Firewall Implementation in the Frenetic Language for OpenFlow Networks

Benjamin Boren, Erin Lanus, Matt Welch
Computer Science
Arizona State University
Tempe, Arizona

***Abstract*— Pyretic is a python implementation of the Frenetic language abstraction to manage OpenFlow controllers. Arguments in favor of Frenetic claim it to be a simpler and higher-level way to manage OpenFlow controllers than the lower-level POX controller on which Pyretic rests. We implemented a proactive-reactive-hybrid firewall design in both POX and Pyretic and compared the two implementations in regards to number of flow entries, code size, and latency. We found Pyretic to not be as simple as some have claimed and offered less flexibility, less ease of use, and poorer performance on several metrics than the POX implementation.**

***Keywords*— OpenFlow; POX; Pyretic; SDN; firewall**

I. Introduction

The developers of the Frenetic language claim that advanced capabilities in software-defined networking (SDN) are implemented more easily through the abstractions of a higher-level language, such as Frenetic, versus implementing these capabilities directly. For details on SDN and Frenetic, we refer readers to [1][2][3][4][5]. An evaluation of the Frenetic language comparing applications in Frenetic against equivalent applications in NOX used the following metrics: lines of code, controller traffic, and aggregate traffic [1]. The lines of code metric was used to compare the complexity of an application, the controller traffic metric was used to compare the amount of traffic in kilobytes between the switch and the controller, and the aggregate traffic metric was used to compare the amount of traffic on the network. Evaluation was performed for three microbenchmarks using ICMP packets and HTTP traffic and using three different policies. For each of the policies and microbenchmarks, the Frenetic application was shown to require fewer lines of code than the NOX application, and the Frenetic performance was comparable to the NOX performance for controller and aggregate traffic. Additionally, Frenetic and NOX were compared in terms of scalability for each microbenchmark. Frenetic was shown to perform as well or better than NOX for the controller traffic metric as the number of hosts was scaled from 1 to 50. However, NOX was shown to outperform Frenetic when the optimal forwarding policy utilized wild card rules.

We feel that this evaluation is lacking in two fundamental measures: delay imposed by the Frenetic run-time system and effect of policy constructs in Frenetic on the size of the flow table. In this paper, we present a detailed comparison of expressing controller functionality in Pyretic, a python implementation of the Frenetic language, and the standard OpenFlow POX controller. For the comparison, we implemented the same firewall controller design in both Pyretic and POX. The two implementations were analyzed in two primary areas: performance measures of the actual firewall functionality (e.g. number of flow entries and latency) and objective and subjective measures of ease of expressing functionality (e.g. lines of code, size of code base, and suitability of language constructs to implement a particular task).

The organization of our paper is as follows. We discuss the design of the firewall implementations in POX and Pyretic as well as the experimental setup in §II. Results for each of the metrics chosen and discussion of possible interpretations appear in §III. We present the conclusions of our comparison in §IV and possible future directions in §V.

II. Experimental Design

A. Common Features of Both Implementations

In order to test the capabilities of the Pyretic language and not a specific whitelist firewall design, the following design features of the firewall were common to both implementations. Only TCP traffic was filtered by the firewall; ARP and ICMP packets were allowed through without filtering. A configuration file containing rules was

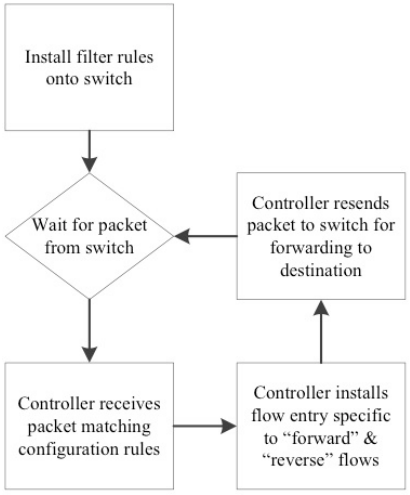

Fig. 1. Proactive-reactive-hybrid firewall flow of control

provided to the program as a text file with one rule per line. A rule consisted of a 4-tuple of the form *source-ip source-port destination-ip destination-port* where any of the values in the tuple could be specified as a wild card via the string *any*. At the beginning of execution, the program parsed the text file and maintained the rules in memory. Additionally, the controller proactively installed flow entries at the switch to allow flows matching the rules, though the implementation of this functionality differs between implementations. As packets arrive at the controller, the packet is matched against rules from the configuration file, and additional flow entries are installed at the switch for allowed flows. The controller then returned the packet to the switch for normal forwarding to the intended destination. The flow of control of the firewall design is summarized in figure 1.

