Designing for Proactive Network Configuration Analysis

Magreth Mushi[1*] and Rudra Dutta[2]

[1]*Department of ICT, Open University of Tanzania*
[2]*Department of Computer Science, North Carolina State University*

[1]magreth.mushi@out.ac.tz, [2]rdutta@ncsu.edu

Abstract

Human operators are an important aspect of any computing infrastructure; however, human errors in configuring systems pose reliability and security risks, which are increasingly serious as such systems grow more complex. Numerous studies have shown that errors by human administrators have contributed significantly to misconfigurations of networks. The research community has reacted with development of solutions that largely directed at detecting and correcting misconfigurations statically, after they have been introduced into the configuration files. This is done either by checking against known good configuration practices or by data mining configuration files. Though to some extent such approaches are useful, they are in fact "treatments" rather than "preventions". Automated tools that abstract complex sets of network administration tasks have also been seen as a potential solution. On the other hand, such tools simply remove the possibility of human error one step, to the development of the workflow, and can have the effect of magnifying the risk of such mistakes due to their speed of operation. There is a need for a proactive solution that examine consequences of a proposed configuration before it is implemented.

In this paper, we describe the research design towards developing a proactive solution for misconfiguration problem. Then present the design and implementation for SanityChecker-an SDN-based solution for intercepting incoming configurations and inspecting them for human errors before committing to the devices. SanityChecker was tested by real-world network administrators and the results show that it can successfully improve network operations by overseen incoming configuration for human errors.

Keywords: *Software Defined Networks, Network Administration, Network Configuration, Misconfiguration, Research Design, Workflow, OpenDaylight, OVSDB, SanityChecker.*

1. Introduction

From numerous studies and incidents (Mushi et al., 2015), (Brown and Patterson, 2001), (Lee, 2013) it is proven that humans play a key role in any computing system or infrastructure, but at the same time involvement of humans has posed increasingly serious reliability and security issues in recent years. In particular, enterprise computer networks form a critical computing infrastructure that is vulnerable to human actions.

As computer networks become larger and more complex, the process of administering and managing networks has also become larger and impractical for completely manual configuration due to simple considerations of scale. The job of configuration has been increasingly shifted to automated protocols (Javvin-

* Corresponding author

Technologies, 2007). For example, routing tables were configured manually in the earliest days of the Internet, but in the last few decades they have become the domain of automated protocols such as Routing Information Protocol (RIP) and Open Shortest Path First (OSPF). However, such protocols do not eliminate the need for human involvement in network administration and management. Rather, they make them more complex (the administrator now needs to configure OSPF). And the expected reduction in the volume of administrative work evaporates in the face of rapidly increasing network sizes and scope.

Since the 1980's, there have been several research efforts geared toward reduction and elimination of network misconfigurations caused by human errors. Earlier research and tools (Feamster and Balakrishnan, 2005), (Yuan et al.,2006) were developed for traditional networks. These are based on a reactive approach of post-scanning configuration files to identify errors made during network configuration. They allow the misconfigurations to occur and then at a later stage reactively scan the network configuration files to find the errors which must then be corrected by an administrator. According to current studies (Khurshid et al., 2012), (Al-Shaer and Al-Haj, 2010), these reactive approaches have not proven successful in eliminating misconfigurations. At best, they mitigate and correct the impact of errors post-facto and cannot prevent.

With the emergence of Software Defined Networking (SDN) concepts and its wide acceptance by the research community, the job of the network administrator and manager is changing with considerations of automating common workflows like Virtual LAN (VLAN) configuration (Sezer et al., 2013). Such an evolution promises increased efficiency and correctness in configuring or reconfiguring networks and the possibility of abstracting workflows commonly prone to human errors into automated processes that will not make such errors. On the other hand, any vulnerability inherent in a set of scripts running automatically on demand (or a single command by an administrator to hundreds of devices) can magnify the risk of such mistakes to an unprecedented level; such scripts may well execute many hundreds of times before any reaction at a human time scale is possible. NEAt (Zhou et al., 2018), VeriFlow (Khurshid et al., 2012) and FlowChecker (Al-Shaer and Al-Haj, 2010) are recent SDN plug-ins that verify the incoming policies and IP forwarding rules from the controller applications as they are inserted. Rather than replacing (potentially faulty) human operation with faultless execution of (potentially faulty) automated scripts and tools, we have taken the approach that it is best to combine both worlds, and let the automated system provide a sanity check, in real-time, on a human administrator's operations. To the best of our knowledge, no similar tool has previously been advanced for verifying the human administrator's configurations in real-time as they are submitted to the devices through the controller.

In this paper, we address the design of the research approach followed to develop a proactive solution for network misconfigurations. Since this area has not been explored to a significant degree in literature, we start from first principles. We first seek to gain an understanding of the tasks network administrators must perform, and insight into what misconfigurations happen, and why. To this end, we use various methodologies, including structured and unstructured interviews, online surveys, and follow-up discussions. We then investigate the relative impact of various misconfigurations on network reliability by using network simulation tools. Finally, we advance an approach using the Software Defined Networking paradigm for

automatic just-in-time detection and prevention of mistakes in network administrators and managers performing configuration tasks.

The paper is organized as follows. In Section 2 we review the existing literature on the subject of human factors in computing and solutions available in traditional and SDN network infrastructures. In Section 3 we explain the approach that our design is based on. Section 4 provides the overall design of the SanityChecker plug-in. Section 5 gives a description of the plug-in implementation while Section 6 provides the test and evaluation performed on SanityChecker. Finally, Section 7 provides conclusions of our research and a description of future work anticipated for subsequent evolution of SanityChecker.

2. Related work

The role of human factor in Information Technology (IT) in general has been examined as far back as the 1980's across a range of disciplinary perspectives, including aircraft, bank databases, and the telephone network (Brown and Patterson, 2001), (Kantowitz and Sorkin, 1983). These researchers found that 20-70% of system failures are attributable to human operator errors. In network administration and management few such studies (Wool, 2004), (Feamster and Balakrishnan, 2005), (Mahajan et al., 2002) have been conducted and researchers again found that the misconfigurations inherent in the devices logs were mainly due to errors by network administrators, poor understanding of configuration semantics, or complex rule sets which are too difficult for administrators to manage effectively. On the other hand, other researchers have focused on tools to automate network administration and management in order to reduce human interaction. Buchmann (2008) described general concepts of network management and provided a prototype implementation of a network management system software to facilitate administration of large, heterogeneous networks. Colwill and Chen (2009) provided similar insights for the provider's network and proposed several joint-vendor approaches that will ensure consistency across different providers. Some of the tools developed involve Router Configuration Checker (RCC) (Feamster and Balakrishnan, 2005) for static analysis of configuration files to find faults in Border Gateway Protocol (BGP) configurations. Along the same line, other tools like FIREMAN (Yuan et al., 2006) and AT&T configuration checker (Feldmann and Rexford, 2001) were developed. For dynamic analysis, MINERALS (Le et al., 2009), a tool based on data mining techniques was developed. MINERALS apply association rules mining to the routers configuration files across an administrative domain to discover local network-specific policies. Deviations from these local policies are considered potential misconfigurations. Other tools such as EDGE (Caldwell et al., 2004) follows the same approach to minimize misconfigurations in networking devices.

