Raspberry Pi Testbed for Software-Defined Networks

Jonathan DeFreeuw
Bradley Department of Electrical and
Computer Engineering
Virginia Tech
Blacksburg, VA 24060
Email: defreeuw@vt.edu

***Abstract*—Software-Defined Networking (SDN) has quickly emerged as a useful alternative to coupled data-control planes for management and configurations of networks, especially in corporate data centers. These configurations can be deployed to secure a network through analysis and redirection. However, SDN-enabled hardware is currently expensive and out of reach for the average consumer without firmware modifications to home routers. Instead, this paper explores utilizing a $35 computer, the Raspberry Pi, to act as an OpenFlow switch to communicate with an OpenFlow controller. To test our switch and controller, we use a POX SDN controller to mitigate the password brute-forcing of an Internet-connected camera.**

I. Introduction

In a traditional computer network, hosts communicate through the use of switches and routers, which handle the forwarding and routing functions. Switches exist on layer 2 of the network stack, using hardware media access control (MAC) addresses to determine the next hop for a packet. Routers lie in layer 3, using IP address to connect networks together. Each of these devices maintain their own configurations, which administrators update manually or through scripts. In either case, management requires significant thought to allow a secure and operational network, particularly in enterprise-level networks.

Software-Defined Networking (SDN) tackles the challenge of increased complexity in expanding networks, by separating the computation of routes, called the *control plane*, from the physical forwarding of packets, called the *data plane*, creating a single management tool in the SDN controller. Through this controller, network administrators can programatically configure their network's hardware switches by creating the *flow rules*. These rules are installed on *flow tables*, and can be updated in real-time in reaction to new packets, or with update statistical information that is queried from the switches. The switch and controller communicate through a standard interface, the most popular being OpenFlow [1].

The centralized controller allows for a more flexible distribution of configurations on a wide network by opening an API to interact with the switch. Administrators develop SDN applications to create new rules for the network such as maintaining quality-of-service levels [2], tracking web statistics [3], or denial-of-service mitigation [4]. Controller frameworks are developed in multiple programming languages, including Python (POX, Ryu), Java (OpenDaylight, Floodlight), and C++ (NOX). This ability to effectively program a network reveals numerous possibilities for network security. An SDN makes way for security not just on the endpoints or edges of a network, but also in the transmission of every packet sent through it.

However, the benefits of a secure software-defined network are out of reach for many Internet users in their homes, as the cost of OpenFlow-enabled switches prevent them from expanding outside of the corporate environment. Recent innovations such as the Zodiac FX or WX [5] have laid groundwork for future consumer-devices, but are still significantly more expensive than a standard switch. Instead, researchers have developed virtualized forms of these switches, such as OpenVSwitch [6] and LINC [7].

The author explores virtualized network infrastructure on commodity hardware through the use of OpenVSwitch and the Raspberry Pi 3. The Raspberry Pi is an ARM-based micro computer running a Linux Debian-based operating system. This OpenFlow-enabled Raspberry Pi will be used to secure an Internet-connected surveillance camera, running an HTTP web server for streaming and management.

In Section II, we look at existing research in security through software-defined networks, while in Section III we examine the OpenFlow protocol. Later in Sections IV.a and IV.b, we discuss software and hardware implementations of our system. In Section V, we test the hardware system in a live network.

II. Related Works

This section reviews relevant research in security through software-defined networking and the use of the Raspberry Pi for network research.

A rapidly growing and still vulnerable field of devices is the Internet of Things (IoT). The IoT is a broad name for given to the network of Internet-connected devices, whether they be cameras, routers, baby monitors, or washing machines. More devices on the Internet means more vulnerabilities and more devices for botnets. The growth of botnets in the last few years have required innovative denial-of-service mitigations, and the SDN has been shown to be an efficient technique. With DefenseFlow, researchers query network devices for flow statistics that may reveal signs of a DoS attack [8]. In the event of a possible attack, traffic is diverted to an analysis program.

Anomaly detection algorithms have been deployed using software-defined networks to provide more agile event management in a network. Mehdi et. al. use four algorithms [9]:

- Threshold Random Walk with Credit Based Rate Limiting (TRW-CB) — network analyzes number of failed connections and their responses (TCP handshakes) to detect scanning worms
- Rate-Limiting — use a baseline of machine-to-machine communication and isolate any traffic that stands out as too rapid
- Maximum Entropy Detector — similar to rate-limiting, but does more in depth comparison using entropy of the last *t* seconds of traffic
- NETAD — analyze first packets of a connection, and use that data to determine if there is an anomaly, otherwise forward as normal

In [10], the authors develop a feature-selecting algorithm for anomaly detection based on how bats identify prey. With their algorithm, they are able to classify attacks with reasonable certainty, including DoS and probe attacks. Bao et. al. deployed an anomaly detection and prevention system in an SDN using the J48-Tree algorithm. Later, they deployed the algorithm on an FPGA to act as an OpenFlow switch on its own.

