POLITECNICO DI TORINO
Repository ISTITUZIONALE

Dealing with misbehaving controllers in SDN networks

Original
Dealing with misbehaving controllers in SDN networks / Zhang, T., Bianco, A., Giaccone, P., Nezhad Payandehdari, A.. - ELETTRONICO. - (2017). (IEEE Globecom 2017 Singapore December 2017) [10.1109/GLOCOM.2017.8254752].

Availability:
This version is available at: 11583/2678306 since: 2019-01-08T19:01:54Z

Publisher:
IEEE

Published
DOI:10.1109/GLOCOM.2017.8254752

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2017 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

14 October 2026

Dealing with misbehaving controllers in SDN networks

Tianzhu Zhang, Andrea Bianco, Paolo Giaccone, Aliakbar Payandehdari Nezhad
Dept. Electronics and Telecommunications, Politecnico di Torino, Italy

***Abstract*—The logical centralized approach in the control of SDN networks allows an unprecedented level of programmability in the network, but also implies the vulnerability in the case of misbehavior of the controller, due for example to software bugs, hardware problems or hacker attacks. In our work we propose to exploit the diversity offered by multiple controllers to manage the network switches and detect misbehaviors whenever one controller issues different OpenFlow instructions for the data plane with respect to the others. We design a behavioral checker, denoted as BeCheck, that acts as a transparent relay in the interaction between the network switches and the controllers. We propose and investigate different policies to relay the messages and to detect the controller misbehavior. We implement and validate our approach in a simple testbed, showing the possible tradeoff between detection reliability and controller reactivity perceived at the switches.**

I. Introduction

Unlike traditional IP networks, software-defined networking (SDN) decouples the network control logic and the forwarding functions. SDN controllers act as the "brain" of the network by making routing decisions and configuring the underlying simple forwarding devices in a logically centralized fashion. This refinement greatly simplifies network configuration and programmability.

However, just like any complex systems, SDN controllers are susceptible to misbehaviors, exacerbated by the centralized approach. Software bugs (e.g., persistent loops, synchronization failure, inconsistent state, response omission etc.) have been discovered in many popular SDN controllers. According to [1], the load balancer of Floodlight [2] may fail to distribute flows consistently and the POX [3] forwarding modules can delete rules installed by other modules. Furthermore, experiments in [4] detected totally 11 bugs on merely 3 applications of NOX [5]. Even the most practically relevant open source SDN controllers, namely OpenDaylight (ODL) [6] and ONOS [7], are not immune to bugs. According to [8], ODL can face flow deletion and instantiation failure bugs while link detection inconsistency and flow rules pending bugs are spotted in ONOS. These bugs can give high risk to misbehaving controllers. To make things worse, various security attacks to the control plane increases the possibility of misbehaving controllers. According to [9], SDN controllers including POX, Maestro [10], OpenDaylight and Floodlight are all susceptible to a diversity of security attacks on network topology and data forwarding. In addition, malicious network administrators can also misconfigure SDN controllers to sabotage the network [11]. Most of the previous works improve the reliability and security of SDN control plane either by verifying the controllers' behaviors through complicated analysis such as model checking, or by advocating secure and dependable design of SDN controllers. Few of them provides solutions for real-time detection of misbehaving controllers.

In this paper, we present a behavioral checker, denoted as *BeCheck*, which is a module that relay the OpenFlow (OF) traffic between the controllers and the network switches, and detects misbehaving controllers in real-time, by comparing the instructions received by the controllers, which operate in locksteps. BeCheck resides *transparently* between the SDN controllers and the data plane, and neither the controllers nor the forwarding devices need to be modified for its presence. We propose a misbehavior detection policy, combined with different forwarding policies, and investigate experimentally the possible tradeoffs between the detection reliability and the reactivity of the application as perceived by the network switches.

The paper is organized as follows. Sec. II discusses the previous work. Sec. III introduces the architecture of the proposed solution and describes different detection policies. Sec. IV describes the implementation of the prototype and the testbed adopted for the experimental validation. The experimental results highlight the different tradeoffs between detection reliability and the controller reactivity as perceived by the network switches. Finally, in Sec. V we draw our conclusions.