The main deficiency of these tools is that they are reactive and based on traditional network infrastructure. They are based on post scanning of configuration files to determine misconfigurations which are then corrected manually by the administrator. It is more logical to be proactive and prevent the misconfigurations or minimize the chance of them happening. Therefore, enhance reliability and security while eliminating the time it takes to post scan files and correct errors.

With the emergence and promises of SDN, several tools are being developed to enhance the performance of the infrastructure. For the misconfiguration problem, Khurshid et al. (2012) developed VeriFlow-SDN based tool to verify incoming

OpenFlow rules from the controller for possible misconfigurations. Zhou et al. (2018) recently developed NEAt-a tool to automatically correct policies from controller applications, while Kazemian et al. (2013) developed NetPlumber - a real time policy checking tool based on Header Space Analysis (HSA). The main limitation of these tools is that they are not looking for misconfigurations introduced into the system by network administrators, considering them infallible. Their approach is based on rules and policies from protocols and controller applications.

3. Research Design

Figure 1 below shows the general approach followed in our research. This approach was motivated by the concepts advocated by the newly emerging area of Science of Security (SoS) (Herley and v. Oorschot, 2017). We began by studying the problem at hand and later proposed a solution based on the findings of our study. In studying the problem, we started with Stage 1-Understanding misconfigurations problem. In this stage we used several research tools: literature search, professional board examinations review, interviews, online survey, and experiments. The results of these studies were reported in our earlier papers (Mushi and Dutta, 2017; Mushi et al., 2015). In Stage II-Understanding the impact of misconfigurations, we modeled an enterprise network and subjected the model to the human errors that were found in our study. Finally, in Stage III-Engineering Solution, we used SDN platform to design and implement the proactive solution to the human errors problem in enterprise network. The design, implementation and evaluation of this solution form the basis for this paper.

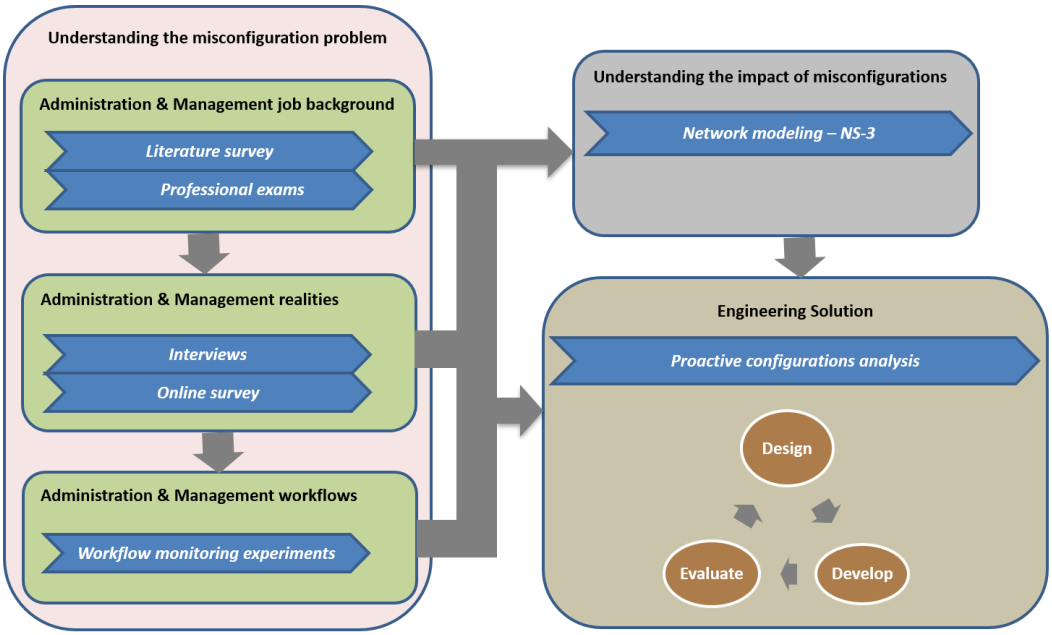

Figure 1: General Design

The foundation of SanityChecker design outlined in Section 4 is based on the results reported in our earlier studies based on Stage I and Stage II where we studied human factors in network administration and management.

Our findings in those studies, obtained from a survey of network administration and management practices across an array of enterprises, indicated that most network administrators are neither highly paid, nor highly skilled, but have an increasing diversity of workload. In particular, only about two-thirds of network administrators and managers hold a bachelor's degree in a field related to computing. Of the rest, about half have a college degree or some college education in a field unrelated to computing, while the rest have no college education. Every person surveyed anonymously admitted to having made mistakes in configuration, and all but a vanishing proportion identified no other contributing cause than themselves (a small minority attributed misconfiguration to their organizational policies).

The other pertinent finding from our previous study is the nature of the errors most commonly made, which guides us in our design of the SanityChecker. We identified several common misconfigurations by network administrators which led to categorize them into three categories below, adapting the human error model developed by Reason (1997).

- **Slips**: Misconfigurations that happened due to human error in executing a configuration workflow. The administrators knew the correct configuration, but they forgot to write it properly during workflow execution. For example, the most common misconfiguration in this category is forgetting "add" keyword when adding a VLAN to a trunk port which resulted in wiping the existing list of VLANs in the port; which in turn denied network access to end users assigned to those VLANs.
- **Mistakes**: Misconfigurations that were introduced into the network due to lack of knowledge on a particular aspect of configuration workflow, so the mistake was made during workflow planning as well as workflow execution. The most common misconfiguration we found in this category is using default configurations for Spanning Tree Protocol (STP). Out of 14 interviewees and 33 survey respondents, only 3 were aware of STP parameters and how they can be configured to perform efficiently and without causing loops in the networks.
- **Violations**: Misconfigurations that were done intentionally by network administrators based on circumstances while knowing it to be against the rules or best practices. For example, not turning off the default native VLAN 1 despite the fact that they are aware of the consequence that might arise. According to two of our interviewees, these kinds of misconfigurations mostly happens during peak times when there is more work to do than the available human resource. Therefore, they focus on configurations that will make the network function and leave other configurations that are not required for basic network functionality (mostly referred to as "cosmetics"). Due to this behavior, the network is left vulnerable to attacks or other reliability problems.

SanityChecker is designed to detect and respond to these misconfigurations by prompting the administrator to fix the problem before committing the configuration to the device. At this point SanityChecker performs checks for the commonest misconfigurations found in our previous research; if a command that is not part of the system is submitted, the administrator will receive notification about that, and the checking will terminate. We envision that in the future a system such as

SanityChecker will provide an interface for administrators to add new misconfigurations to be checked, as such misconfigurations are identified over time.