Other researchers have looked to the viability of the Raspberry Pi as a means of deploying software-defined networks. In [11], the authors compare network throughput of a Raspberry Pi based SDN to other implementations like Mininet and NetFPGA. They show that performance is almost as the same in comparison to NetFPGA. Asghar et. al. compare performance of an OpenWRT OpenFlow router with a Raspberry Pi switch, using the Ryu and Floodlight controllers. They found that there is significant packet dropping in the Raspberry Pi under load in some scenarios, while performing better than the router in others. They conclude that because they perform similarly on average that the limiting factor is OpenVSwitch, a dependency in both systems [12].

An interesting application of a Raspberry Pi system is for rapid deployment of a network in an emergency situations. The authors in [13] test the use of a software-defined Raspberry Pi network for disaster relief networks, where the need for Internet access and available databases is critical. They utilize Docker containers to easily mobilize and configure network functions.

III. OpenFlow Protocol

In this section, we examine the OpenFlow protocol, one of the most utilized interfaces for communication between an SDN controller and its switches.

While there are no standards defined by the IETF, RFC 7149 [14] and RFC 7426 [15] discuss the topic of software-defined networking. However, in 2008, McKeown et. al. published the OpenFlow white paper, defining the encapsulation techniques for their protocol [1]. In 2009, Heller et. al. published a white paper with specifications for Open Flow switches [16], which has grown and updated to become the standard for OpenFlow-enabled switches. The OpenFlow project has then grown to form the Open Networking Foundation, which has developed tools like the Open Networking Operating System (ONOS).

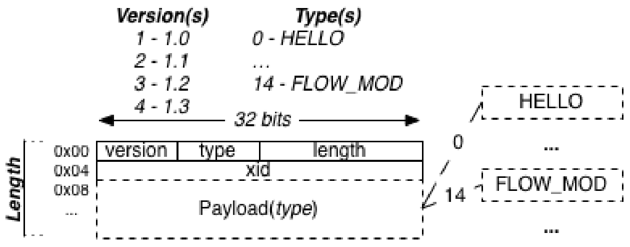

Fig. 1. OpenFlow packet header

The OpenFlow protocol defines a few different header values that are the same through versions:

- *version* — the version of OpenFlow running (1-5 for 1.0-1.4)
- *type* — defines the type of message being sent (`Hello`, `PacketIn`)
- *length* — where the end of the message is
- *xid* — a unique transaction identifier used to relate request to reply
- *payload* — a type-specific data sent from the switch

OpenFlow communication depends on a few different message types, some more than others. Listed below are some of the message types used in the software-defined network created for this paper:

- `Hello` — handshake message done between switch and controller, used to determine the OpenFlow version for communication
- `PacketIn` — when a switch does not contain a flow related to a specific packet (flow miss), then the switch sends this message with data related to the packet, often times the packet itself encapsulated in an OpenFlow packet, to the controller.
- `FlowMod` — a controller modifies the flow rules on a switch using this message type. Flows can match on TCP/IP values, Ethernet values, or port number on the switch.
- `StatsRequest` — used by the controller to request information about a particular flow, such as packet or data count
- `StatsReply` — used by the switch to relay information about a particular flow, usually in response to `StatsRequest` message from the controller

IV. Software and Hardware Experiment

For this experiment, the author proposed a private network connected to the Internet through a NAT router. In this network, there are five separate components with unique roles for the purpose of this research:

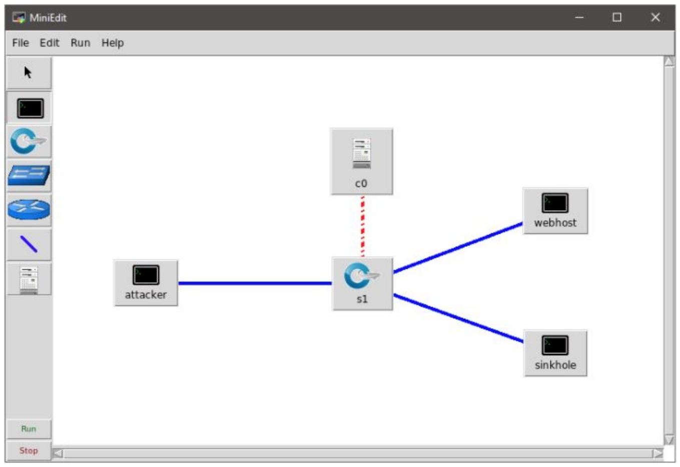

Fig. 2. Sample topology as shown in MiniEdit

1) Switch — An OpenFlow-enabled switch that forwards packets to and from hosts connected to it. It communicates with the Controller for flow rules.
2) Controller — A POX controller with applications for standard routing, flow modifications for the Web Server, and flow statistics for anomaly detection. Communicates with the Switch through OpenFlow.
3) Webhost — A simple HTTP server connected to the Switch
4) Attacker — A malicious actor within the network that attempts to access secure information on the Web Server
5) Sinkhole — A machine within the network that captures and saves all data relayed to it from the Switch

As an application for this network, we look to apply rate-limiting concepts to secure our webhost. Our proposed threat model focuses primarily on attackers with low-resources and effort, looking to brute-force their way into an accessible web server. Therefore, we focus on the number of packets transmitting to the server per second as a means of detecting anomalies.

A. Software Experiment

To begin building the system, the author used Mininet, a virtual network testbed written in Python (http://mininet.org). It includes a Python API to script the setup of virtual networks, and a command-line interface to interact with the network in real-time. Mininet can operate with different OpenFlow-enabled switches, but for the purposes of this experiment, the system used the default, OpenVSwitch. Mininet allows multiple controllers, switches, and hosts to be create and connected in complicated network topologies. In Figure 2, the system topology is shown using the MiniEdit tool, which is a graphical interface for Mininet. A virtual machine is available for Mininet, which also includes the dependencies for OpenFlow and the Python SDN controller called “POX”.

For this experiment, POX was chosen for its ease in rapid development, and its availability in the Mininet VM. The POX repository includes sample code and controllers such as a layer 2 forwarding switch controller and an ARP responder. The POX controller written for the software prototype had three components initially: flow control for TCP/IP traffic to and from hosts, an ARP module that responded to ARP requests with known locations, and a flow statistics module that queried the switch for information about packet stats to the Webhost. The controller was configurable at runtime to allow different rate limits, query frequencies, and hosts and services to be protected. Any other communication going to the webhost not on the specified port was dropped.

Using the Mininet hosts, we can run different bash commands as we would on the virtual machine, including Python commands and HTTP queries. In our attacker host, we run a bash script to query an HTTP server hosted on our webhost. The server is a simple Python server that gives access to the local file system, but can only handle a limited number of requests per second before failing. This was effective in determining whether the attack mitigation was successful or not. Running the attack without the rate limiting controller crashes the webhost’s server. With the controller running and the switch connected to it, the switch would relay flow statistics about the communication between the attacker and the webhost using `FlowStatsReceived` messages, and forward packets to the sinkhole. The sinkhole ran the `tcpdump` utility to capture any traffic not intended for it to use. After an idle time, the flow would be re-enabled and communication could resume.

B. Hardware Experiment

Because of the Raspberry Pi’s ease of use and Linux architecture, both the controller, attacker, and sinkhole could be ported from the Mininet VM to their respective hardware hosts. The webhost was attached to the network and set to a static IP. However, setup of the switch required significant configuration changes and experimentation.

OpenVSwitch is a network function virtualization tool that allows a user to create and connect virtual bridges and ports in a virtual network. A virtual bridge is connected to the network stack of the host, and virtual ports are connected to the bridge. In common uses of OpenVSwitch, the virtual ports would be connected to other virtual switches or virtual machines, creating a software-based network topology. But for this part of the experiment, each of the virtual ports were connected to a physical port, with a physical machine on the other side of that port. Each of the virtual ports created was disconnected from the IP stack, such that any communication to those ports needed to be routed through the bridge. The bridge also mitigated communication between the IP stack and the Ethernet port the host used for its own communication. Figure 3 shows the connection between virtual ports and the physical ports of our setup.

