Log Parsing in Software-Defined Networking to generate DyNetKAT models

GOOR, J.G.L., University of Twente, The Netherlands

Much like there is a need for validation within software engineering, a growing need is discovered within the networking world. This need comes from the novel research done in the software-defined networking domain, specifically regarding the OpenFlow protocol. There are a few solutions already that allow for reasoning about computer networks, but these solutions require hand-crafting models before one has the ability to reason about them. Therefore, the implementation of a tool that analyses OpenFlow controllers for DyNetKAT model generation is proposed. The symantics and operators of DyNetKAT are discussed as well as the internal workings and the research this tool is built on. Finally, possible future research is uncovered.

Additional Key Words and Phrases: Software-Defined, SDN, OpenFlow, POX, MiniNet, DyNetKAT, NetKAT, Computer Networks

1 INTRODUCTION

1.1 Software-defined networking

Current networking architecture is ill-suited to the growing needs of their users [11]. The Open Networking Foundation (ONF) spearheaded an initiative that standardises a new method for network architecture and maintenance: software-defined networking. In software-defined networking, the data plane is split from the control plane. By contrast, traditional networking architecture has no such separation. This leads to the need to manually configure each networking device before traffic flows in the intended directions with the correct rule-set applied.

For example, a traditional networking switch needs configuration done on the switch to configure which VLAN is assigned to which port. The process of controlling network flow on a device, is the control plane. The traffic flow through the networking device itself, is the data plane. The process of configuring each networking device by hand is tedious and unscalable in bigger networks [11].

Software-defined networking is a term used for the ability to programmatically instruct networking devices of the intended network flow. Therefore, a networking device becomes 'dumb' in the sense that all configuration is done elsewhere. The purpose of the networking device has become to only relay traffic; no longer to configure and relay. The Open Networking Foundation has further specialised this idea into a standardised networking protocol: OpenFlow. OpenFlow works with a controllers and switches. The controllers and switches must be able to communicate with each other.

The process begins with a handshake of sorts. The switch is configured to connect to a controller and tries to do so. The controller then receives a message from the switch with their capabilities. Now, the controller is free to configure the switch through defined OpenFlow control messages.

When traffic is generated behind a switch, the switch sends information about the traffic to the controller. The controller can then decide what to do with the traffic. The basic options come down to sending it to a specific port or sending a flow modification. A flow modification is sent to the switch to signify the allowance or rejection of a certain traffic flow for a specified amount of time. The switch can directly send the traffic to the right outgoing port, without the need of verifying each packet with the controller. A flow modification is sent with a time-to-live (TTL). After the TTL expires, the flow modification is deleted and the switch communicates new packets with the controller. Alternatively, without a flow modification the controller handles each packet of the flow manually and decides on an appropriate action per packet. The well-known implementations of OpenFlow controllers (i.e. Pox [10], Floodlight [4], and ONOS [9]) allow for programming the control flow. With this, the network instrumentation happens via programmable logic. This is arguably easier to debug than traditional networking, because one only has to verify a program instead of configuration that could span multiple network devices.

1.2 DyNetKAT

Within the field of Computer Science, there is a specialisation dedicated to software validation [8]. Software validation can range from writing unit tests, to system tests, to programs trying to mathematically prove the correctness. An example of the latter is VerCors. A tool developed by the University of Twente to analyse and mathematically prove the functionality of parallel programs [13].

A number of solutions have emerged in the software-defined networking domain also. An example of this, is NetKAT [1]. NetKAT is an algebraic language invented to reason specifically about software-defined networks. It does this by providing a framework for reasoning about networks with Kleene algebra with tests, or KAT. NetKAT is built on a set of policies: filter predicates (or tests) and field assignments (or actions). These policies can then be combined in order to make a models of OpenFlow controller applications. The NetKAT framework allows one to reason about the network architecture and its correctness in an equational fashion. This is done through a sound and complete axiomatisation, by proving or disproving certain equalities over policies. The full syntax of NetKAT is described in the original paper [1].

A simple example of a NetKAT program is a policy for forwarding traffic based on simple predicates [1]. The example is as follows:

$$p \triangleq (dst = H_1 \cdot pt \leftarrow 1) + (dst = H_2 \cdot pt \leftarrow 2)$$

Here, we declare a policy p. This policy is the composition of two other policies. The function of the example policy, is to forward a packet to port 1 if the destination is H_1, or to port 2 if the destination is H_2. From the example, a few of the NetKAT operators become clear. The $\triangleq$ is used to define a policy. $a \cdot b$ is used to signify with composition that policy a and b both apply. In the example, $dst =$

TScIT 39, July 7, 2023, Enschede, The Netherlands

$H_1 \cdot pt \leftarrow 1$, it is used to create a test where the destination matches some host (H_1) and then apply an action (sending the packet out of port 1). The $\leftarrow$ operator is used for actions and the + is used for alternate policies (i.e. policy *a* matches *or* policy *b* matches).