4. SanityChecker Design

We chose to design and develop SanityChecker within OpenDaylight (ODL) controller (OpenDaylight.org, 2014). ODL is a modular open platform for customizing and automating networks, it uses the Model-Driven Service Abstraction Layer (MD-SAL). In OpenDaylight, underlying network devices and network applications are all represented as objects, or models, whose interactions are processed within the SAL. Despite the complexity of ODL, it makes the design far more realistically deployable. In ODL terms, SanityChecker exists as a Model Driven Service Abstraction Layer (MD-SAL) based plug-in. SanityChecker is designed to act as a gateway between the network administrator's configuration interface and the southbound protocols. As seen in Figure 2 below, our proactive solution (labeled SC service) resides in the ODL controller as part of its service. It is using RESTCONF API for northbound communication and OVSDB protocol for southbound communication.

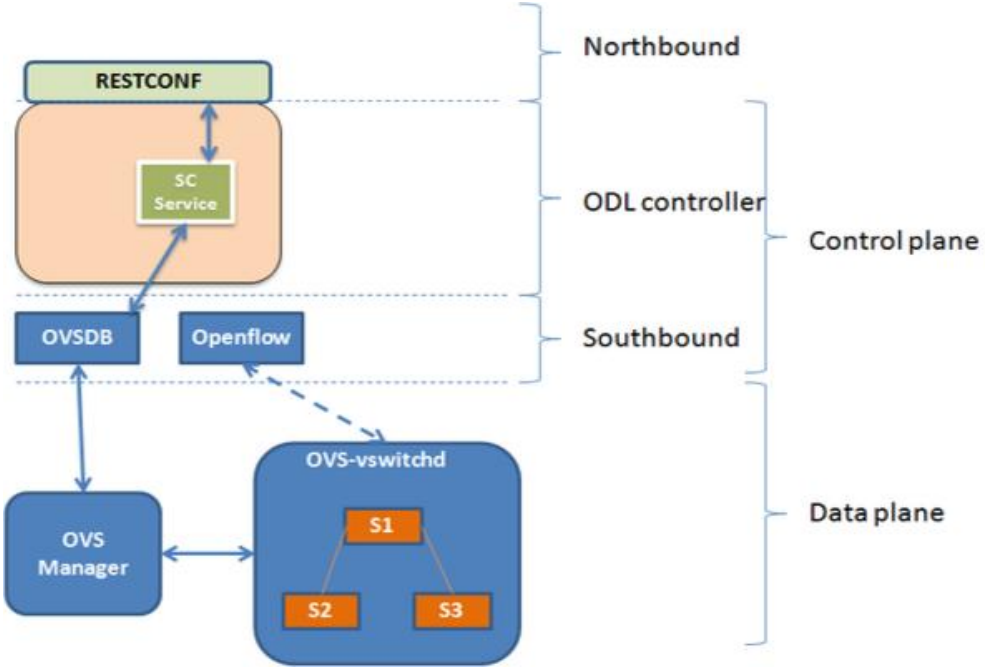

Figure 2: SanityChecker Design in the ODL Controller

The flowchart in Figure 3 shows algorithmic steps performed by the administrator and the plug-in. In the beginning, the administrator submits configuration commands one at a time or by using a text file with commands (one per line). Then the SanityChecker modules (described in detail next) will interact as follows: First, the InConfig Server will receive the command(s), parse them and send the details (such as switch name, VLAN ID) to CheckEngine for the checking process. Next the CheckEngine will check the individual configuration commands for violations, mistakes, or slips (details given in Section 4.2. If an error is detected, the administrator will be prompted to make correction. If no error is detected,

CheckEngine will pass the command to the OutConfig server to commit to the device through the OVSDB manager.

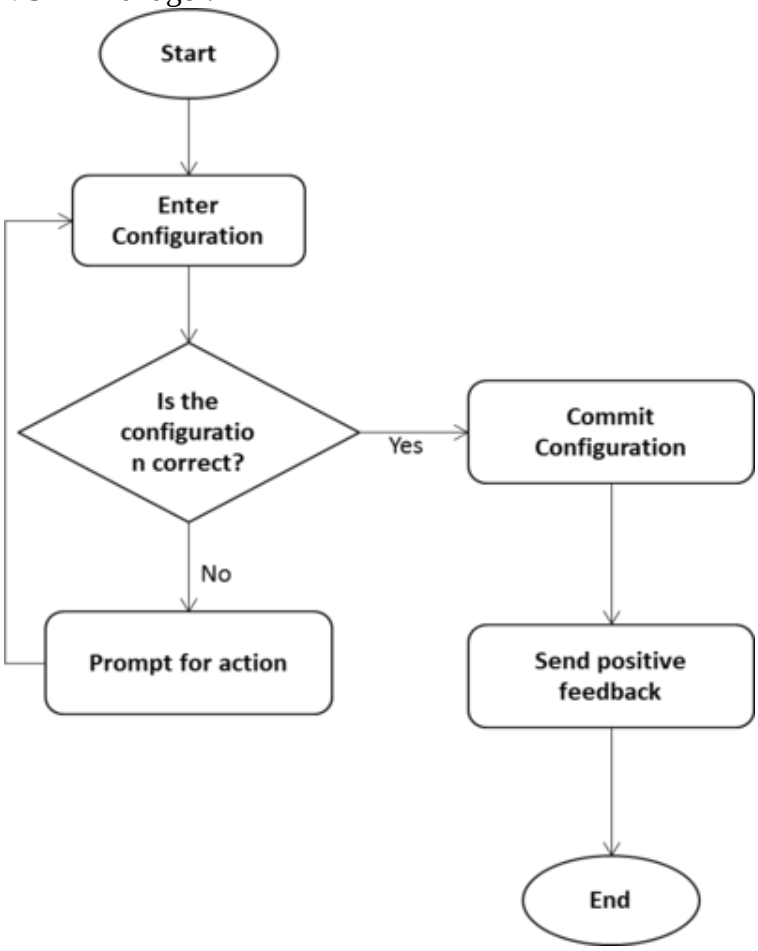

Figure 3: SanityChecker Flowchart

4.1. SanityChecker Modules

Figure 4 shows the three main SanityChecker modules and their interactions. A brief description of each module follows.