II. Related works

Many recent works [1], [4], [12]–[15] focused on checking software bugs and verifying the correctness of SDN control plane. In particular, [12] proposed a system that deductively verifies if an SDN program is correct on all feasible network topologies. Similarly, [1] implemented a dynamic analyzer to identify software bugs and prevent a network from concurrent violations. The work [14] developed a distributed model checker to verify some security properties related to the network state. In [15] a troubleshooting technique was implemented that can automatically detect the minimal sequence of random inputs responsible for bugs in the SDN control plane. The work in [4] presented a model checking technique with symbolic execution to test the applications running on top of SDN controllers. On the other hand, the studies [11], [16] focused on the security aspects of SDN. In particular, [16] analyzed the threats related to the SDN paradigm and advocated the design of secure and dependable SDN controllers. Finally, [11] presented a first prototype of secure SDN controller designed to deal with malicious SDN administrators. However, all the above works adopted complex analysis processes, such as model checking, which are very time-consuming and difficult to run in real-time. Unlike them,

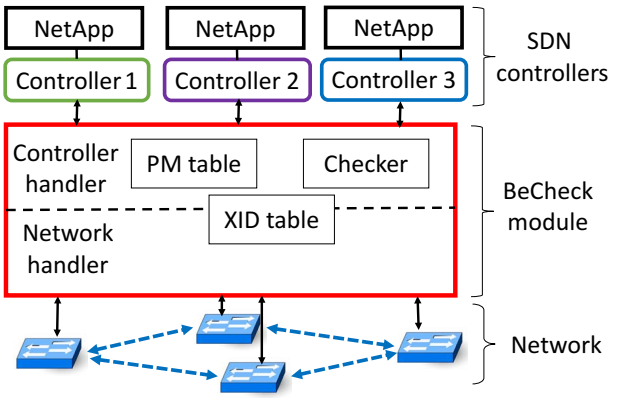

Fig. 1: The architecture of BeCheck with $C = 3$ controllers

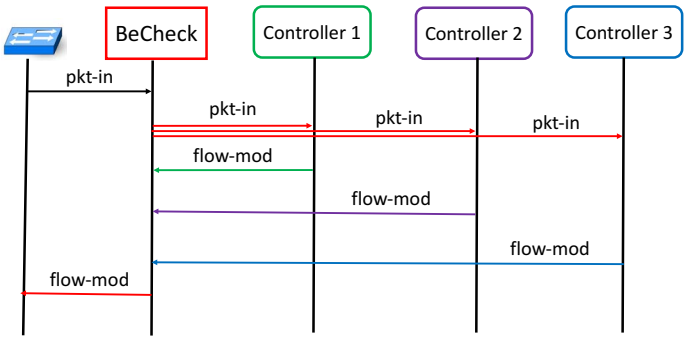

Fig. 2: Example of BeCheck behavior with $C = 3$ controllers

our work detects misbehaving controllers in real-time by comparing the behavior of independent controllers, in a transparent way for both the controllers and the network switches.

More similar to our work, [13] proposed to check some network-wide invariants in real time (e.g. loop free routing) by deploying a software layer between the control and data plane and dynamically inspecting each flow installation rules. It was based on a single controller, whereas BeCheck operates with multiple controllers. The advantage of our approach is that BeCheck runs completely oblivious of the network application. Finally, [8] aims at validating the behaviors of distributed SDN controllers. Similar to us, a consensus process is employed to determine the correct actions and detect misbehaving controllers. However, the detection is based on replicating the network events on the other controllers within the cluster, thus requires some modification of the internal management of the cluster. Instead, BeCheck is completely transparent with respect to the controllers, which do not require any modification.

III. Architecture of our behavioral checker

Fig. 1 shows the general architecture of our behavioral checker, which exploits the diversity offered by multiple controllers running in lockstep the same network application. Let C be the number of controllers, with $C \geq 2$. All C controllers operate on the same network, and each controller is responsible for managing all the switches, i.e. acting as master for OpenFlow (OF) switches. The controllers run the same network application; thus, if they all behave correctly, they will show the same sequence of messages sent to the switches. Thus, misbehaviors can be detected by comparing the messages received by the controllers and check if inconsistent messages are sent to the switches. Our behavioral checking module, denoted as *BeCheck*, is responsible to digest the OF instructions arriving from the controllers, check their consistency and replicate a copy of the instructions to the destined switch. At the same time, BeCheck replicates all the network events to all the controllers, to ensure that the network applications proceed in lockstep with coherent states. BeCheck is connected to the switches as it was their master controller, but actually its role is just to detect misbehaviors and relay the messages in both directions between switches and controllers. Thus, BeCheck does not substitute the controller and does not take any decision for the data plane, being completely oblivious of the specific network application running on the controllers.

