Byzantine Fault Tolerant Software-Defined Networking (SDN) Controllers

Karim ElDefrawy
Information and Systems Sciences Lab
HRL Laboratories
kmeldefrawy@hrl.com

Tyler Kaczmarek*
Bren School of Computer Science
University of California, Irvine
tkaczmar@uci.edu

Abstract—**A Software-defined Network (SDN) with a centralized controller suffers from a single point of compromise and failure which is detrimental to both security and reliability. Currently, the design space for robust and reliable distributed controllers remains largely unexplored except for some initial proposals incorporating simple majority voting. This paper develops, and assesses performance of, a prototype SDN controller that can tolerate *Byzantine faults* in both the control and data planes. The performance of our resilient controller implementation is measured against current standard fault vulnerable open source SDN controllers. We experiment with our prototype and show a reasonable slowdown as is expected in the transition from a fault vulnerable to fault tolerant design; our best controller exhibits only a 2x slowdown even though we have 4 replica components, and thus can tolerate a single compromised component without affecting control and/or forwarding decisions in the networks. Our controllers do not yet achieve high performance levels to be adopted in large-scale networks, e.g., to handle tens of thousands of new flows or flow modification requests per second, but we argue that (as a proof of concept) our controllers demonstrate feasibility of constructing such resilient programmable networks.**

I. Introduction

Software-Defined Networking (SDN) decouples forwarding operation (data plane) from control decisions (control plane). One of the main appeals of such separation is enabling a more expressive environment in which high-level routing and traffic engineering, and access control policies can be quickly and correctly implemented without requiring administrators to translate such policies into low-level configuration commands on a router or some other networking "black box." OpenFlow, has recently emerged as the generally accepted standard for SDN [14]. In OpenFlow, all forwarding in the data plane is handled by switches, and the control plane is represented by a (logically) centralized controller. Despite its flexibility, low cost and ease of management, the mainstream approach of SDN is to host all the "intelligence" of the network in a (logically) centralized controller node that programs "dumb" traffic switching devices. This allows for forwarding devices and switches to focus only on the actual forwarding and routing operations, rendering them into simple, programmable, and cost effective devices. However, this reliance on a (logically, and potentially physically) centralized controller, if not carefully designed and implemented, can result in a single point of failure for the entire network, and renders the network vulnerable to a new set of threats that are unique to SDN. The authors in [10] identified seven threats to an SDN network operating according to the paradigm above; chief amongst the threats is the damage that can be caused if the control plane is compromised, or if communication between the control plane and data plane is compromised or cut off. In the case of a compromise of the control plane, an adversary can issue switching and routing commands at will, and the switches would have no recourse but to forward packets along the faulty pathway(s), as the standard specification does not provide switches with a mechanism to identify faulty instructions if they are correctly authenticated, i.e., they originate from a compromised controller. Even if the centralized controller employs some form of authentication via digital signatures or message authentication codes, a compromise of the controller is likely to result in a compromise of the authentication secret keys, unless they are stored in a trusted platform module

*Work conducted while at HRL Laboratories.

or are secured by similar hardware mechanisms. In summary, the introduction of a single point of failure via the controller, as well as the implicit limits to scalability such a bottleneck brings with it, is one of the biggest criticisms highlighted by the opponents of SDN [16].

Contributions: The goal of this paper is to take a first stab at addressing the shortcomings of SDN controllers discussed above. To that end, we design and prototype *a Byzantine fault tolerant distributed SDN controller* that protects against the threats listed above; our controller can tolerate malicious faults in both control and data planes. We implement our controller design by integrating two existing open source Byzantine fault vulnerable SDN controllers with an existing Byzantine fault tolerant state machine replication software suite. We conduct experiments using open source SDN emulation software, and measure performance in a network of switches, and compare the relative performance of the fault tolerant controller with that of the fault vulnerable ones. Our experiments show a reasonable slowdown, around 2x in number of flows that could be handled per second[1], as is expected in the transition from a fault vulnerable to fault tolerant implementation of similar software.