B. POX Implementation

The POX firewall was implemented as a proactive-reactive-hybrid firewall that allows TCP traffic matching rules specified in a configuration file and drops all other TCP traffic. A packet matching a rule in the configuration is forwarded to the controller which then adds two flow entries in the switch: one to allow traffic on the forward path (i.e. from *source-ip* to *destination-ip*), as well as one to allow traffic on the return path (i.e. from *destination-ip* to *source-ip* of the matching packet). This implementation was chosen as it minimized the number of packets sent to the controller.

C. Pyretic Implementation

In the Pyretic implementation, we adhered closely to the design, while using the Pyretic language capabilities wherever possible. The principal construct in Pyretic is the policy, a function on a packet that returns a set of packets. Filter policies are functions that do not modify the packet. An example of a filter policy is *match(f=v)* where *f* is a field and *v* is a value. For example, *match(srcip=pkt_in['sourceip'], srcport=pkt_in['sourceport'])*, would return a packet if the packet's source ip was *sourceip* and source port was *sourceport*; it would not return any packet that did not match on both of these fields. Another filter policy is *drop*, which simply returns the empty set. Non-filter policies may modify the packets returned. For example, *fwd(x)* sets the packet's outport to *x* before returning it. Similarly, *flood()* takes a packet and returns a set of packets with one copy of the packet set to the outport for each port on the spanning tree. Negation can be performed on policies and policies can be composed under conjunction (also called union or *sequential composition*) and disjunction (also called *parallel composition*). *Dynamic policies* have behavior that changes over time, and a *query policy* is a type of dynamic policy. A query policy registers a *callback*, a function that matching packets are passed to, and allows a programmer to group packets having identical values in specified fields (such as packets of a specific flow) and limit how many packets from a group are sent to the callback.

The Pyretic firewall was implemented as the union of two policies. The first policy was further defined as the union of sets of packets that matched rules from the firewall configuration file. The second policy was defined as a query to return the first packet of each flow for all TCP traffic, which was then matched to the configuration rules to update the policy in a reactive manner, if necessary. Our initial intention was to disallow any traffic not matching a rule in the configuration file to reach the controller, but this did not work as intended in practice.

D. Experimental Setup

We used the Mininet simulator for data collection. This tool is flexible enough to allow multiple host instantiation and complex topologies that can be used for experimentation. We used the same 15-host topology given for our POX firewall project (Figure 2). This topology allows multiple hosts to exist in the same network and show the effect of the shared resources such as the OpenFlow controller and switch. The topology is used to compare the performance of a simple Pyretic firewall to that of our simple Python firewall running as a POX controller module. The platform used for the analysis was an Ubuntu 13.10 virtual machine running an instance of Mininet to simulate the network topology and Open vSwitch simulating the switch. The Pyretic/POX controller ran on the same virtual machine as Mininet and exposed its control interface on TCP port 6633.

We used standard network analysis tools including tcpdump, nc (netcat), Open vSwitch (OVS) tools, and Python timing output to measure packet arrival times, flow instantiation times, and flow table entries. We compare the performance of the firewall implementations to determine if the compilation of rules in Pyretic has performance benefits. The timing systems relied on the shared clock of the virtual machine, shared by all emulated hosts in Mininet, and the timing output of tcpdump along with calls to the Python sys.time library. We additionally compare the performance of the programs to the difficulty of their implementation, which helps to relate the complexity of the controller language to the performance required for a particular task.

