormal REST message containing an out-of-range value is sent to the SDN controller, the flow rule can be installed due to the problem of the controller and a `200` status code can be returned in the response message. Thus, we first make a malformed request and get a response message. If the response message has `200` status code, we determine the requested flow rule as suspicious, check whether the flow rule is installed, and then compare the installed flow rule with the requested flow rule. Finally, if the comparison result shows that the two rules are different, it is determined to be unexpected behavior. Additionally, we periodically check the state of the SDN controller and switches to discover if there are any critical problems that fall into the unexpected state, such as the controller crash or switch-performance degradation.

B. RE-CHECKER Components

RE-CHECKER consists of five main components; controller interface, rule preprocessor, rule generator, anomaly detector, and log manager as shown in Figure 2.

Controller Interface: The controller interface is used to communicate with a target SDN controller. Since the RESTful services are provided via HTTP protocol, the interface sends various HTTP requests to the target for installing flow rules and getting information about flow rules from the switches.

Rule Preprocessor: The rule preprocessor receives a *seed file* from a network administrator and preprocesses it so that the file can be parsed by the rule generator. The seed file has a default HTTP header and payload based on a JSON format consisting of `parameter` and `value`, as shown in Figure 3. When starting RE-CHECKER, the administrator can also determine which parts should be randomized among them.

Rule Generator: This component creates a number of malformed REST messages by putting arbitrary values in multiple fields based on the preprocessed rule. In the case of the parameter and value field, the generator sets all possible values (e.g., boolean, string, integer, etc.) in each field. Regarding the format part, it randomly writes an arbitrary string that destroys the syntax of the format.

Anomaly Detector: The anomaly detector verifies whether the SDN controller properly handles the REST messages created by the rule generator. It determines the unexpected behavior by using RESTful services to get and compare information for a particular network element. For example, the anomaly detector uses a REST message “GET /flows/” to query the flow rule information from ONOS’s database and then compares it with the requested flow rule, which is generated in the previous step. If the two flow rules are different from each other, it is determined as the unexpected behavior. Also, the anomaly detector can detect the controller crash or the switch performance degradation by communicating with the controller and switch.

Log Manager: The log manager leaves all the issues from each component in a *log file*. For example, if the anomaly detector discovers the unexpected behavior, the log manager writes both the requested flow rule and installed one on the actual switch to the log file. The network administrator can later check the log file to see which messages are causing the problem.

C. Implementation

We implemented RE-CHECKER in approximately 450 lines of Python codes and leveraged *pyjfuzz* [19], an open source JSON fuzzer to generate malformed JSON inputs.

IV. Evaluation

This section provides an evaluation of the REST API implementation in each target SDN controller. We tested ONOS, Floodlight, and Ryu controllers, which are popular open source SDN controllers nowadays. The ONOS version is 1.14.0, the Floodlight version is 1.2, and the Ryu version is 4.26 respectively. First, we categorize security vulnerabilities and bugs that we have found into four types, and then we introduce the use case of each type.

TABLE I: The test results of each bug for the RESTful service that installs a flow rule provided by each controller: ONOS, Floodlight, and Ryu

	ONOS	Floodlight	Ryu
Arithmetic Overflow and Underflow	O	O	X
Invalid Value Type	O	O	X
Unchecked Prerequisite	O	O	X
Undefined Parameters	O	O	O

A. Bug type categorization

Arithmetic Overflow and Underflow: This case is an overflow and underflow bug for the numeric types. By default, the type and range of each parameter in the flow rule are defined in OpenFlow specification [1]. Therefore, if one parameter needs to get a numeric value, the SDN controller should verify if the parameter receives an out-of-range value. For example, according to the OpenFlow specification, the `priority` should have the numeric value, and its range is from 0 to 65535. However, if a network administrator may put a very large number (e.g., 65536) into the priority and the controller does not check the range of it, the priority can be 0 due to the overflow.

Invalid Value Type: Similar to the previous case, if an SDN controller receives an invalid value type for a particular parameter, the controller should handle the exception appropriately. For example, `isPermanent` parameter is the boolean type supported in the ONOS controller. If it is set to true, an installed flow rule should remain permanently. However, if the controller receives a string value "True" for that parameter, it is encoded into a boolean value *false* incorrectly, so the non-permanent flow rule is sent to the switch.