Outline: The rest of this paper is organized as follows, Section II overviews some building blocks required in our design and implementation, Section III describes the design of our controller, Section V presents the results of the experiments that we performed, Section VI discusses related work, Section VII discusses open issues and potential future work, and Section VIII concludes the paper.

II. Building Blocks

A. OpenFlowJ and Beacon

Implemented in Java, Beacon [5] was designed with the intention of providing an alternative to the existing, Python-based, control platforms NOX [6] and POX [13] that did not sacrifice ease of programming for performance. The authors demonstrate that Beacon could maintain high performance in Cbench [18], the deFacto performance benchmark for OpenFlow controller throughput. Furthermore,

[1]In some cases the computer used for simulation runs out of memory due to scale of the simulation and we witness a larger unjustified slowdown.

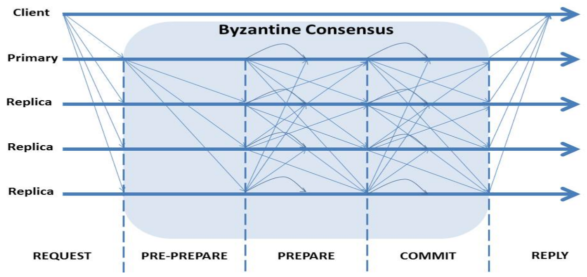

Fig. 1. How consensus is reached in BFT-SMaRT

the authors argue that the garbage collection within Java, as well as its cross-platform accessibility makes it an ideal candidate for the development of controller software [5]. OpenFlowJ [5] was developed shortly before Beacon as a basic implementation of an OpenFlow compliant controller platform in Java adhering to the OpenFlow 1.0 standard, and as such suffers in performance but is more readily usable for rapid development. We use Beacon and OpenFlowJ as the basis for our Byzantine Fault Tolerant (BFT) controllers, this choice is made primarily due to their comparability in design philosophy with BFT-SMaRt framework and tools discussed below.

B. BFT_SMaRt

BFT-SMaRt is a state of the art tool for creating Byzantine Fault Tolerant (BFT) systems implemented in Java. Developed by Bessani et al [2], BFT-SMaRt utilizes a protocol close to Lamport's original solution to the Byzantine Generals Problem [11], and achieves tolerance for f faulty replicas out of a network of $3f + 1$ total actors, with an arbitrary replica designated as the primary to facilitate execution.

A few assumptions are made about the physical network in the implementation of BFT-SMaRt, namely that the replicas are connected via a crossbar, and each client has a method for direct communication with each replica without relying on another replica as a go-between. This allows the network to emulate Lamport's case of ”Verbal Messages” as per the Byzantine General's Problem [11].

Consensus in BFT-SMaRt on a client's request is reached by executing the following protocol (illustrated in Figure 1):

1) REQUEST: the client issues a request via multicast to every replica
2) PRE-PREPARE: once the primary receives a client request it assigns a sequence number the primary and sends a PRE-PREPARE message to all other replicas
3) PREPARE: Once a replica i accepts a PRE-PREPARE message, it multicasts a PREPARE message to all other replicas, and adds both the PRE-PREPARE message and the PREPARE message to its log. Once a PREPARE message is sent out by replica i, it waits for a quorum of $2f$ PREPARE messages from the other replicas, those messages are used to create a quorum certificate for the message
4) COMMIT: One a quorum is collected by each replica i (including the primary) it multicasts a COMMIT message to all other replicas. This ensures that replicas agree on a total request order across view changes. A replica then collects message until it has a quorum of $2f + 1$ COMMIT messages, once it reaches a quorum, it adds the messages to a *committed certificate* and it is said that the request is committed by the replica
5) REPLY: Once a request is committed to by a replica, it executes the request and sends the response to the client, who waits for $f + 1$ identical requests before accepting it as the correct action

This process ensures that all requests are executed according to an exact total ordering (which preserves strong consistency throughout the system) and since the replicas are identical, deterministic state machines, the client is certain that the response must be correct once it receives the $f+1^{th}$ identical reply. Messages are not assumed to be received in order, so it is possible for a replica to commit to requests out of order. However, execution is in order, as a replica will keep its PRE-PREPARE, PREPARE and COMMIT messages logged until the corresponding request is up for execution.