The fact that NetKAT is built on Kleene algebra with tests, allows for interesting modeling behaviour. Formally, policies and predicates defined in NetKAT take a history and produces a set of histories. These histories denote the flow of the packet. For example, if a policy returns the empty set ($\emptyset$), the packet is dropped. Alternatively, if the policy outputs one or more histories the packet is sent to one or more destinations respectively. Moreover, because NetKAT is built on top of mathematical logic with predicates, theorems can be derived. NetKAT is built with a few base theorems in mind, from which new theorems can be proven (or disproven). A theorem here, could be the functionality of a computer network that can then be algebraically tested using the NetKAT axioms.

However, one of the main problems with NetKAT is that it is stateless and does not allow for modeling concurrency within networks [2]. It is not possible to model a network where state changes influence the packet flow. Moreover, NetKAT does not allow for dynamic updates to the flow tables in a software-defined network (SDN). To solve these problems, a dynamic extension to NetKAT is proposed and named DyNetKAT. DyNetKAT retains the robust core of NetKAT while adding new operators. These new operators, semantics, and proofs can be found in the original paper [2].

Because DyNetKAT allows for the modeling of concurrency in networks, a new construct is developed. This is called the **dup** construct. It allows one to inspect the intermediate history of a packet, during incomplete application of flow tables in the network. Incomplete application of flow modifications can happen in the network, due to the concurrent nature of the controller (i.e. multiple switches requiring flow modifications, some flow modifications may reach one switch quicker than another).

DyNetKAT can be used to reason about concurrent SDNs with stateful behaviours. Once a model of a network is created in DyNetKAT, one can reason about the states in the network with the use of a labelled transition system (LTS) behavioural model. Using such an LTS, one can verify the network for improper states and reason where the errors come from. Therefore, DyNetKAT can be used to verify computer network correctness, much like a system test in the software validation domain. The verification here is that improper network states (e.g. wrong firewall configuration or unexpected switching behaviour) can be detected and removed.

1.3 Generating DyNetKAT models

A prototype tool for extracting DyNetKAT models from SDN logging is currently under development at University of Twente. This tool requires logs generated by an OpenFlow controller and then generates a formal DyNetKAT model describing the network. This tool requires the input logs to be in a specific format, akin to the log format used in SDNRacer [6]. SDNRacer analyses SDN log files to reason about race conditions in SDNs. The current paper builds on top of the aforementioned prototype tool.

2 MOTIVATION

There is currently a gap between writing OpenFlow compatible controller applications and the ability to reason about the validity of the resulting network. The current process of generating models for network validation is labour-intensive and time consuming. This is because the models need to be hand-crafted before thorough analysis can take place. There is currently no tool that automatically generates DyNetKAT models based on real-world SDN controllers.

Because no software exists that bridges the gap between writing OpenFlow-compatible controller applications and the ability to reason about the resulting validity of these networks, this research was authored. Using the tool described in this paper allows one to write a simulation and then use the OpenFlow-compatible controller to generate logs in the SDNRacer format. These logs are then further processed to generate DyNetKAT models. Finally, these models can be reasoned about by using a labelled transition system derived from the DyNetKAT model.

3 RUNNING EXAMPLE

A simple running example used in the DyNetKAT paper [2], is a stateful firewall that requires a secure connection before traffic flow is allowed to happen. This firewall is modeled as follows:

$$
\begin{aligned}
Host \triangleq\ & secConReq!\mathbf{1}; Host \oplus \\
& secConEnd!\mathbf{1}; Host \\
Switch \triangleq\ & ((port = int) \cdot (port \leftarrow ext)); Switch \oplus \\
& ((port = ext) \cdot \mathbf{0}); Switch \oplus \\
& secConReq?\mathbf{1}; Switch' \\
Switch' \triangleq\ & ((port = int) \cdot (port \leftarrow ext)); Switch' \oplus \\
& ((port = ext) \cdot (port \leftarrow int)); Switch' \oplus \\
& secConEnd?\mathbf{1}; Switch \\
Init \triangleq\ & Host || Switch
\end{aligned}
\tag{1}
$$