It is worth to highlight that our approach is different from a cluster of distributed controllers, where each controller acts as master only for a subset of switches. Distributed controllers indeed are aimed at improving reliability and scalability of the SDN control plane. Instead, in our case we exploit the diversity provided by multiple controllers, in order to detect misbehaviors. It would be also possible to extend our approach to cluster of controllers, i.e. each controller in Fig. 1 might be a different cluster of distributed controllers, but we have left this extension for future work.

BeCheck acts as a relay for the OF messages between the controllers and the switches. Not all the OF messages are actually processed by BeCheck, since some of them are not meaningful to detect misbehaviors in the data plane. In OF standard [17], three kinds of messages are defined: (i) controller-to-switch messages, which are initiated by the controller and allow to manage the switches; (ii) asynchronous messages, which are initiated by the switch and allow to update the controller about some local events (e.g. changes in the switch state); (iii) symmetric messages, which are initiated by either the switch or the controller and sent without solicitation. In our case, we consider just the main OF messages dictating the forwarding behavior of the data plane in the network, i.e. `pkt-in` (which is an asynchronous message), `flow-mod` (which is a controller-to-switch message) and `pkt-out` (which is a controller-to-switch message). All the other kinds of messages are instead transparently forwarded by BeCheck.

As example of BeCheck behavior, consider the scenario of a reactive forwarding network application. Assume now that a switch receives a packet from the network, generates a `pkt-in` message and its master controller reacts by sending to the switch a `flow-mod` message. As shown in Fig. 2, the switch sends the `pkt-in` to BeCheck, which replicates it to all the three controllers. Now each controller reacts to the `pkt-in` independently, and sends a `flow-mod` message to BeCheck. If now BeCheck finds out that all the 3 messages received by the controllers are coherent, then it sends to the switch one copy of the `flow-mod`. Otherwise, in the case of inconsistency, i.e. one controller behaving incorrectly, BeCheck can react in different ways: e.g., generates an alarm towards the controllers, or disables the interaction with the

OF message type	Datapath-id	Switch xid	Controllers xid
`pkt-out`	...:0A	1001	{101, 201, 301}
`flow-mod`	...:FB	1002	{102, 202, 302}

TABLE I: Example of XID table for 3 controllers

OF message type	Datapath-id	Action	Controller bitmask B
`pkt-out`	...:0A	output 1	100
`flow-mod`	...:FB	output 3	110

TABLE II: Example of PM table for 3 controllers

misbehaving controller. The details on how to manage misbehaving controllers are outside the scope of the current paper.

There are three critical challenges about implementing BeCheck. First, to guarantee the communication between BeCheck and all the controllers, the transaction id ("xid") for each single controller must be managed correctly. Since BeCheck module interacts with multiple controllers which may react to a network event using different xids, OF reply messages with wrong xids from the network can result in connection refusal by the controllers. BeCheck implements a scheme to map the xids used to interact with the controller and the xids to interact with the switches. Second, to detect misbehaving controllers, the action/s associated with each OF controller-to-switch message (i.e. `pkt-out`, `flow-mod`) must be recorded and the messages are checked in terms of coherence between the different controllers. BeCheck records the messages from the controllers throughout a Pending Message (PM) table. Third, the policy according to which the behavior check is preformed as well as the timing to send the required OF messages to the switches have different impacts on the detection reliability and on the reactivity of BeCheck. BeCheck implements one of the three detection policies described in Sec. III-C.

The software architecture of BeCheck, as shown in Fig 1, mainly consists of two components: *controller handler* and *network handler*. The controller handler manages the message exchange with the controllers whereas the network handler manages the message exchange with the network switches. The *XID table* is shared between the two components and it implements a one-to-many table mapping for the xids used for the network switches and for the controllers. The *PM table* buffers the instructions of the OF messages received from the controllers and it is updated by the controller handler. The Checker submodule is in charge of running the policy to verify the behavioral consistency among the messages received by the controllers and to detect possible misbehaviors.