In our implementation of this protocol, Message authenticity is ensured through the use of Message Authentication Codes (MACs) though support exists for the use of more costly public-key based digital signatures. The total number of one-way communication rounds to reach consensus is always 5, and assuming n controllers, there will be a total of $2n + 3n^2$ messages.

C. Adversary Model

We consider an adversary that actively corrupts and compromises up to f components of a distributed controller, as well as arbitrarily many switches and force them to behave in a Byzantine fashion, i.e., the adversary may modify their software and use them to launch any combination of attacks using injection, replay and modification of messages. Consistent with the threats specific to SDN as identified by [10], an adversary is able to cause a compromised controller to issue any arbitrary responses, including no response at all, to any incoming requests from a switch, and cause a compromised switch to issue arbitrary requests to the controller. The adversary's ability to cause a controller to issue no response is consistent with the results of a successful denial-of-service (DOS) on that controller, and will be considered as a special case of the adversary's ability to induce arbitrary behavior.

III. Controller Design

We design the Byzantine Fault Tolerant (BFT) SDN controller with the following goals in mind:

1) Minimize the deviation from and add-ons to a Byzantine fault-vulnerable controller
2) Avoid the introduction of an additional (possibly centralized) point of failure
3) Minimize the performance impact caused by the transition to a BFT controller
4) Enable a physically distributed controller

To this end, we chose to design and implement a BFT control suite that could be compatible as an extension to any SDN controller. We started with the development of an extension to the Java-based Beacon and OpenFlowJ controller suites. This was accomplished through the re-tooling of the communication traditionally between the controller and switch as local communication between the switch and a "Proxy" which took the $Packet\_In$ requests from its associated switch and reformatted it in a manner that the BFT server could digest, and similarly reformatted the replies from the BFT replicas into the $Packet\_Out$ and $Flow\_Mod$

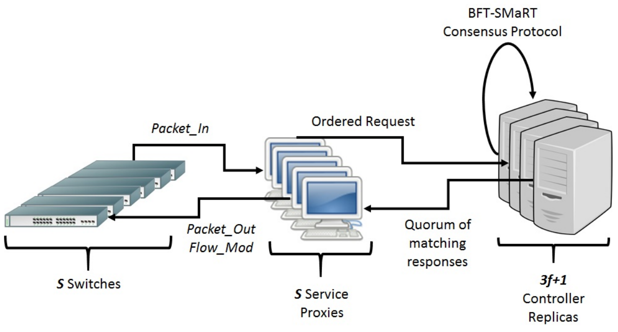

Fig. 2. Desired Message Flow

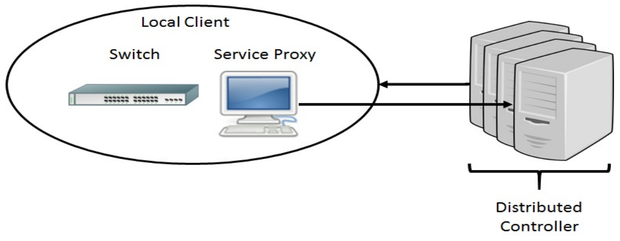

Fig. 3. Desired Architecture of a BFT Switch

messages that were expected by the switch as per OpenFlow's specifications. Figure 2 details the typical communication flow in our network. In this model, the controller regards the proxy and switch as the same entity, referred to as a client. Figure 3 illustrates the client-controller relationship.

We prototyped two controllers: $SimpleBFT$, the product of the integration of OpenFlowJ with BFT-SMaRt, and $BeaconBFT$, the product of the integration of Beacon with BFT-SMaRt. Both controllers are capable of handling f faulty replica out of a total of $3f + 1$ replicas, and any number of faulty clients.

In the tested mode, that of a simple learning switch, the controller replicas store a hashed table of tables of flow entries corresponding to the flow tables of the switches in the network. At the initialization of the network, these tables are empty, and the controller is unaware of any switch.