The Raspberry Pi 3 has 1 10/100 Ethernet port, and 4 USB 2.0 ports. The one Ethernet port was used to connect our “switch” to the router for access to the Internet through NAT, while four 10/100 USB-to-Ethernet adapters were connected to the USB ports. This expanded the switch to five ports, not

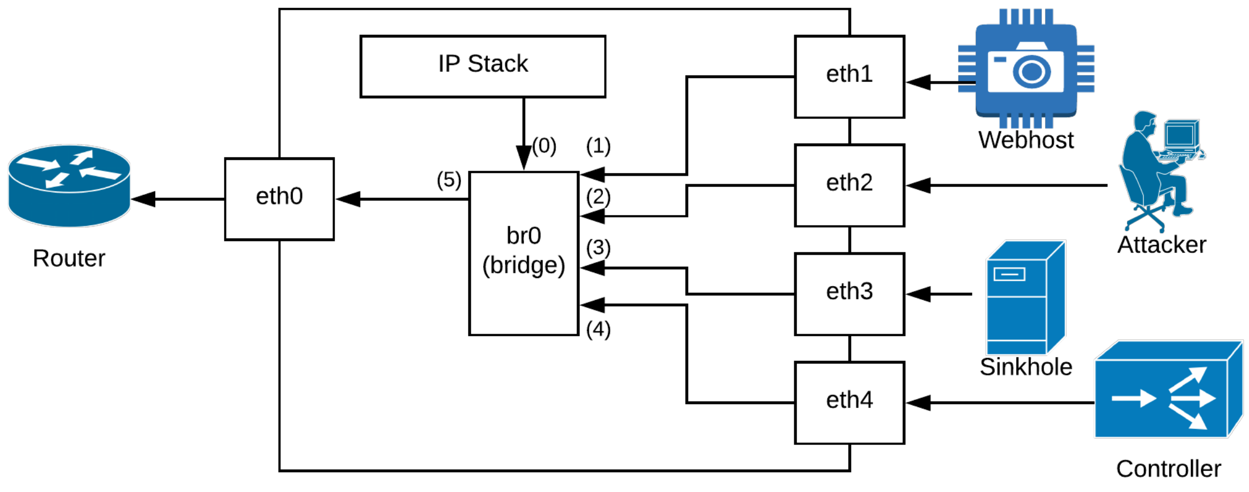

Fig. 3. Network interface configuration for OpenVSwitch on a Raspberry Pi 3

including the non-configured 2.4GHz wireless chip also on the board.

By default, OpenVSwitch runs a standard layer 2 switch configuration, allowing communication through all ports without the use of a controller. In the case of controller failure, OpenVSwitch can be configured to use this as a fallback, or use existing flows to attempt to continue to operate. After modifications to our controller from the virtualized testbed, we ended with three similar modules: a flow statistics query module, a module to modify flows for the known hosts on the subnet, and a layer 2 learning switch. Flow modifications in the hardware testbed were similar to the virtualized network, except that after processing a PacketIn message, the controller would not allow other modules to access a packet. This made it easier for our flow modifying module to process packets that were necessary for securing the webhost, and skipping the layer 2 switch modifications. In the event that a packet that the security module did not recognize, the layer 2 switch would process it and create the flow modification message.

Our webhost in hardware was a D-Link Internet-connected camera. While capable of being accessed through WiFi, the author felt it more feasible for this experiment to connect it via Ethernet to the Raspberry Pi Switch. The camera operates an HTTP server, hosted on port 80 by default. Username and password are required to view the camera stream or make any modifications. This made it a viable target for brute-force attack, a recognizable anomaly for our security system. We used the communication between the camera and a host while viewing the stream as the baseline for the anomaly detection.

The sinkhole was modified to prevent the recording of natural traffic moving through its interface. This was done to separate the traffic captured for the anomaly, and natural traffic such as ARP requests.

V. Testing and Discussion

To test the hardware implementation of our SDN, we had to create a better script than the one used in Mininet. Because Mininet runs locally, all communication between hosts happens near instantaneously, exaggerating the number of packets that travel between hosts per second. Initially, the author tested using Python libraries such as `scapy`, `requests`, and `httplib`, but no library would communicate fast enough to distinguish itself from normal traffic to the webhost.

Instead, the author utilized a series of tools included in the Kali Linux system, such as BurpSuite and THC-Hydra, to craft a directed attack on the system. Using BurpSuite, we can capture the HTTP request and response in the event of a failed login attempt. Using this information, we can configure the network logon cracker THC-Hydra to use a particular username and password list to attempt to log into the web page. Hydra could make a maximum of 64 connections simultaneously, which was not quite enough to trigger the anomaly detection. However, running 2 instances of the Hydra program at the same time was enough to trigger an anomaly, and the login attempts were forwarded to the sinkhole.

It is reasonable to assume that many attacks on the web server hosted on the camera would not be detected via our system. Other attacks not on the open port would be dropped before it even reaches the endpoint. However, with simple modifications to our anomaly detection module, we would be able to detect several other anomalous traffic patterns, such as many more connections being opened than normal (SYN flood).