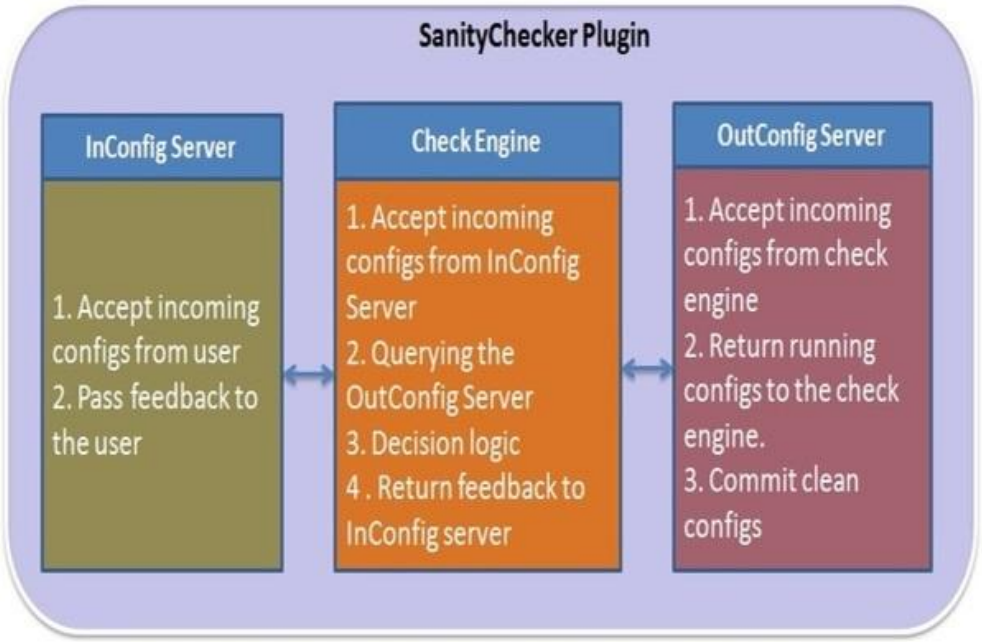

Figure 4: SanityChecker Modules

4.1.1. InConfig Server: This module receives the incoming configurations and passes them to the CheckEngine module. It also receives feedback from CheckEngine and passes it back to the administrator via RESTCONF. This module takes configuration commands from a text file that contain one configuration command per line or takes in one configuration command at a time from the administrator.

For the case of a text file, the administrator provides absolute link to a file, and the server fetches the file from that location. If the path is invalid or for some other reason the file could not be located, the InConfig server will notify the administrator. If the file is found, the server retrieves one command at a time and pass it to the CheckEngine. Upon receipt of the outcome from the CheckEngine, this server writes the returned message to the output file which is saved in the user home directory. Upon completion of the execution, the administrator will then have to check the output file to see which commands successful and which ones were not. For unsuccessful commands, more details about the error are also provided in the output file.

Upon submitting individual command, the administrator provides one configuration command at a time. This server will pass the command to the CheckEngine and return the outcome to the administrator.

4.1.2. CheckEngine: This is the module that performs the checking logic. It communicates with the InConfig and/or the OutConfig servers to accomplish its tasks depending on the nature of the incoming configuration. This module receives individual configuration commands from the InConfig Server, then checks them for known mistakes, slips, or violations. If any of these human errors is found, the CheckEngine returns an appropriate message to the InConfig Server. If none of the errors is found, the CheckEngine passes the command to the OutConfig Server. In performing the checks, this module might need to consult the OutConfig server to

retrieve information from the intended devices as described in detail in Subsection 4.2.

4.1.3. OutConfig Server: This module serves CheckEngine requests by either retrieving configurations from the devices or committing configurations to the devices. As explained in the Section 4.1.2 above, the CheckEngine sends its requests to this server in two occasions; either it needs more information from the devices to make its decision or if it has determined that the configuration command has no problem so it should be committed. The OutConfig server serves these two specific functions. After completing each task, this server passes the outcome to the CheckEngine which in turn pass it to the InConfig server. This outcome might also be an error messages from OVSDB that cannot be solved by SanityChecker and therefore should be passed to the administrator.

4.2. Performing the Checks

How does SanityChecker determine if the configuration is correct? For ease of understanding we will explain this by using an example of adding VLANs to the trunk port in OVS setup. Based on OVS configuration semantic, when an administrator is adding VLAN to a list of available VLANs in a trunk port, she has to specify all the VLANs (i.e., the existing and the new VLANs) in the command. If she does not do that, the new list of the VLANs will replace the existing VLANs causing denial of service to the end users in the earlier VLANs.

In order to perform the checks, the InConfig will extract the switch name and port, then pass them to the CheckEngine with instruction on what is to be performed in the port, in this case is adding new VLANs. The CheckEngine will check with OVSDB manager if the switch and the port exist; if not, the error will be sent back to the administrator. If they both exist, the CheckEngine will consult the OutConfig server to retrieve information about the existing VLANs in the port. If the port has some VLANs, the CheckEngine will check if those VLANs are included in the command that was submitted. If the VLANs were included, the CheckEngine will submit the command to the OutConfig server which will in turn commit the configuration to the device through the OVSDB server, and the success message will be returned to the administrator through the same channel. Otherwise, the error message will be sent back with instructions on what is wrong.

5. SanityChecker Implementation

In this section we explain the setup of our development environment, the implementation of the plug-in, and give two examples of the checks that are performed by the plug-in. As discussed in Section 4 we used OpenDaylight controller and Mininet network emulator. Our network consisted of three switches connected to the controller through the OVSDB manager. The plugin's InConfig server receives incoming configuration through RESTCONF; then parses the command and sends the required parameters to CheckEngine to perform the checks. When all the checks are complete, CheckEngine will send the feedback to the InConfig server, which will in turn send the feedback to the administrator.

Two examples are explained in Sections 5.1 and 5.2 below. Example 1 shows a configuration error that can either be categorized as a *Violation* or a *mistake*; while

Example 2 shows configuration error that is a slip. It should be noted that at this point the plug-in is not taking any action to correct the commands. When it encounters a configuration that has a violation, mistake, or slip, it returns feedback to the user including a brief description of the consequence of the configuration being submitted.

5.1. Sample Scenario 1

Figure 5 shows an OVS configuration command that is intended to add a port to a switch. This command is syntactically correct, but it is adding a port without specifying its status (i.e., access or trunk). By default, the OVSDB protocol will assign trunk status to any port that is added without specifying its status. In our study of human network administrators (Mushi et al., 2015), it was clear that in many cases administrators do not specify port status but instead use the default configuration. The problem with the default configuration is that a trunk port will pass traffic for all (or a number of specified) VLANs. Therefore, if this port is going to be used as an access port, users can sniff traffic that is not intended for them. To prevent such errors, the SanityChecker plug-in will prompt the administrator to specify the status of the new port to be added as indicated in Figure 5.