Similarly, each of the switches starts with an empty flow table, and is only aware of the hardware directly connected to it. At the initialization of the network, each switch is paired with exactly one BFT-Service Proxy, which will act as its representative to the controller for the duration of the network's existence.

When a packet is generated from a host to a switch that is headed to a destination that the switch does not have an entry in its flow table for, it will send a $Packet\_In$ message to the switch's BFT-Service Proxy, which will then act on the switch's behalf and format the message into a request that the controller can reach consensus on. The proxy will wait for $f + 1$ identical responses from the controller before formatting the response into a $Packet\_out$ or $Flow\_mod$ request that the switch can understand and act upon, thus updating its flow table.

The net result is that packets are handled in an identical manner in our BFT controllers as they are in the non-fault tolerant models, with the only exception that the control logic is being handled by replicas that must reach consensus on the appropriate action instead of a single controller that is allowed to act arbitrarily. As we will discuss in the following section, the communication requirements imposed by this consensus protocol does carry with it some performance overhead.

IV. Security Analysis

The integration of the BFT-SMaRt based replication with both OpenFlowJ and Beacon results in SDN controllers that are identical from a security standpoint. Considering a model in which an adversary can actively compromise up to f replica out of the total $3f + 1$ allows for the adversary to attempt the following: the adversary may attempt to provide unsolicited replies to arbitrary clients. This corresponds to the threat of a compromised controller as identified by [10]. In a typical, non-fault tolerant system, such unsolicited messages could potentially replace a portion or the entirety of any arbitrary switch's flow table with erroneous or malicious entries, which could be devastating for any host connected to that switch. However, in our implementation the controllers are forbidden from sending any unsolicited message to the client, and even if such a message were to arrive, the client would not act on it until it received $f+1$ matching messages from the replicas, a feat that the adversary could not preform without compromising more than f replicas. Additionally, the requirement for $f+1$ identical responses prevents a faulty message to be taken as true by the client in response to a valid request. Similarly, the adversary is restricted in compromising the replicas' ability to reach consensus in response to an incoming request, as for each of the steps in the consensus protocol, only a quorum of $2f$ replicas is needed for any message to be added to the transcript, and thus acted upon. In this model, the incorrect action (or inaction) of a faulty replica cannot cause the consensus to be delayed by any time greater than the difference in the time it takes for the compromised replica and the slowest replica to complete a round of communication.

V. Experiments

We evaluate the performance of our implementation relative to the fault vulnerable controllers that our BFT controllers build on. We only consider a purely reactive controller logic when testing the average throughput in establishing new entries in the flow table from the controllers to the switches, i.e., each switch starts with an empty flow table, and each new flow request is passed directly to the controller, which will then install a single entry on the flow table corresponding to that request.

Our tests were conducted in a network simulated in Mininet [12], a network simulation framework for small-scale prototyping. The simulated network consists of 64 hosts and 63 switches in a binary tree layout, meaning that in the worst case a single request produces 11 new flows across 11 switches. While Mininet does not promise highly optimized performance, the relative performance between the various controllers is accurate. Table I shows the measurements we collected for evaluating the performance of our controllers. The metrics we measure are based on recommended SDN benchmarking metrics [4]. The "End-to-End Flow Setup Duration" is the average time required to establish a full path from one host to another, the "Flow Setup Delay" is the time it took for a single entry in the flow table of a single switch to be established, and the "Flow Setup Rate" is the average number of flow table entries established each second. SimpleBFT and BeaconBFT were both tested in an environment in which they could handle a single fault in the control plane, and thus had 4 controller replicas.

TABLE I
Flow Setup Measurements (fm/sec = flow modifications per second). SimpleBFT and BeaconBFT are our Byzantine fault tolerant versions of OpenFlowJ and Beacon.