A. Network handler

The network handler plays the role of forwarding OF messages between the controller handler and the network. Whenever an OF message is received from the network, the network handler adds the corresponding xid field in the XID table and replicates the message to all the controllers, storing the corresponding values of xids used for the interaction with the controllers. The XID mapping is required since multiple controllers may send OF messages with different xids to react the same event, and OF reply messages with unexpected xids are discarded by the controllers.

The XID table consists of 4 fields, mainly based on the 8-byte header of OF. The first two fields are aimed at identifying the message, while the last two store the xids mapping. The message is identified through the OF 1-byte "message type" and the corresponding switch is identified through the 8 byte "datapath-id" (based on the switch MAC address). Finally, the xid is detected by the 4-byte xid present in the OF header of the messages exchanged with the network and with the controllers. An example of XID table is shown in Table I.

B. Controller handler and detection policy

The controller handler is the central part of BeCheck. It maintains the connections with the SDN controllers and forwards OF messages between the controllers and the network handler. It also collects and inspects the messages from the controllers' side, so as to detect possible misbehaviors. All the OF messages, except `flow-mod` and `pkt-out`, are sent directly to the switches in a seamless fashion.

If all the controllers behave identically, BeCheck expects exactly the same OF message received from all the three controllers in some random order. A message is considered "pending" if it has been only confirmed by a subset of controllers. The handler operates on the Pending Message (PM) table, based on the OF messages (either `pkt-out` or `flow-mod`) received from the controller.

The *detection policy* is based on the full consensus on the messages received by the controllers and works as follows. Only when a message is confirmed by all the controllers, then it is considered correct and removed from the PM table. Otherwise, after a fixed timeout, an entry is removed and a misbehavior event is generated, since the corresponding command has not been confirmed by all the controllers.

The PM table contains a message identifier identical to the one in XID table, based on the message type and the datapath-id. Now the particular action associated to such message is stored in the table and a controller bitmask $B = [b_i]_{i=1}^{C}$ is updated to keep track of the controllers that sent the message; the ith bit is defined as:

$$b_i = \begin{cases} 1 & \text{if the message was received from controller } i \\ 0 & \text{else} \end{cases}$$

By construction, $\sum_{i=1}^{C} b_i$ gives the number of controllers from which the message from received. Whenever $\sum_{i=1}^{C} b_i = C$ (i.e. the same message has been received from all the C controller), then the corresponding entry is removed from the PM table.

A sample PM is shown in Table II, which refers to the two messages of Table I. According to it, the `pkt-out` message has been already confirmed by controller 1, whereas the `flow-mod` message by controllers 1 and 2.

C. Forwarding policy

The Checker submodule in Fig. 1 reacts on changes in the PM table and triggers two kinds of events: (i) the transmission to the switch of an OF message, (ii) the misbehavior detection. We define reaction time as the latency experienced by the controllers to react to a new event in the network (e.g. `pkt-in`

due to a new flow) as perceived by the network switches. E.g., in the toy example of a reactive forwarding application, the reaction time is the interval of time between the generation of the `pkt-in` for a new flow and the reception of the `flow-mod/pkt-out` at the switch, generated by BeCheck. We propose three different forwarding policies, each of them with a distinct trade-off between detection reliability and reactivity:

- **Consensus** policy (CO) sends the OF message to the switch only when the message has been received by all the controllers, i.e. only when $\sum_{i=1}^{C} b_i = C$. This approach introduces the largest reaction time, since it depends on the slowest controller, but it is the most reliable policy since it is able to detect immediately misbehavior of just 1 controller and sends to the switches only fully correct messages. In the case of a misbehavior, BeCheck will not relay the messages from the controllers to the switches, since they are unconfirmed by all the controllers, and this results into a network outage.
- **First Response** policy (FR) sends the OF message to the switch just after the first message has been received by any controller, i.e. as soon as $\sum_{i=1}^{C} b_i = 1$. In the case of a misbehavior, BeCheck will keep relay the messages until the timeout of the corresponding entries in the PM table expires and the detection occurs. This approach introduces the smallest reaction time, due to the fastest controller, but it is the least reliable policy since incorrect messages may be sent to the switches (e.g. when the fastest controller is misbehaving) and the misbehavior detection takes longer.
- **Majority** policy (MA) sends the OF message to the switch just when the message has been received by the majority of the controllers, i.e. when $\sum_{i=1}^{C} b_i > C/2$. The behavior of this approach is intermediate in terms of reaction time and reliability with respect to CO and FR policies.