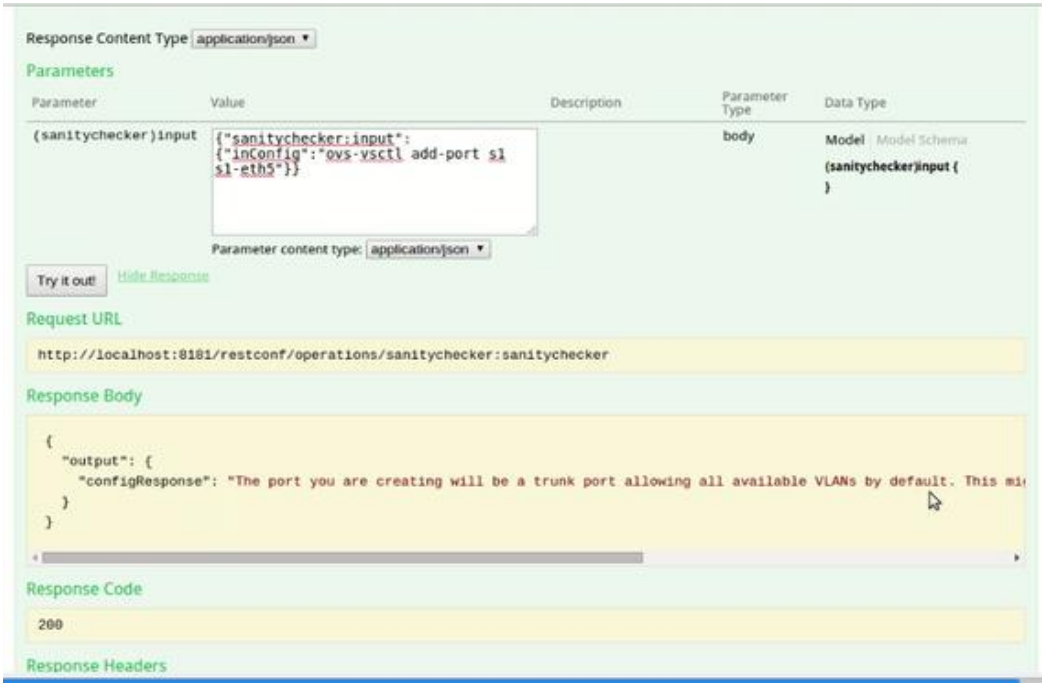

Figure 5: Example of a Wrong Configuration in Adding Port

As you can see in Figure 6 the port was added successfully after the administrator specified the status of the new port (i.e access port in VLAN 300).

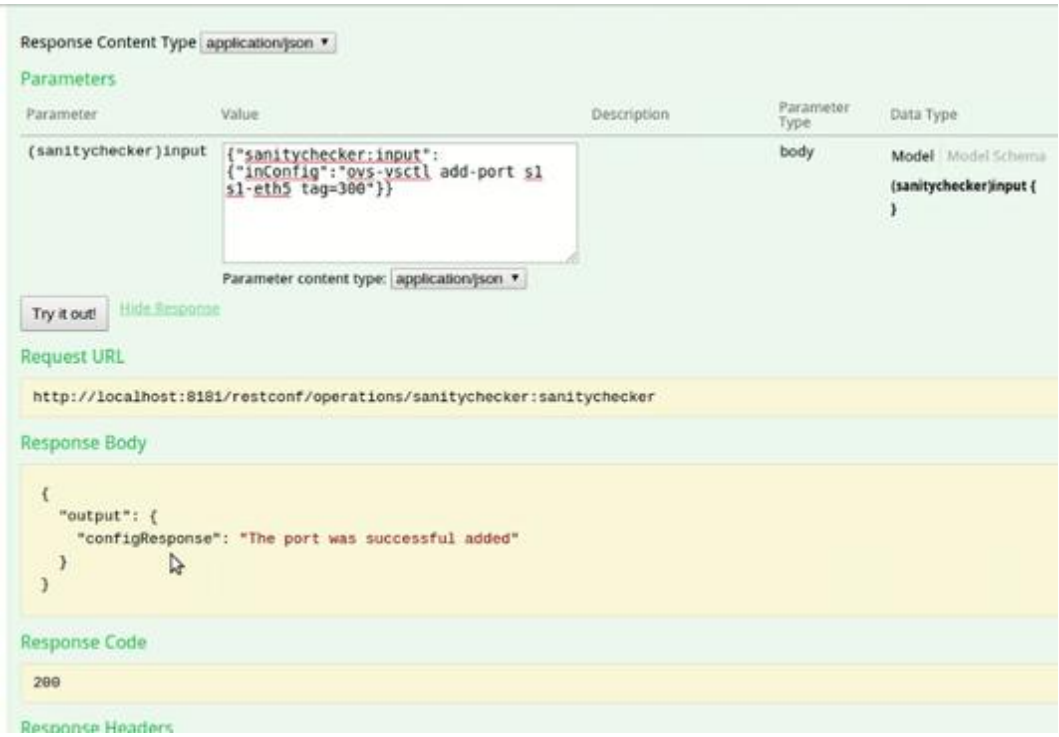

Figure 6:Example of a Right Configuration in Adding Port

5.2. Sample Scenario 2

It was clear in our study that administrators make accidental mistakes (e.g., configuring a port that was not intended because the administrator confused a port's name with another port's name, or confused a physical port with a logical port). For example, Figure 7 shows OVS configuration command that is trying to delete port s1-eth1 from a virtual switch s1 (i.e., `ovs-vsctl del-port s1 s1-eth1`). This particular command will delete a port that is part of the configuration and is currently sending and receiving data. Since this command has the potential to cause Denial of Service (DoS) in the current topology by deleting the port, the plug-in will prompt the administrator. If the administrator is absolutely sure she wants to perform this action, the plug-in will then accept the re-submission.

Figure 7: Example of a Command that will Cause DoS

The command in Figure 8 was successfully accepted because port s1-eth5 (created in Figure 6) was not connected to any other port and therefore was not active. Based on its status the CheckEngine determined that it is safe to be deleted.

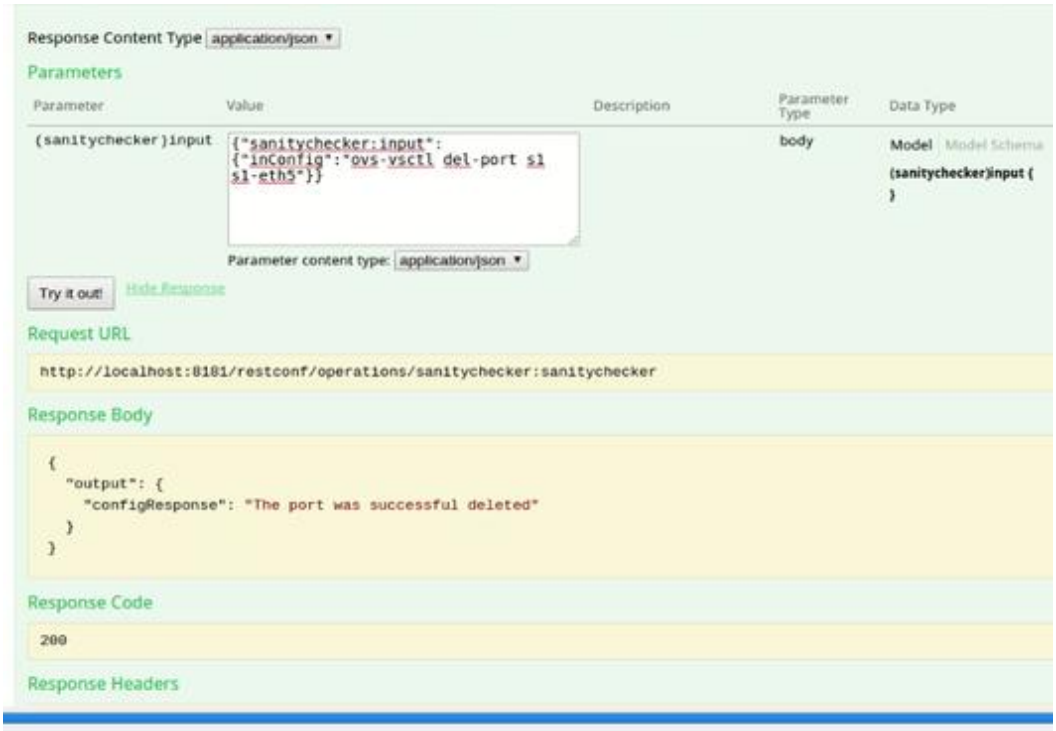

Figure 8: Example of a Command that will not Cause DoS

6. Test and Evaluation

To validate our concept, we asked volunteer participants to use our prototype system in performing network administration tasks. We were able to recruit 20 participants, all personnel (students or staff) of North Carolina State University. The majority of the participants were apprentice network administrators: graduate students who had received instruction in network administration and management, but with no previous professional experience. A small minority were participants with professional experience of managing real networks.