In the interest of brevity, we do not present the code herein, but outline the algorithm used by the POX firewall in Figure 1. In order to compare performance as accurately as

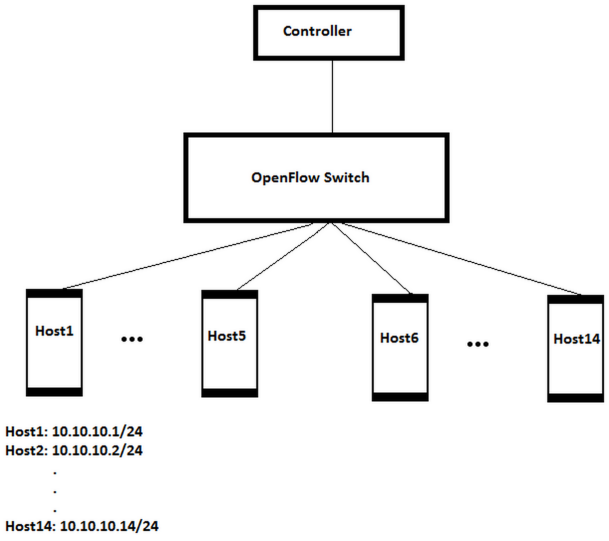

Fig. 2. Large Mininet topology

```
A: 10.0.0.6 any any 6666
B: 10.0.0.7 any any any
C: 10.0.0.8 any 10.0.0.3 5555
D: 10.0.0.9 any 10.0.0.1 5555
E: 10.0.0.10 any 10.0.0.5 any
F: 10.0.0.11 any 10.0.0.4 any
G: any any 10.0.0.2 any
H: 10.0.0.12 any any any
I: any any any 7777
J: 10.0.0.0/24 any any 64000
```

Fig. 3. Configuration file for firewall

possible, we attempted to model our Pyretic firewall after this algorithm, despite the eventual discovery that Pyretic may not be well suited to this particular program flow of control. Our firewall explicitly allows ICMP and ARP traffic to enable debugging and only allows TCP flows that explicitly match the entries enabled by the configuration rules. The configuration file shown in Figure 3 was read in from the command line to represent the filter rules that are proactively installed onto the switch. The alphabetical labels were not included in the configuration file but are here as points of reference. Only packets that match one of these rules were forwarded to the controller for inspection. The controller then installs flow entries to allow the forward and reverse direction of the TCP flow. The first packet of the TCP flow is then returned to the switch so that it may be sent on to its intended destination.

The traffic patterns studied herein consisted of TCP sessions initiated with the netcat utility. Netcat is a powerful utility with simple syntax that can listen for an incoming connection on a server with a command like `nc -l <LISTENING PORT>`. A client system can then connect to this listening server with the command `nc <SERVER IP> -p <SENDING PORT>`. This pair of commands establishes a bidirectional TCP connection with the three-way handshake packets of TCP: SYN, SYN/ACK, ACK. The timing of these three packets can be measured as they are sent or received by the hosts, switch, and controller by instructing tcpdump to listen for TCP traffic on a particular interface with the command `tcpdump -i <INTERFACE> tcp`. The output of tcpdump shows individual packets as they depart and arrive on these interfaces along with timestamps with microsecond precision. These timestamps and the packet flow were analyzed to determine flow initiation and controller timing.

III. RESULTS

We present our analysis of the POX and Pyretic firewall implementations in the following sections in regards to the following metrics: lines of code; size of code in bytes; number of flow entries created for configuration file, allowed flows, and blocked flows; and latency introduced by the controller.

A. Lines of Code

We compared the number of lines of code (LOC) in our Pyretic firewall implementation against our POX firewall implementation by separating lines of code into two categories: lines of code to support debugging during development and necessary lines of code to achieve firewall functionality. The number of lines of code in each category for each implementation along with a percent reduction in lines of code in the Pyretic implementation over the POX implementation is displayed in table 1.

TABLE I. LOC PER CATEGORY PER IMPLEMENTATION AND REDUCTION IN LOC IN PYRETIC OVER POX

Implementation	Debugging code	Necessary code
POX	47	171
Pyretic	32	88
Reduction	31.9%	48.5%

We found that the Pyretic implementation required 31.9% fewer lines of debugging code and 48.5% fewer lines of necessary code. Clearly, the most significant category is necessary lines of code, and our preliminary results may support the claim that the Frenetic language allows programmers to express functionality more succinctly. For example, code in our POX firewall explicitly created a flow entry for each rule in the configuration file proactively and then a pair of symmetric flow entries for the forward and reverse directions of a specific allowed flow. The match() filter of the Pyretic language automatically created a flow entry for the forward direction of an allowed flow, so we did not need to write code to explicitly create this flow entry, but rather only had to specify a match for the reverse direction. This only accounted for a reduction of five lines of code, but it suggests that the match() filter in other places may have contributed to the reduction in necessary lines of code.