The performance of these three policies are evaluated in the following section.

IV. Validation and Experimental Evaluation

We evaluate the performance of BeCheck and compare the different forwarding policies running in the Checker submodule. For easy reference, we introduce the following notation to identify the variants of our approach: "BeCheck-X(C)" where X is either CO, FS or MA depending on the forwarding policy, and C is the number of controllers.

We implement BeCheck module using OpenFlowJ [18] and NIO.2 libraries [19]; the final source code is around 1500 lines. The experiments are performed in the scenario shown in Fig. 3, based on C controllers and N switches in the network. We run BeCheck instance directly in a server, together with the controller instances and the network, emulated with Mininet [20]. The switches communicates with BeCheck through port 6633, thus BeCheck appears as a classic master controller. BeCheck, in turn, is connected to each controller through their predefined port 6633.

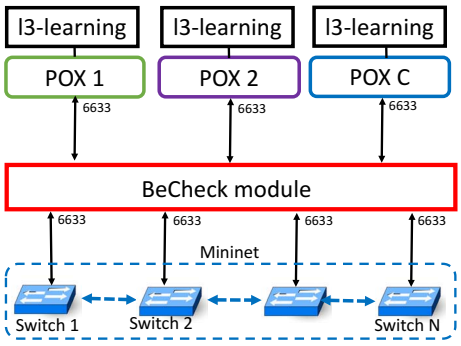

Fig. 3: Experimental setup

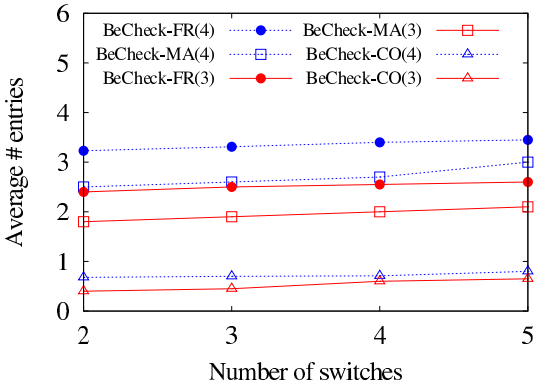

Fig. 4: Average size of the PM table

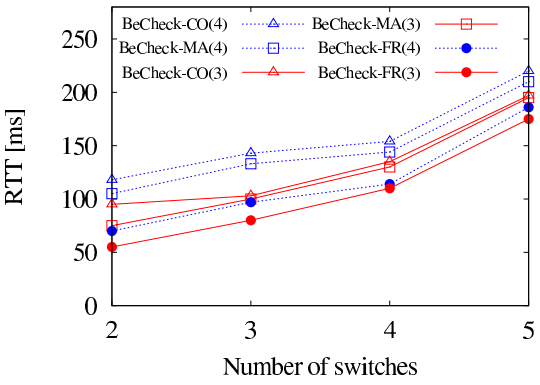

Fig. 5: Response time obtained with ping command

We evaluate the time-average occupancy of the size of the PM table, since its small size is crucial for the scalability of the proposed approach, in particular due to misbehaving controllers. We do not report the size of the XID table since its occupancy (never larger than PM table) is not affected by possible misbehaving controllers.

We perform our tests with C independent instances of POX controller [3], each of them running the default `l3-learning` application for reactive forwarding. We show the results for $C \in \{1, 2, 3, 4\}$. Note that for $C = 1$ the forwarding policy does not have any effect, since acts just as a message relay, and this case is used as term of comparison for the evaluation of the computation overhead of the bare