Unchecked Prerequisite: The OpenFlow specification also describes various prerequisites for each parameter. For example, if a network administrator wants to install a flow rule for IP packets, he should specify that the Ethernet type of the flow rule is IPv4, and otherwise, an SDN controller should return an error. In another example, in order to use `group` parameter supported from OpenFlow 1.3, the corresponding group should exist in a switch's group table.

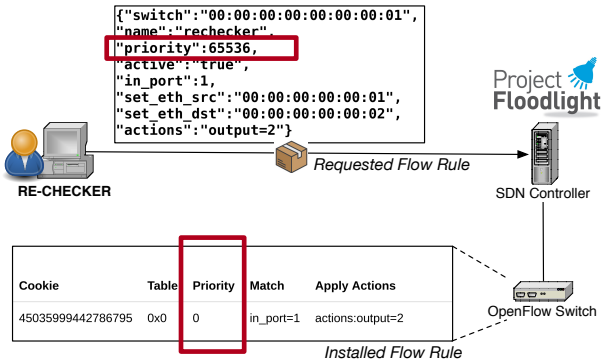

Fig. 4: Arithmetic Overflow in Floodlight

Undefined Parameters: If a network administrator builds a REST message with undefined parameters and sends it to the SDN controller, the controller should return an error message with the proper reason. However, as a result of RE-CHECKER testing the controllers, none of ONOS, Floodlight, and Ryu provide any exception handling for the undefined parameters.

Table I shows the test result whether each controller has the implementation bugs of the RESTful service related to the flow rule against each bug category. In the case of the ONOS and Floodlight controllers, they have the bugs for all the types. However, in the case of the Ryu controller, only undefined parameter bug was found because unlike the ONOS and Floodlight controller, Ryu controller is a lightweight python-based that does not store the state of the flow rule information. On the other hand, since the ONOS controller manages the internal database by periodically checking the switches in order to provide data consistency and fault-tolerant services, it can cause a little more serious internal problems discussed in Section IV-B2.

B. Use cases

1) Arithmetic Overflow and Underflow: It is an effective bug in ONOS and Floodlight controller, and we show a use case targeting Floodlight controller, which is caused by not checking an out-of-range value of `priority` parameter. As mentioned before, the priority value has to be set from 0 to 65535 because it is an unsigned short type. Therefore, if the input is out-of-range value, an error message should be returned by the controller. However, when the network administrator fills the priority with an out-of-range value to install a high priority flow rule as shown in Figure 4, the installed flow rule has the minimum priority (i.e., 0). The reason is that the controller does not check the range and arithmetic overflow occurs in the priority field. Therefore, since the flow rule has the lowest priority, some matched packets may be delivered to the undesired port number on the switch.

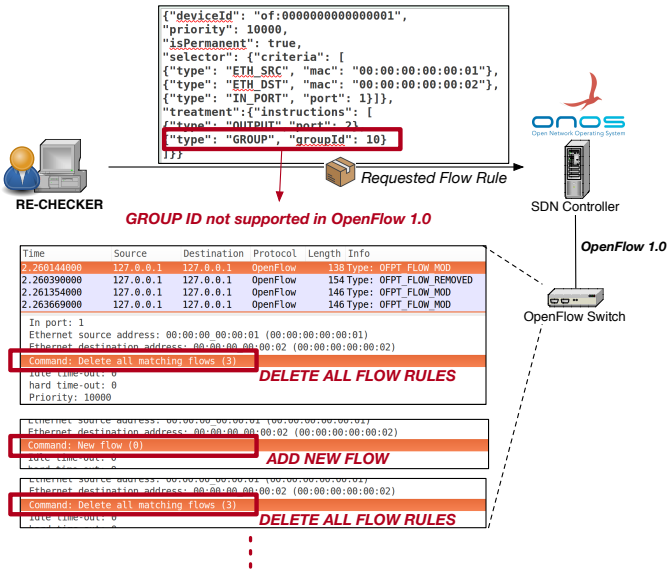

Fig. 5: Unchecked Prerequisite in ONOS

2) Unchecked Prerequisite: It is a possible bug in ONOS and Floodlight controllers, and here we show a bug scenario targeting ONOS controller, which results from not checking the prerequisites related to `group`. When installing a flow rule, since each OpenFlow version has different supported features, a network administrator should consider which OpenFlow version is used between an SDN controller and switches. However, assuming that the administrator misunderstands that the OpenFlow version is set to 1.3 but actually 1.0, an undesired flow rule can be installed on the switch. For example, the `group` feature is not supported in OpenFlow 1.0. So, the controller has to reject it with a message indicating that it is the unsupported feature in OpenFlow version 1.0.