B. Code Size

In addition to lines of code, we compared the size in bytes of both firewall implementations in terms of size of firewall source code, size of the compiled firewall code, and the size of the entire code base required to run a controller, as pulled from the git repository for both POX and Pyretic.

We found that both the firewall source code and compiled firewall code for the Pyretic implementation exhibited a similar reduction as we found for lines of code, which is not surprising. What is surprising is that the total code base for Pyretic is less than 20% larger than the POX code base. We note that Pyretic runs on top of POX and we expected a greater increase in size due to the addition of the Pyretic run-time system to POX. This leads us to believe that the Pyretic version of POX was optimized for its use. Upon further inspection, we found that elements of standard POX were not included in the Pyretic version of POX. For example, constants such as OFPP_NORMAL were not defined in the POX version, and in order to use this and other constants, we were forced to modify the Pyretic version of POX for certain tasks.

TABLE II. CODE SIZE PER IMPLEMENTATION AND REDUCTION IN SIZE OF PYRETIC OVER POX IN BYTES

Code Size	Firewall Source Code	Compiled Firewall Code	Code Base
POX	13,858 (13.53KB)	11,343 (10.08KB)	10,176,722 (9.71MB)
Pyretic	7,720 (7.54KB)	6,107 (5.96KB)	12,002,392 (11.45MB)
Reduction	44.3%	46.2%	-18.0%