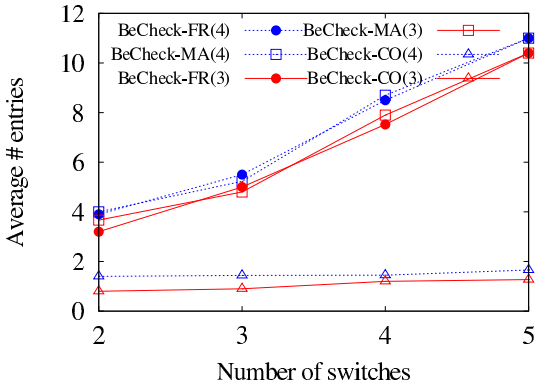

Fig. 6: Average size of the PM table in the case of one misbehaving controller

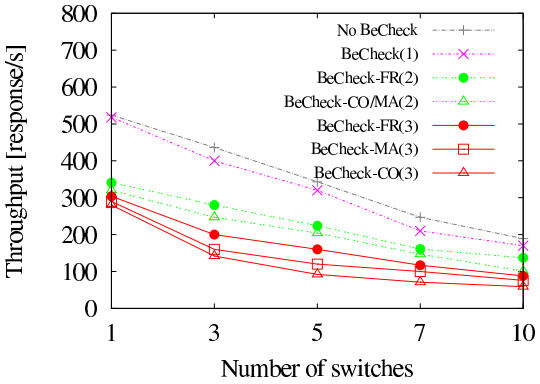

Fig. 7: Performance evaluated with Cbench tool

BeCheck application (e.g. due to the socket management for the transmission and reception of OF messages). In all our experiments, the network topology is linear, connecting N switches, with $N \in \{1, \ldots, 10\}$.

We also evaluate the reactivity perceived by the switches by running the `ping -c 1` command at the terminal connected at the first switch, generating one single ICMP packet towards the Nth switch, and recording the corresponding Round Trip Time (RTT) for the first ICMP request/reply exchange in the switch.

Fig. 4 shows the table size for $C \in \{3, 4\}$ and $N \in \{2, 3, 4, 5\}$, for different forwarding policies. Consensus (CO) policy always presents the minimal average number of entries, whereas First Response (FR) policy shows the largest number of entries. The reason can be easily understood assuming that one controller is much slower than the others to process the `pkt-in` messages received by the network, relayed by BeCheck. Indeed, for CO only when all the OF `flow-mod` messages are received by BeCheck for a target switch, then the message is sent to the switch, which forwards the ICMP packet one hop further along the path. Thanks to the deletion policy in PM table, the corresponding entry is deleted and thus the maximum number of entries is one independently from the number of controllers. On the contrary, FR policy sends the `flow-mod` to the destined switch as soon as it receives the message from the fastest controller, without waiting for the slowest controller. Thus each switch can forward the ICMP packet to the next switch in the path, without waiting for the `flow-mod` generated by the slow controller. But this implies that a large number of entries will be present in the table, which will be deleted only when the slowest controller sends the `flow-mod` message. Majority (MA) policy instead, by construction, behaves in an intermediate way with respect to CO and FS.

Our intuitive explanation is corroborated by observing the reaction time perceived at the network switches, as shown in Fig. 5. FR achieves always the best reactivity (i.e. the smallest RTT), CO the worst, whereas MA is in the middle of the two. The graph shows also that the RTT increases with respect to the network size, due to the larger number of hops in the network.

We now consider the scenario in which a controller is misbehaving. We modify the `l3-learning` application in one controller to send always the ICMP packet towards an unused port of the switch. Fig. 6 shows the occupancy of the PM table. The number of entries for CO is now bounded by two, independently from the number of switches, since CO receives a coherent OF message from N-1 controllers and another one from the misbehaving controller. CO detects at once the inconsistency. The other two policies are also able to detect the misbehavior but, differently from CO, allow the network application run for the non-misbehaving controllers. The main difference between FR and MA is that MA assures always a correct behavior (since the majority of the controllers are behaving correctly in our scenario), whereas FR generates wrong instructions on the data plane whenever the misbehaving controller is the fastest. In all these cases, the occupancy of PM tables grows proportionally to $2N$, since around 2 pending messages are stored for each switch.

To evaluate the actual overhead due to BeCheck, we keep the same configuration as in Fig. 3 with all the controller behaving correctly, but we use Cbench to emulate the network with N switches instead of Mininet. Cbench is not able to emulate a real network topology, but just a set of N switches flooding the controller with `pkt-in` messages. We run now `l2-learning` application on POX controller. We record the "throughput" of BeCheck in terms of maximum number of responses per second, as measured by Cbench. The results are obtained by averaging the results for 100 tests, each of them lasting for 1 sec. The results of the tests are shown in Fig. 7. As expected, the throughput decreases with increasing number of switches, since in Cbench the switches generate `pkt-in` messages simultaneously. As term of comparison, we report also the scenario, denoted as "No BeCheck", in which BeCheck is not present between the network and the single controller. Thus, from Fig. 7 it is clear that the reduction of throughput for larger networks is due to the POX controller. Furthermore, BeCheck(1) (i.e. with just one controller) shows a minimum degradation in terms of performance, due to the basic operation of relaying the messages between the controller and the switches. Instead, when the forwarding policies are effective, the performance degrades almost proportionally with respect to the number of controllers. This result is quite expected since each POX controller is able to answer to the flooding requests of Cbench without almost any processing, and thus the main bottleneck becomes the BeCheck module, on which the processing of multiple controllers converges.