We installed the ODL controller with SanityChecker, a network in mininet, and sample test cases, on virtual machines on the campus computing system. Test participants reserved the image, performed the tests, and filled in the test cases template which was then returned to us. In testing SanityChecker, participants performed both positive (using valid input) and negative (using invalid input) tests that they determined themselves. That is, when a participant made a configuration error, it was typically an intentional error to test SanityChecker's response. Ten participants provided data on every test case; the rest provided partial data. We gathered data for the functional and operational requirements provided in Table 1. For gathering operational test results, we used the Dstat tool.

Table 1: Sample Functional and Operational Requirements

Type	Requirement	
Functional	F1	Should be able to receive incoming configurations
	F2	Should be able to check incoming configurations agains well known human errors and best practices.
	F3	Should be able to save configurations to the intended device.
	F4	Should be able to return appropriate errors for invalid input e.g. malformed command or path to the input file, incorrect set of parameters etc.
Operational	O1	Provide minimal network overhead
	O2	Provide minimal CPU overhead
	O3	Provide minimal memory overhead

6.1. Functional Testing

In functional testing, our goal is to verify that the system provides functionality as designed. Table 2 shows the false positive and false negative data. In our system tests, false positive indicates that the system accepted input that was a misconfiguration but did not categorize it as such; for example, failing to flag deleting a port that does not exist. False negative indicates that the system accepted input that was not a misconfiguration but categorize it as such; for example, flagging the addition of a port that has a proper status specified. As indicated in Table 2, one tester reported one false positive out of the 10. He found that SanityChecker successfully

accepted his command to add port even though he did not specify to which switch the port should be added. For the false negative, all testers found one case; i.e., SanityChecker refused the command to update the trunk port that already has some VLANs, it returned the error that says, “there exist some VLANs in the port”. These issues will be addressed in the next version.

Table 2: False Positive and False Negative

	Yes	No
Misconfiguration	**90%**	**10% (False positive)**
Not misconfiguration	**10% (False negative)**	**90%**

6.2. Operational Testing

In operational testing, our goal was to check the performance of the system with the SanityChecker running, using different performance metrics. The goal is to ensure that the overhead of incorporating SanityChecker does not negate its benefits. For performance comparison, we collected the data in two scenarios: (i) Controller running without SanityChecker and (ii) Controller running with SanityChecker.

6.2.1. Network Overhead: The data presented in Figure 9 below illustrate network usage statistics over time when the controller was running **without** SanityChecker installed. Figure 10 shows the same statistics when the controller was running with SanityChecker installed. For the different times we observed the network usage, the controller alone was using up to 8500Kbps (8.5Mbps) while the controller with SanityChecker was using up to 9000Kbps (9Mbps).

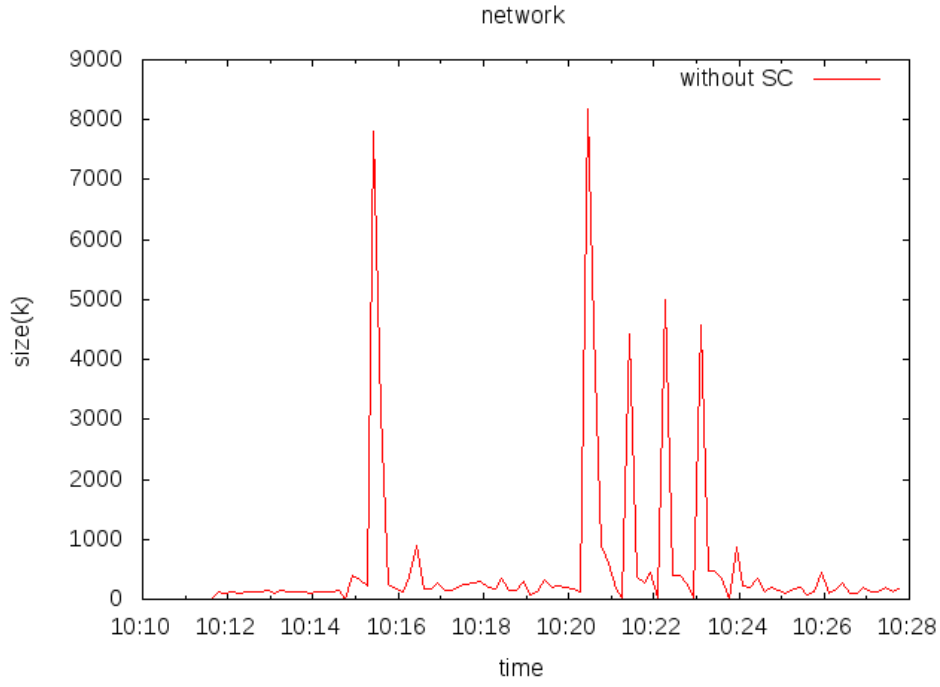

Figure 9: Network Usage Data without SanityChecker

As Seen below, we also observed more spikes in network usage with the SanityChecker and these represent the time when queries were being sent to OVSDB. As was expected, network overhead is still minimal since it works as a service inside ODL controller.

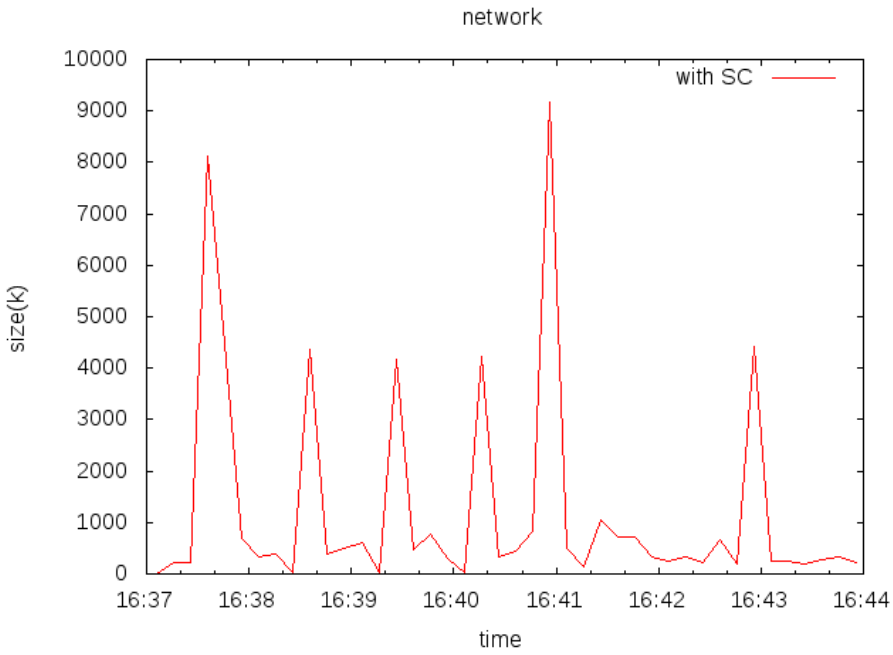

Figure 10: Network Usage Data with SanityChecker

6.2.2. CPU Overhead: We also checked the CPU usage for the two scenarios over time as seen in Figures 11 and 12. The maximum CPU usage was about 9% for the controller alone and about 13% with SanityChecker.