Controller	End-to-End Setup Duration	Flow Setup Delay	Flow Setup Rate
OpenFlowJ [1]	376ms	9.44ms	106.9 fm/sec
SimpleBFT	775ms	31.7ms	59.3 fm/sec
Beacon [5]	77ms	0.5ms	550.6 fm/sec
BeaconBFT	475ms	14.5ms	87.0fm/sec

The results in Table I show that the performance of the fault tolerant adaptation of Beacon suffered the most from the extra costs of Byzantine Fault tolerance, though it was still able to maintain a higher throughput than the controller constructed by extending OpenFlowJ. These drops in performance highlight the increased cost in communication complexity required to reach consensus on each $Packet\_In$ message received by the controller.

However, there appears to be an asymmetrical impact on performance between SimpleBFT, which experiences a slowdown close to 2x when compared to OpenFlowJ, the insecure controller on which it was based, and BeaconBFT, which experiences a slowdown of around 6x when compared to Beacon, the insecure controller on which it was based. This result is not entirely unexpected, as OpenFlowJ processes messages from switches serially, while Beacon leverages parallelism. However, as BFT-SMaRt relies on establishing a total ordering of incoming messages, the potential benefit gained through the parallel handling of requests is greatly reduced, and thus BeaconBFT experiences a larger slowdown than SimpleBFT.

This impact on performance in both cases could be mitigated through several methods, namely the development of a client that sends requests at a coarser granularity than a single $Packet\_In$, the development a controller suite that can more effectively bundle together responses, or the development of a BFT consensus protocol that operates either on a coarser granularity or can operate with a less rigorous consensus protocol in times of faultless execution.

VI. Related Work

Reliable and Resilient SDN Controllers: CORONET is an SDN controller that is mainly concerned with faults in the data plane, that is, faults that occur when a switch or link fails [8]. The authors in [8] note that existing fault recovery schemes from legacy networks will not work in SDN, as small-scale local fault recovery is not possible without some control logic on the switch, which defeats the goal of separating the control and data planes, they also note that the substantial cost in reestablishing every flow on a freshly-reset switch is not an efficient solution. FatTire [17] seeks to provide a programming language for SDN controllers that is inherently fault tolerant for faults in the data-link plane. The language's central feature is a construct based on regular expressions. This construct allows specification of a set of valid paths for a packet to take through the network as well as the degree of fault tolerance required. This is implemented by a compiler that focuses on in-network fast-failover mechanisms as specified by OpenFlow 1.3, with a focus on making the language expressive and correct. Control-plane faults seem not to be detailed as a priority in FatTire at the point of this writing.

The authors in [3] argue for the necessity of fault tolerance in the controller layer and put forth a design in which many physically diverse controllers communicate with a fault tolerant "Data Store (DS)" that lies above the control plane. This approach, while effective in tolerating faults in the control plane, introduces a new bottleneck at the DS level, as the primary factor in determining the speed of the controller in handling requests now relies on the amount of time it takes for the DS to communicate with the controller, reach consensus, and then reply to the controller, before the controller can reply to the switch. Finally, the DS is not distributed, which again introduces a single point of failure.

Our approach differs from the above in several aspects. First, unlike FatTire or CORONET, which are primarily concerned with faults in the data plane, we are concerned with arbitrary Byzantine faults in both the data plane and the control plane. Additionally, we seek to realize such resilience without introducing a new layer of communication or additional entities that switches need to communicate with, as in [3]. Finally, we aim for a design that extends and builds up on existing controller suites, and makes them compliant with our Byzantine fault tolerant model, instead of developing entirely new paradigms for constructing SDN controllers, as FatTire and CORONET do.

VII. Future Challenges and Research Directions

There are several directions to extend our work. First, the development and use of a consensus protocol tailored for the tasks of an SDN controller would undoubtedly yield a communication complexity that has less of an impact on controller performance and will lead to a controller that is more capable of handling realistically large workloads. This could either be accomplished through adaptation of a protocol similar to Speculative BFT [9], which focuses on minimizing the communication necessary in period of faultless execution by requiring only $f + 1$ replicas participate in the protocol until a fault is encountered, or the optimization and tailoring of a more traditional, Lamport-like consensus protocol, like BFT-SMaRt through appropriate batching and refinement on the appropriate granularity on which messages requiring consensus should be received.