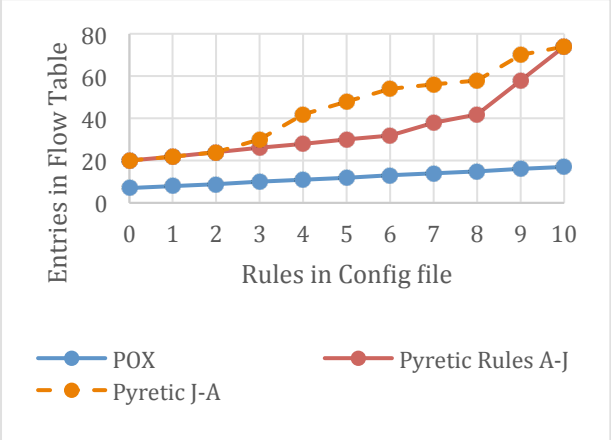

Fig. 4. Flow entries installed proactively before receiving traffic

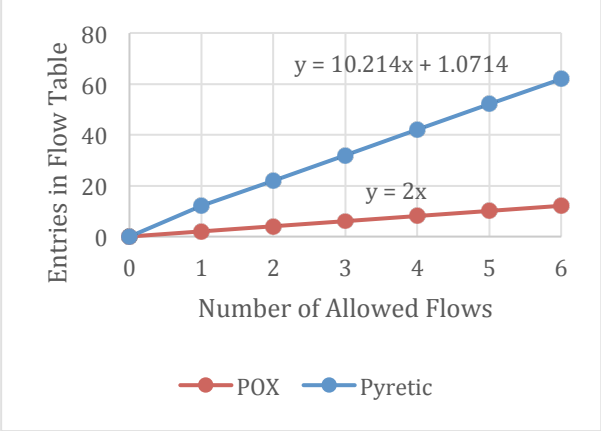

Fig. 5. Flow table entries as a result of allowed TCP connections

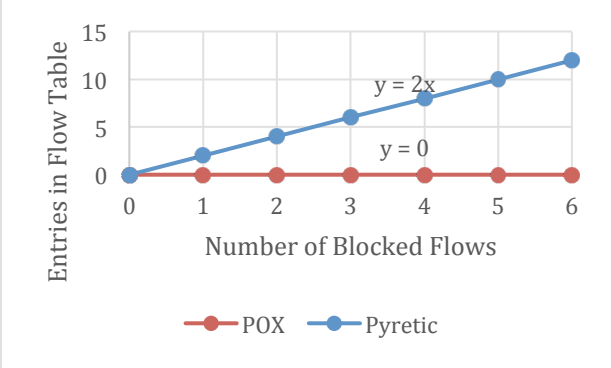

Fig. 6. Flow entries with blocked flows

C. *Number of Flow Entries*

All flow entry experiments were conducted twice and both times gave identical results. Figure 4 shows the increase in flow table entries as additional rules are added to the firewall configuration file (Figure 3). Both times, the rules in the configuration file were added one at a time starting at rule A and continuing to rule J. This process was repeated starting with rule J and continuing to rule A. The data collected shows that there is an interaction between rules as they are aggregated together. The fact that both the reverse and forward ordering of rules resulted in the same number of flow table entries after all rules were added implies that Pyretic uses a deterministic policy compilation process; however, the number of flows entries installed by Pyretic is not easy to predict. Our POX implementation shows a uniform increase of one flow per rule added regardless of order of the rules . Additionally, for the Pyretic firewall, a 17 rule configuration file with fully specified non-overlapping rules was generated that showed a linear increase of two flow entries per rule. Further experimentation is needed to determine the interaction of flows in Pyretic.

Figure 5 shows the increase in flow table entries as traffic is allowed through the firewall. Traffic was generated using netcat to connect to a listening host specified as allowable traffic in our firewall configuration. The specific traffic generated matched the first 6 rules of the configuration file in descending order. Both implementations had a uniform increase in flow table entries for each additional flow with an exception for Pyretic's first allowed flow that generated twelve flow entries, two more than all subsequent traffic. Pyretic shows a greater increase at ten entries added compared to POX two entries added for POX. Figure 6 shows the change in the size of the flow table as the firewall blocks traffic. Traffic was generated using netcat as well, but specified a source host, destination host pair that was not allowed in the firewall configuration. Pyretic adds two additional flows for each initiated flow that is blocked. POX, however, does not need to update the flow table to block traffic, due to the low priority drop rule that covers all disallowed traffic.

The size increase as "a multiplicative 'blow-up'" is discussed in [4], but the authors state that scalability is an issue whenever multiple network management tasks are combined, either manually by a programmer or automatically, as in Pyretic. It is important to note that our results suggest some "blow-up" could occur even in the Pyretic implementation of a fairly simple controller, while the POX

implementation only exhibited a linear growth rate in flow entries. The authors of [5] state "In this slightly more elaborate policy, there are components that look somewhat like OpenFlow rules -- they match different kinds of packets and perform different actions. However, as the simpler flood example shows, these policies do not necessarily map to OpenFlow rules in a one-to-one fashion." This further suggests that the developers of Frenetic are aware that controllers developed using the Frenetic language may create more, and possibly many more, flow entries than if the rules were written at the lower level.

Additionally, the authors of [4] state "If anything, a smart run-time system should do a better job in applying optimizations that minimize the number of rules required to represent a policy." We expected that Pyretic would perform optimization so that redundant flow entries would not be created. For example, the set of flows matching rule "10.0.0.8 8888 10.0.0.9 9999" is a subset of the set of flows matching rule "10.0.0.8 any any any." It was expected that the existence of the second rule would subsume the need for the first rule and so a flow would only be created for the second proactive rule from the configuration file. Instead, both flow entries are created. This suggests that Pyretic does not perform the level of optimization that we expected. Additionally, rules that used a wild card IP address seem to cause a significant increase in the number of flow entries required to cover that flow, while rules with a wild card port did not seem to exhibit the same size increase.

D. *Timing*

Detailed timing analysis was performed on the departure and arrival of times of packets using the Linux tcpdump utility. Timing of the establishment of a single TCP connection between hosts was analyzed in order to determine the time required by the two controllers. Timing data of the sequence of operations is shown in Table 3 and Table 4. In the tables, events corresponding to the SYN or initial synchronization of the flow, also referred to as the forward direction, are in red. Events corresponding to the controller installing flow entries to the switch are shown in green. The blue events correspond to the handling of the SYN/ACK packets for the reverse flow. Here you can see that, under the Pyretic firewall, the first packet of both flow directions (SYN and SYN/ACK, respectively) was resent by their respective hosts. This is because, when Pyretic updates the flow entries on the switch, it re-installs the entire set of flows onto the controller rather than merely adding flow entries corresponding to the new flows.