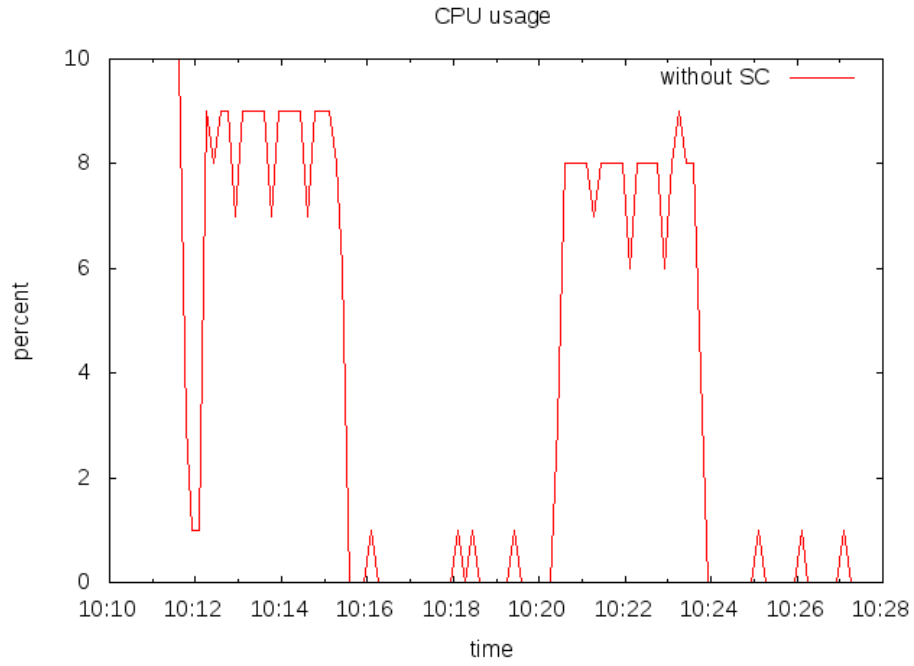

Figure 11: CPU Usage Data without SanityChecker

As it was the case with the network, the time we observed the increase in CPU usage was the time we executed queries that needed to fetch data from OVSDB to accomplish certain checks.

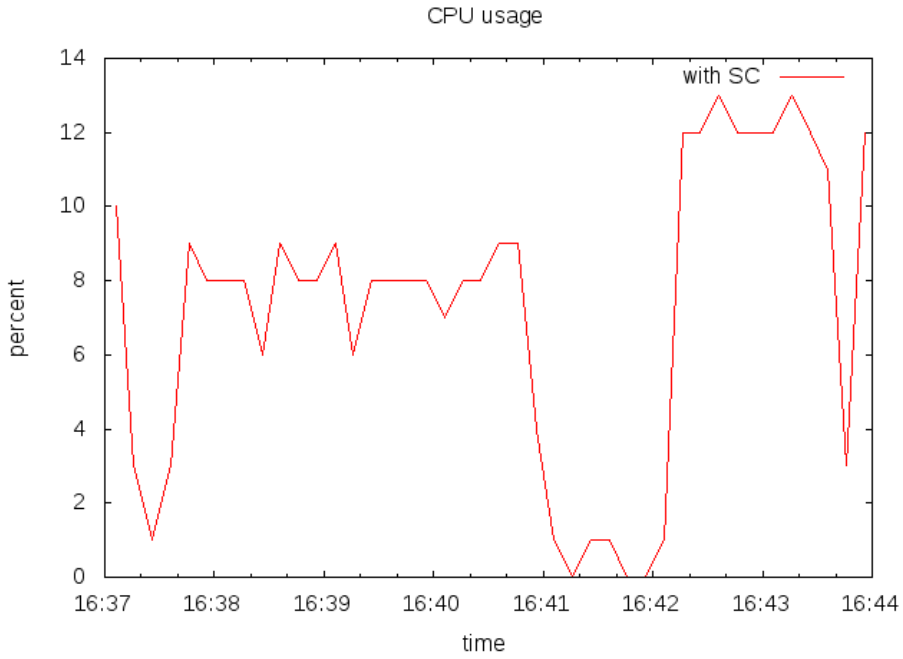

Figure 12: CPU Usage Data with SanityChecker

6.2.3. Memory Overhead: We also checked server buffer and cache memory usage for the two scenarios and observed the same behavior like network and CPU. Figure 13 shows server memory usage when the controller was running without SanityChecker.

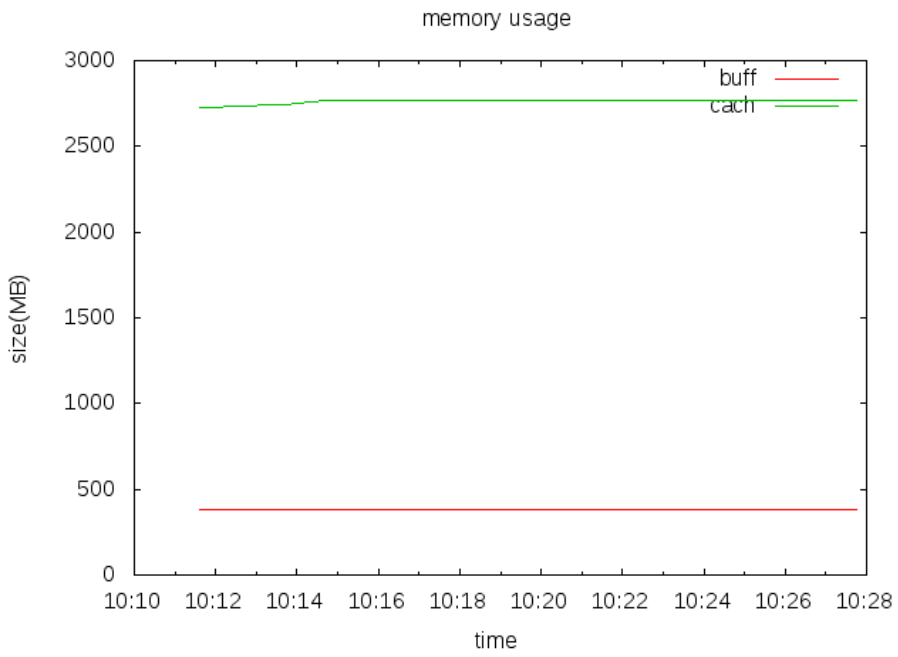

Figure 13: Memory Usage Data without SanityChecker

Figure 14 shows memory usage with SanityChecker. As with the network and CPU data presented above, the memory overhead was also minimal because the increase in cache memory usage was about 200MB after installing SanityChecker while the buffer memory usage showed no increase.