While a writeup of the specific configurations made to the Raspberry Pi's should allow most people to set up an OpenFlow switch with this rate-limiting module, software-defined networking and network function virtualization is not currently in a place where the average Internet user would be able to secure their home in this way. Future work would include developing manageable user interfaces and modular applications to allow security-as-a-service (SaaS) in home networks.

VI. Conclusion

Innovations in open-source solutions to network management such as OpenFlow and OpenVSwitch continue to improve and allow new ways to provide secure and flexible networks in many environments. Security applications will continue to advance as machine learning techniques improve and are applied to both anomaly detection and software-defined networking.

In this paper, we discussed current research related to using SDN to secure networks. We described the protocol OpenFlow, and how it used to communicate messages between controllers and switches. We presented the exploration of software and hardware implementations of software-defined networks, and how they can be used to secure a network using rate-limiting anomaly detection. Our results show that while SDN security techniques on commodity hardware is financially feasible for the average user, its complexity may cause it to be inaccessible for many people.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: Enabling Innovation in Campus Networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, p. 69, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1355734.1355746

[2] M. A. Karaman, B. Gorkemli, S. Tatlicioglu, M. Komurcuoglu, and O. Karakaya, “Quality of service control and resource prioritization with Software Defined Networking,” *1st IEEE Conference on Network Softwarization: Software-Defined Infrastructures for Networks, Clouds, IoT and Services, NETSOFT 2015*, no. 318665, 2015.

[3] W. H. Muragaa, K. Seman, and M. F. Marhusin, “A POX Controller Module to Collect Web Traffic Statistics in SDN Environment,” vol. 10, no. 12, pp. 2048–2053, 2016.

[4] M. E. Ahmed and H. Kim, “DDoS Attack Mitigation in Internet of Things Using Software Defined Networking,” *2017 IEEE Third International Conference on Big Data Computing Service and Applications (BigDataService)*, pp. 271–276, 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7944950/

[5] Northbound Networks, “Zodiac FX,” 2017. [Online]. Available: https://northboundnetworks.com/products/zodiac-fx

[6] G. Velusamy, D. Gurkan, S. Narayan, and S. Baily, “Fault-tolerant OpenFlow-based software switch architecture with LINC switches for a reliable network data exchange,” *Proceedings - 2014 3rd GENI Research and Educational Experiment Workshop, GREE 2014*, pp. 43–48, 2014.

[7] H. Y. Pan and S. Y. Wang, “Optimizing the SDN control-plane performance of the Openvswitch software switch,” *Proceedings - IEEE Symposium on Computers and Communications*, vol. 2016-Febru, pp. 403–408, 2016.

[8] S. T. Ali, V. Sivaraman, A. Radford, and S. Jha, “A Survey of Securing Networks Using Software Defined Networking,” *IEEE Transactions on Reliability*, vol. 64, no. 3, pp. 1–12, 2013. [Online]. Available: http://www2.ee.unsw.edu.au/ vijay/pubs/jrnl/15tor.pdf

[9] S. Mehdi, J. Khalid, and S. Khayam, “Revisiting Traffic Anomaly Detection Using Software Defined Networking,” *Lecture Notes in Computer Science*, vol. 6961, pp. 161–180, 2011.

[10] R. Sathya and R. Thangarajan, “Efficient Anomaly Detection and Mitigation in Software Defined Networking Environment,” *INTERNATIONAL CONFERENCE ON ELECTRONICS AND COMMUNICATION SYSTEMS*, pp. 1679–1684, 2015.

[11] H. Kim, J. Kim, and Y. B. Ko, “Developing a cost-effective OpenFlow testbed for small-scale Software Defined Networking,” *International Conference on Advanced Communication Technology, ICACT*, pp. 758–761, 2014.

[12] M. Z. Asghar, M. A. Habib, and T. Hamalainen, “Performance Evaluation of OpenFlow Enabled Commodity and Raspberry Pi Wireless Routers,” vol. 10531, pp. 132–141, 2017. [Online]. Available: http://link.springer.com/10.1007/978-3-319-67380-6

[13] R. Austin, P. Bull, and S. Buffery, “A Raspberry Pi Based Scalable Software Defined Network Infrastructure for Disaster Relief Communication.”

[14] M. Boucadair and C. Jacquenet, “RFC 7149,” pp. 1–20, 2014.

[15] E. Haleplidis, J. H. Salim, D. Meyer, S. Denazis, and O. Koufopavlou, “RFC 7426,” pp. 1–35, 2015.

[16] B. Heller, “OpenFlow Switch Specification 1.0.0,” *Current*, vol. 0, pp. 1–36, 2009.