Additionally, the flow entry updates are sent from the controller to the switch *after* the controller resends the packet received by the query, so no flow entry yet exists on the switch that would allow the packet to be routed to its destination, causing the resent packet to be dropped. Flow entry installation occurring after the controller re-sends the packet suggests that queries are not intended to be used in the creation of new rules, despite query objects inheriting from the DynamicPolicy parent class, a class specifically designed for policies that change over time.

TABLE III. POX_FIREWALL TIMING RESULTS

POX Events	time since t0 (µs)
host7 Tx SYN	0
control Rx SYN	205
control Tx FwdRule	27,092
control Tx RevRule	27,610
control Tx SYN	27,851
Host6 Rx SYN	28,061
Host6 Tx SYN/ACK	28,137
Host7 Rx SYN/ACK	28,202

TABLE IV. FRENETIC_FIREWALL TIMING RESULTS

Frenetic Events	time since t0 (µs)
host7 Tx SYN	0
control Rx SYN	174
control Tx SYN	40,010
control Tx Rules (Start)	191,568
Host7 Tx SYN (dup)	997,909
Host6 Rx SYN	997,936
Host6 Tx SYN/ACK	998,007
control Rx SYN/ACK	998,992
control Tx SYN/ACK	1,034,164
control Tx Rule (End)	1,077,922
Host6 Tx SYN/ACK (dup)	1,999,042
Host7 Rx SYN/ACK	1,999,095

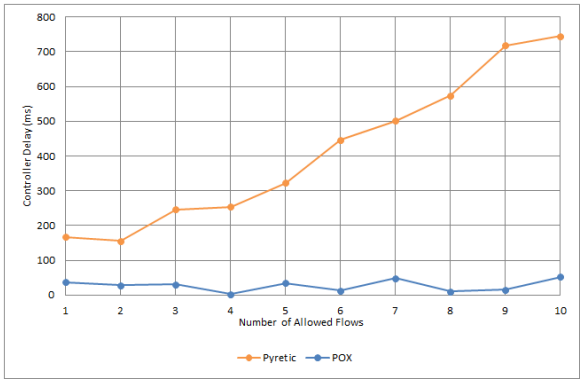

FIGURE 7: CONTROLLER DELAY AS A FUNCTION OF ALLOWED FLOWS

The authors of [4] discuss the need for programs to react to changes in the state of the network in their discussion of querying network state, so we expected that queries were the mechanism within Pyretic that would allow the controller to dynamically adjust the policy in the presence of active flows. However, additional discussion of queries suggests their use may be restricted to monitoring traffic. The correct way to dynamically update policies in Pyretic remains the subject of future work.

Assuming that there exists a better mechanism in Pyretic to dynamically install new flow entries, the controller delay was measured for a consecutive series of active flows for both controllers and is shown in Figure 7. The controller delay is

the time required for the controller to receive an encapsulated packet and then re-send it on to the switch for forwarding ([control Tx time] – [control Rx time]). This delay is representative of the additional latency that the controller would add to the initiation of new TCP flows and is desired to be kept to a minimum. The flows measured in Figure 7 were installed consecutively as flows between a pair of hosts where only the port number of the netcat client changed between consecutive flows. For the POX controller, the controller delay remained relatively constant with a mean of 27.3 ms and standard deviation of 15.9 ms. The Pyretic controller, however, showed a steady increase in the controller delay as more flow entries are added to the switch. Linear regression was performed on the the Pyretic controller delay and was found to have a slope of 70.6 ms/flow and an intercept of 24.5 ms. It was expected that the POX firewall controller module would only have slightly faster performance than that of the Pyretic firewall, but the best-case minimum controller delay of Pyretic, 155 ms, represents a 570% increase over the mean delay of POX at 27.3 ms! The size of this increase is an indication that Pyretic queries in their current implementation are not optimized to be used in reactive flow entry installation as we have done, but are more intended to be used for data collection and analysis.