This example shows new operators defined by the DyNetKAT extension over NetKAT. In order for an intuitive sense to be gained for DyNetKAT and its semantics, this example and the DyNetKAT operators used will be thoroughly explained. Firstly, we have $N; D$. This means the processing of policy N is finished and the next packet is processed according to policy D. In this example, it is used to recursively process packets over the *Host* policy. Additionally, it is used for the modeling of the dynamic behaviour of the switch according to the secure connection. The next operator used is $\oplus$. This denotes a non-deterministic policy choice. In this example it is used to model the different behaviours that can occur in the network for both *Host* and *Switch*. The final extension used in this example are the $x?N; D$ and $x!N; D$ operators. The first part of these are the $x?N$ and $x!N$ operators. These denote sending and receiving policies through channel x. In other words, this means the network configuration is updated with policy N. Then, the second part $; D$ is used when the network configuration has been updated. After the update, the next packet is processed according to policy D. In the stateful firewall example, the $x?N; D$ and $x!N; D$ operators are used to update the network configuration based on whether there is an open secure connection. The **1** and **0** signify the state of the

secure connection, i.e. *secConReq*!**1** signifies the start of a secure connection and *secConReq*?**1** asserts whether there is a request to open a secure connection. Finally, the rule $Init \triangleq Host||Switch$ allows for parallel composition of the two policies *Host* and *Switch*. This denotes that both *Host* and *Switch* process packets at the same time.

The result of the stateful firewall model is simple. The host either sends a secConReq, requesting a secure connection, or a secConEnd, indicating a secure connection will be closed. Then, the switch is modeled to allow outgoing traffic and drop incoming traffic. However, if the switch receives a secure connection request, it starts operating differently. After the secure connection request, the switch now allows incoming traffic to flow to the host until the secure connection is closed. After closing, the switch goes back to the original behaviour.

This model is extensively worked out in the DyNetKAT paper. Therefore, this is the example used during the development of this tool. Because there is already a hand-crafted model of this running example in the DyNetKAT paper, verification of the output of this tool is made trivial.

4 METHODOLOGY

4.1 Approach

In order to build this tool, extensive research into software-defined networking was done. Then, the design of the tool could be split into multiple parts. The parts are as follows:

(1) Build an OpenFlow-compatible controller implementation of the stateful firewall described in the DyNetKAT paper.
(2) Design and architect a tool which takes the controller implementation, outputs the right logging format, and generates a DyNetKAT model.
(3) Use the controller implementation of the stateful firewall example to generate a DyNetKAT model for it.
(4) Validate the generated model against the hand-crafted model.

Specific reasons for certain design decisions are discussed in the corresponding sections below.

4.1.1 Building an OpenFlow-compatible controller implementation of the stateful firewall. Building an OpenFlow-compatible controller implementation of a stateful firewall first requires a choice for the controller software. For this research, Pox was chosen [10]. Because software-defined networking is usually done with large-scale networks, a network simulation tool is needed also. For this purpose, MiniNet was chosen [7]. Both Pox and MiniNet are pieces of software widely used in the SDN research domain. They are well-documented and can be easily integrated with one another.

The DyNetKAT paper discusses a host which sends a secure connection request and which can end a secure connection. Since no specific packet predicates are included in the example, a choice was made to model the secure connection with TCP connections to specific ports. Thus, the implementation is as follows. If a host in the network connects to an IP address on a certain TCP port, a secure connection opens. To end a secure connection, the host must connect to a different TCP port with the same destination IP address.

This was specifically chosen as this mimics behaviour to the hole-punching technique [5]. With hole-punching, hosts connect to an unrestricted host that then allows information exchange to build a session. In this case, the 'unrestricted host' is the OpenFlow controller noticing an attempt for session initiation (i.e. the opening of a secure connection).

4.1.2 Design of the tool. The main problem identified in the design of a tool that takes an OpenFlow controller and outputs a DyNetKAT model with the purpose of network validation, is how to run the controller as-is in a way where logging output is consistent and the controller function remains the same. After literature analysis on the required logging format, a solution was found. A software stack that supports the three major OpenFlow controller frameworks (i.e. Pox, Floodlight, and ONOS) and outputs logging in the format required by the DyNetKAT model generator tool.

This solution is named STS [12], or the SDN Troubleshooting System. STS allows one to run their controller, along with a troubleshooting system. It runs a pre-configured simulation against the controller and generates the required logging format used in the generation of DyNetKAT models.

4.2 Use of MiniNet

In the early stages of familiarising with the inner workings of OpenFlow, MiniNet was used. This is because MiniNet allows for complex network simulations that run on top of OpenFlow. Within MiniNet, one can then analyse how and which traffic flows. This is an essential tool for verifying the workings of the in-development controller.

Throughout this research, MiniNet is always connected to a 'remote' controller running on the same machine. This means MiniNet does not use an internally simulated OpenFlow controller. Rather, it uses the Pox controller already running on the same machine. Even when running interactive simulations with STS, MiniNet is ran alongside to assist the understanding of traffic flow.