V. Conclusions

We propose a behavioral checker, denoted as BeCheck, to detect in real-time possible misbehaviors of SDN controllers. We assume that multiple, independent controllers are running the same network application, and thus their behavior must be coherent. BeCheck is based on a module which relay the OpenFlow traffic between the controllers and the network switches, and compares the OpenFlow instructions to detect possible misbehaviors. We propose a detection policy based on the full consensus on the messages from the controllers. Combined with the detection policy, we consider the effect of different forwarding policies, First Response (FR), Majority (MA) and Consensus (CO), which offer different tradeoffs between the detection reliability and the latency introduced by the BeCheck module. BeCheck runs completely transparent from the point of view of the network switches and the SDN controllers, and operates obviously from the specific network application running on the SDN controllers.

We implement BeCheck as a standalone module and investigated its performance for the specific case of a basic reactive forwarding application running in multiple instances of POX controller. We show the possible tradeoff between the detection reliability and the controller reactivity as perceived by the switches, for the different detection policies. We also evaluate the throughput degradation due to BeCheck.

Our results, even if preliminary, are promising, and can be extended to consider applications, coherent in terms of behavior, running on completely different controllers, in order to detect misbehaviors in a more reliable way. We leave this extension for future work.

References

[1] A. El-Hassany, J. Miserez, P. Bielik, L. Vanbever, and M. Vechev, "SDNRacer: Concurrency analysis for software-defined networks," *SIGPLAN Not.*, vol. 51, no. 6, pp. 402–415, Jun. 2016.

[2] Floodlight. [Online]. Available: http://www.projectfloodlight.org/floodlight/

[3] The POX controller. [Online]. Available: https://github.com/noxrepo/pox

[4] M. Canini, D. Venzano, P. Peresini, D. Kostic, J. Rexford *et al.*, "A NICE way to test OpenFlow applications." in *NSDI*, vol. 12, 2012, pp. 127–140.

[5] The NOX Controller. [Online]. Available: https://github.com/noxrepo/nox

[6] The OpenDaylight Platform. [Online]. Available: https://www.opendaylight.org/

[7] ONOS - a new carrier-grade SDN network operating system. [Online]. Available: http://onosproject.org/

[8] K. Mahajan, R. Poddar, M. Dhawan, and V. Mann, "Jury: Validating controller actions in software-defined networks," in *IEEE/IFIP DSN*, June 2016, pp. 109–120.

[9] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: Detecting security attacks in software-defined networks." in *NDSS*, 2015.

[10] Maestro. [Online]. Available: https://code.google.com/p/maestro-platform/.

[11] S. Matsumoto, S. Hitz, and A. Perrig, "Fleet: Defending SDNs from malicious administrators," in *HotSDN*. New York, NY, USA: ACM, 2014, pp. 103–108.

[12] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "Vericon: Towards verifying controller programs in software-defined networks," *SIGPLAN Not.*, vol. 49, no. 6, pp. 282–293, Jun. 2014.

[13] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "Veriflow: Verifying network-wide invariants in real time," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, 2012.

[14] R. Majumdar, S. D. Tetali, and Z. Wang, "Kuai: A model checker for software-defined networks," in *IEEE FMCAD*, 2014, pp. 163–170.

[15] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock *et al.*, "Troubleshooting blackbox SDN control software with minimal causal sequences," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 395–406, 2015.

[16] D. Kreutz, F. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *HotSDN*. ACM, 2013, pp. 55–60.

[17] OpenFlow Switch Specification v1.5.1. [Online]. Available: https://www.opennetworking.org/images//openflow-switch-v1.5.1.pdf

[18] OpenFlowJ Loxi. [Online]. Available: https://github.com/floodlight/loxigen/wiki/OpenFlowJ-Loxi

[19] Java I/O, NIO, and NIO.2. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/io/

[20] Mininet: An Instant Virtual Network on your Laptop (or other PC). [Online]. Available: http://mininet.org/