IV. Conclusion

Despite the expectation that the Frenetic language as implemented in Pyretic would provide a simpler development platform for SDN controllers, significant difficulty was encountered in the creation of a fairly simple whitelist firewall. Our results indicate that the Pyretic implementation of the firewall offers significantly slower dynamic rule installation and will likely offer slower flow entry look up due to the large increase in the number of flow entries over the POX firewall. Pyretic's controller delay over POX indicates that it would not scale well due to its addition of significant latency to the network. The mechanism by which Pyretic creates flow entries is abstracted from the programmer, making it difficult to predict how to express functionality in Pyretic in order to minimize the number of flow entries generated. Due to the attempt at aggregating rules in Pyretic, we were unable to determine a fixed number for the amount of flow table entries required for every rule specified for the firewall configuration; thus, we had to examine numerous flow entries one at a time for every rule added to the firewall, which contributed to difficult debugging the Pyretic firewall.

One benefit of Pyretic seems to be the reduction in necessary lines of code to achieve the same functionality as a firewall written in POX. However, we estimate that our Pyretic implementation took at least 50% more worker hours, so the reduction in lines of code does not correspond to efficiency of implementation. The primary cause of this difficulty was lack of documentation of the Pyretic language. A few examples of source code exist in the Pyretic repository, but it is difficult to understand the semantics of a language from a small selection of examples. The documentation on queries, specifically how to use them and what exactly they do, was particularly lacking and contributed strongly to difficulty in achieving a proactive firewall design, in which the policies are updated dynamically based on packets received by the controller, in Pyretic.

Further difficulties were encountered when expected features of POX were found to be excluded in the reduced Pyretic version of POX. It is unknown whether these features were excluded due to the developers of Pyretic assuming their high-level abstraction eliminated the need for programmers to utilize these lower-level features, if the reduced POX code base contributes to efficiency of the Pyretic run-time system or in analysis of rules for aggregation, or if these features were excluded for some other reason. At minimum, documentation including an overview of what has been removed from the Pyretic version of POX would assist programmers in transitioning from writing controllers for POX to writing controllers in Pyretic.

V. Future Work

For future work, we would like to optimize the performance of the Pyretic firewall and implement additional functionality in the firewall. Some of these additional functions may include automatic statistic reporting from the router to the controller categorized by port or address, additional blacklist or whitelist of traffic based on the contents of the TCP packets, or redirection of traffic to particular servers similar to a content distribution or load balancing scheme. This modification could then be used to measure the impact of extraneous traffic on a server both with and without the additional functionality present in the controller.

Our Pyretic firewall is unable to parse a configuration file that contains CIDR address with a non-zero host. This is due to an error in POX that requires any CIDR address to contain a zero host. This is an issue that exists in both POX and Pyretic. Our POX firewall implemented a solution that zeroed the host for CIDR address. We would like to port this code into our Pyretic implementation but found that it required dependencies that no longer existed in Pyretic's implementation of POX. We would like to implement these dependencies to allow both firewall implementations to have the same conformity in regards to configurations files.

Acknowledgment

The authors thank Professor Syrotiuk for guidance and the anonymous reviewers for helpful comments.

References

[1] N.Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A Network Programming Language," in ACM SIGPLAN International Conference on Functional Programming (ICFP), Tokyo, Japan, September 2011.

[2] N. Mckeown, S. Shenker, T. Anderson, L. Peterson, J. Turner, H. Balakrishnan, and J. Rexford, "OpenFlow: Enabling Innovation in Campus Networks," in ACM SIGCOMM Computer Communication Review, 38(2):69–74, 2008.

[3] B. Nunes, M. Mendonca, X. Nguyen, K. Obraczka, T. Turletti, "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks," in IEEE Communications Surveys and Tutorials in Communications Surveys & Tutorials, IEEE.

[4] N. Foster, A. Guha, M. Reitblatt, A. Story, M. J. Freedman, N. P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker and R.

Harrison, "Languages for Software-Defined Networks," in IEEE Communications Magazine, February 2013.

[5] J. Reich, C. Monsanto, N. Foster, J. Rexford,, and D. Walker, “Modular SDN Programming with Pyretic,” in ;login Magazine, 38(5):128-134, 2013.