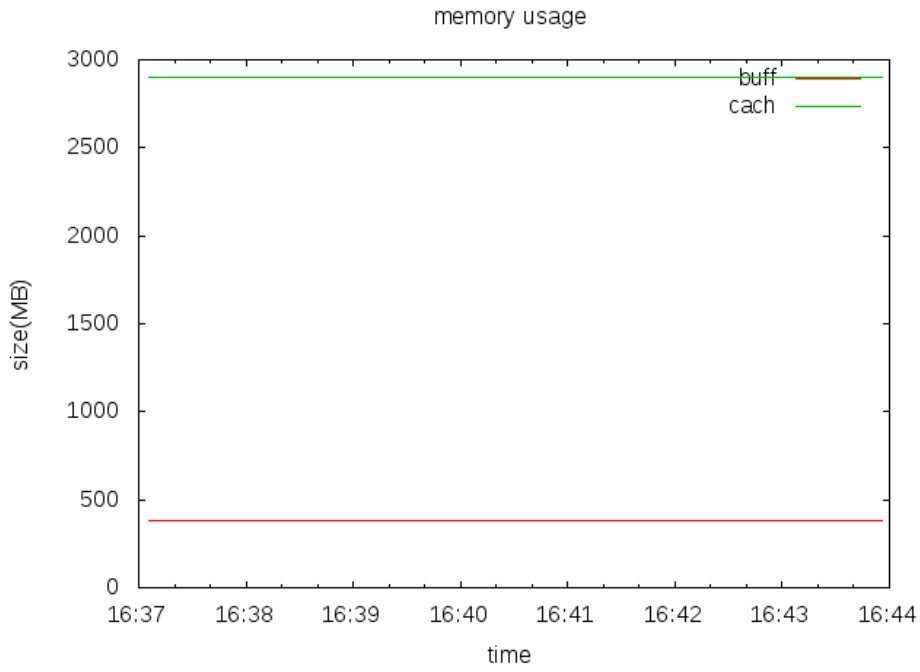

Figure 14: Memory Usage Data with SanityChecker

7. Conclusion and Future work

We have presented SanityChecker, a network administration oversight system, implemented as an SDN plug-in, that can work in conjunction with human administrators to eliminate the inevitable human errors from being implemented in the network, by raising flags when operations suspected to be typical mistakes are attempted. We have successfully designed and prototyped the system, and our testing results show promise that such an approach can be a powerful one.

There are a few areas that we believe deserve further study. First, it is important to separate and extend the error checking logic in the CheckEngine module. The module should be able to perform network-wide checks instead of focusing on individual commands for individual devices. Second, for scalability purposes, SanityChecker should provide interface for adding newly detected misconfigurations in future (the present version only performs checks for specific misconfigurations that we discovered to be common in the course of our prior research). Overall, we feel this area is a promising one worth further research.

References

1. Al-Shaer, E. and Al-Haj, S. (2010). Flowchecker: Configuration analysis and verification of federated openflow infrastructures. *In Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration, SafeConfig '10*, pages 37–44, New York, NY, USA. ACM.
2. Brown, A. B. and Patterson, D. A. (2001). Embracing failure: A case for recovery-oriented computing (roc).
3. Buchmann, D. (2008). *Verified network configuration: Improving network reliability*. PhD thesis, Faculty of Maths and Natural Sciences, Univ. of Fribourg, Switzerland.

4. Caldwell, D., Gilbert, A., Gottlieb, J., Greenberg, A., Hjalmtysson, G., and Rexford, J. (2004). The cutting edge of IP router configuration. *SIGCOMM Comput. Commun. Rev*., 34(1):21–26.
5. Colwill, C. and Chen, A. (2009). Human factors in improving operations reliability. *Annual IEEE CQR International Workshop*.
6. Feamster, N. and Balakrishnan, H. (2005). Detecting BGP configuration faults with static analysis. *In Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation* - Volume 2, NSDI'05, pages 43–56, Berkeley, CA, USA. USENIX Association.
7. Feldmann, A. and Rexford, J. (2001). Ip network configuration for intradomain traffic engineering. *IEEE Network*, 15(5):46–57.
8. Herley, C. and v. Oorschot, P. C. (2017). Sok: Science, security, and the elusive goal of security as a scientific pursuit. *2017 IEEE Symposium on Security and Privacy (SP)*.
9. JavvinTechnologies, I. (2007). *Network Protocol Handbook*. Javvin Press, fourth edition.
10. Kantowitz, B. H. and Sorkin, R. D. (1983). *Human Factors: Understanding People-System Relationships*. Wiley, first edition.
11. Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., and Whyte, S. (2013). Real time network policy checking using header space analysis. *In Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, NSDI'13*, pages 99–112, Berkeley, CA, USA. USENIX Association.
12. Khurshid, A., Zhou, W., Caesar, M., and Godfrey, P. B. (2012). Veriflow: Verifying network-wide invariants in real time. *In Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN '12*, pages 49–54, New York, NY, USA. ACM.
13. Le, F., Lee, S., Wong, T., Kim, H. S., and Newcomb, D. (2009). Detecting network-wide and router-specific misconfigurations through data mining. *IEEE/ACM Transactions on Networking*, 17(1):66–79.
14. Lee, M. (2013). Linkedin just one of thousands of sites hit by DNS issue: Cisco. Mahajan, R., Wetherall, D., and Anderson, T. (2002). Understanding BGP misconfiguration. *SIGCOMM Comput. Commun. Rev*., 32(4):3–16.
15. Mushi, M. and Dutta, R. (2017). Human factors in network reliability engineering. *Journal of Network and Systems Management*, 26(3):686722.
16. Mushi, M., Murphy-Hill, E., and Dutta, R. (2015). The human factor: A challenge for network reliability design. *In Design of Reliable Communication Networks (DRCN), 2015 11th International Conference on the*, pages 115–118.
17. OpenDaylight.org (2014). The opendaylight SDN controller.
18. Reason, J. (1997). *Managing the Risks of Organizational Accidents*. Ashgate, first edition.
19. Sezer, S., Scott-Hayward, S., Chouhan, P. K., Fraser, B., Lake, D., Finnegan, J., Viljoen, N., Miller, M., and Rao, N. (2013). Are we ready for SDN? implementation challenges for software-defined networks. *IEEE Communications Magazine*, 51(7):36–43.
20. Wool, A. (2004). A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67.
21. Yuan, L., Chen, H., Mai, J., Chuah, C.-N., Su, Z., and Mohapatra, P. (2006). Fireman: a toolkit for firewall modeling and analysis. *In 2006 IEEE Symposium on Security and Privacy (S P'06)*, pages 15 pp.–213.
22. Zhou, W., Croft, J., Liu, B., Ang, E., and Caesar, M. (2018). Automatically correcting networks with neat. *In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 595–608, Renton, WA. USENIX Association.