However, the REST API implementation of the ONOS controller accepts the message and instructs the switch to install the flow rule as shown in Figure 5. In addition, the controller stores the flow rule with `group` in its internal storage, while it actually delivers the flow rule including `drop` instead of the group if there is no `output`. As a result, different flow rules are placed between the ONOS controller and the switch. Also, we found that the process of removing and re-installing the installed flow rule is repeated at intervals of 5 seconds infinitely because ONOS determines that the rule is not installed. If the anomaly that we found is reproduced over and over, the performance of the SDN controller can be greatly reduced, which can cause a catastrophic problem for the overall network performance.

V. Conclusion

Nowadays SDN controllers provide useful network functions through RESTful services, so anyone can efficiently manage their network policy and network state. However, there have been remained many security-relevant problems with the RESTful services, and the issues have not been addressed until now. Therefore, in this paper, we introduce RE-CHECKER that provides a mechanism to find out the implementation vulnerabilities or bugs of the RESTful services offered by various SDN controllers using black-box fuzzing. We have found four bug types and demonstrated some vulnerable bugs.

We expect that RE-CHECKER will be able to find bugs in other SDN controllers such as OpenDaylight [20] and commercial products in the future work. Furthermore, since SDN controllers have other RESTful services as well not only the flow management, we plan to extend RE-CHECKER so that it can find the vulnerabilities in those services.

Acknowledgment

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

References

[1] "Openflow switch specification: Version 1.3.0," 2012, https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf.
[2] "Onos rest api documentation," https://wiki.onosproject.org/display/ONOS/Appendix+B%3A+REST+API.
[3] "Floodlight rest api documentation," https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343539/Floodlight+REST+API.
[4] D. Kreutz, F. M. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.
[5] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "Sdn security: A survey," in *Future Networks and Services (SDN4FNS), 2013 IEEE SDN For*. IEEE, 2013, pp. 1–7.
[6] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, "Rosemary: A robust, secure, and high-performance network operating system," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*, November 2014.
[7] C. Yoon, S. Shin, P. Porras, V. Yegneswaran, H. Kang, M. Fong, B. O'Connor, and T. Vachuska, "A security-mode for carrier-grade sdn controllers," in *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM, 2017, pp. 461–473.
[8] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras, "Delta: A security assessment framework for software-defined networks," in *Proceedings of NDSS*, vol. 17, 2017.
[9] ONF, "Onos project, 1.14.0, 2018," https://onosproject.org/.
[10] Floodlight, "1.2, project floodlight, 2016," http://www.projectfloodlight.org/floodlight.
[11] Ryu, "4.26, ryu, 2018," https://osrg.github.io/ryu/.
[12] K. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, February 2015.
[13] S. Jero, X. Bu, C. Nita-Rotaru, H. Okhravi, R. Skowyra, and S. Fahmy, "Beads: Automated attack discovery in openflow-based sdn systems," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 311–333.
[14] X. Feng, H. Jianwei, and L. Peng, "Hacking the brain: Customize evil protocol to pwn an sdn controller," DEFCON 26, 2018.
[15] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, "Attacking the brain: Races in the sdn control plane," in *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017, pp. 451–468.
[16] J. Networks, "What is behind network downtime?" 2008.
[17] "Rfc7617: Http authentication: Basic and digest access authentication," https://tools.ietf.org/html/rfc2617.
[18] A. Cui and S. J. Stolfo, "A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan," in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010.
[19] mseclab, "Pyjfuzz," https://github.com/mseclab/PyJFuzz.
[20] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven sdn controller architecture," in *2014 IEEE 15th International Symposium on*. IEEE, 2014, pp. 1–6.