Secondly, the development of an OpenFlow-compatible switch that is integrated with the client in the BFT protocol, instead of using an additional proxy would remove an unnecessary step in the communication and therefore provide a boost in performance. A few subtle tweaks to the standard Open Virtual Switch [15] would yield a compatible software implementation, but hardware integration presents a significant roadblock moving forward.

As a tangent to these directions, it would also serve to highlight the effectiveness of a controller such as ours to examine it in the proactive case, where installing hundreds of thousands of new flows ever second may not be a strict requirement. If instead the majority of the flow table could be constructed without a great deal of concern on the timeliness of the construction, and new flows were not expected to be a exceedingly common occurrence. For instance, Kanizo et. al's proactive controller, Pallette [7] seeks to create flow tables for every switch in bulk through preprocessing in such a manner that the average flow table is as small as possible. Since the distribution of rules onto flow tables occurs in a preprocessing stage before the network comes online, and does not rely on a massive flow of requests from the switches to the controller, a controller such as ours could provide the desirable qualities implicit in its fault-tolerant design without as large as a performance degradation as was seen in the purely reactive case.

VIII. Conclusion

Efficient SDN controllers that can withstand malicious faults at both control and data planes are an important building block to ensure resilience and security in such programmable networks. Our Byzantine fault tolerant controllers currently experience a 2x slowdown compared to the insecure cases; note that we have 4 controller replicas, so 2x as opposed to 4x is already a promising result. Our controllers do not yet achieve practical performance levels to be adopted in large-scale networks, e.g., to handle tens of thousands of new flows or flow modification requests per second, but we argue that as a proof of concept, our controllers demonstrate feasibility of constructing such resilient programmable networks. Furthermore, with ongoing developments in SDN controllers that can operate outside of a strictly reactionary mode, the performance degradation of a fault tolerant controller such as ours may already provide a reasonable performance level that is capable of handling realistic network loads, improving the performance in the non-reactionary setting to handle tens of thousands of requests per second is a promising avenue for future work.

References

[1] Openflowj controller, 2015.

[2] Alysson Bessani, Joao Sousa, and Eduardo Alchieri. State machine replication for the masses with bft-smart. *DI/FCUL, Tech. Rep*, 2013.

[3] Fabio Andrade Botelho, Fernando Manuel Valente Ramos, Diego Kreutz, and Alysson Neves Bessani. On the feasibility of a consistent and fault-tolerant data store for SDNs. In *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, page 3843. IEEE, 2013.

[4] Gery Czirjak and Ramdas Machat. Benchmarking methodology wg sarah banks internet draft akamai intended status: Informational fernando calabria expires: October 8, 2014 cisco. 2014.

[5] David Erickson. The Beacon OpenFlow Controller. In *HotSDN*. ACM, 2013.

[6] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[7] Yossi Kanizo, David Hay, and Isaac Keslassy. Palette: Distributing tables in software-defined networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 545–549. IEEE, 2013.

[8] Hyojoon Kim, J.R. Santos, Y. Turner, M. Schlansker, J. Tourrilhes, and N. Feamster. CORONET: Fault tolerance for software defined networks. In *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, pages 1–2, October 2012.

[9] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, page 4558. ACM, 2007.

[10] Diego Kreutz, Fernando Ramos, and Paulo Verissimo. Towards secure and dependable software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 55–60. ACM, 2013.

[11] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382401, 1982.

[12] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[13] Murphy McCauley. About pox, October 2014.

[14] B Nunes, Marc Mendonca, X Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. 2014.

[15] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In *Hotnets*, 2009.

[16] Gergely Pongrácz, Laszlo Molnar, and Zoltán Lajos Kis. Removing roadblocks from sdn: Openflow software switch performance on intel dpdk. In *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, pages 62–67. IEEE, 2013.

[17] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. FatTire: declarative fault tolerance for software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, page 109114. ACM, 2013.

[18] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W Moore. Oflops: An open framework for openflow switch evaluation. In *Passive and Active Measurement*, pages 85–95. Springer, 2012.