MiniNet has proved to be an invaluable diagnostic tool with this research.

5 RESULTS

The final tool can be found on the University of Twente GitLab environment. The link to the tool is
https://gitlab.utwente.nl/s2311720/sdn2dynetkat.

5.1 Requirements and execution

The requirements for running the tool and generating a DyNetKAT model of a network are:

(1) There must be a controller implementation in Pox, Floodlight or ONOS.
(2) A simulation has to be written as configuration for STS.

Once these requirements are fulfilled, the tool can successfully be used to generate a DyNetKAT model. The final tool can be run in Docker according to the repository instructions.

5.2 High level description of the workings

The first steps of the tool are software installation and configuration. The tool then takes the simulation details configured by the user

and runs the user-made OpenFlow controller. After the simulation has ended, the logs are collected in a specific location. Then, the tool converts the generated log to a DyNetKAT model. This DyNetKAT model is then saved and returned to the user.

5.3 The tool in detail

Within Pox, one can subscribe to events. The two events most relevant to this tool are the *ConnectionUp* and *PacketIn* events. The full source code for the Pox controller implementation can be found in appendix A. The stateful firewall implementation runs an instance of the class *FirewallController* which listens for incoming OpenFlow connections (i.e. switches connecting to the controller). Then, a special object in the controller is created to manage the switch. This is an instance of the *FirewalledLearningSwitch*. The *FirewalledLearningSwitch* learns which MAC addresses are behind which ports and handles the traffic flow on the switch itself. The ARP protocol is always allowed through. Other traffic is blocked, unless a secure connection is opened between hosts. A secure connection is opened by connecting to another host on TCP port 6000. The resulting connection can be closed by connecting to the same host on TCP port 7000.

Additionally, the learned MAC table is cleared every few seconds to not keep stale data. As soon as a traffic flow is allowed through with a secure connection, a flow modification is made. In this way the controller does not need to check all packets coming through, but can grant an accept for a few seconds at a time.

With the Pox controller implementation ready, the next step is to run an STS experiment. In order to run such an experiment, first the experiment details must be configured in an STS configuration. The full STS experiment configuration for the interactive control flow can be found in appendix A. The interactive control flow allows for manual injection of traffic in the simulation. This manual injection is done through a command-line interface. The STS experiment configuration seems trivial, as the code is small. However, after defining the trivial parts of the experiment (e.g. how to start the controller, which topology to use and where to store the results), one has to choose a control flow. The control flow determines how STS examines the controller. There are a few built-in solutions to STS, such as a Fuzzer and an Interactive controller. One can also 'play-back' other STS experiments with a certain control flow.

The main issue here, though, is that for complex host interactions there is no built-in control flow that suits well to advanced scenarios. In the case of the stateful firewall, there is no control flow that allows for opening and closing a secure connection. Therefore, generating results from advanced experiments in STS is non-trivial. It requires one to write a custom control flow. Due to time constraints with this research, no such specialised control flow could be written.

As can be viewed in the STS controller configuration, there are options called *hb_logger_class* and *hb_logger_params*. These are of great importance to this tool. Using the *HappensBeforeLogger*, or HB logger, one does not need to adapt the OpenFlow controller implementation to generate custom logs. The HB logger is injected into the simulation and generates the output required by the tool that generates the resulting DyNetKAT model for a simulation. An example of running an STS experiment with the *Fuzzer* control flow with the stateful firewall controller, is given in appendix B.

As can be viewed from the output, the HB logger generates a JSON log with the events that happen within the simulation. The explanation of this format is beyond the scope of this research, but can be found in the SDNRacer paper [6]. The most important events logged by the HB logger are the *HbMessageHandle* and *HbAsyncFlowExpiry*. These are the main messages used by the DyNetKAT model generator.

The DyNetKAT model generation tool takes the HB logger ouput and generates a DyNetKAT model in JSON format. An example of the output of the DyNetKAT model generator can be viewed in appendix B. This example is from a hand-crafted controller that generates a smaller output for easier viewing.

6 CONCLUSION

In conclusion, this paper outlines the developments in software-defined networking and the importance of network validation. With the introduction of software in networking, comes a revolution in the approach towards network management and control. However, the introduction of programmability also necessitates the validation of the underlying software that now drives these networks. In other words, there is an inherent requirement for verification of network correctness within the domain of software-defined networking.

While existing domain-specific languages already allow for model generation, and thus network validation, the process of creating these models is meticulous and time-intensive. This is because the models currently need to be hand-crafted. This labour-intensive process poses a significant challenge for network engineers and hinders efficient development.

To address this challenge, this paper introduces a novel tool which, when implemented, streamlines the process of network validation by eliminating the need for manual model generation. The models can now be generated by using the already written software to drive the network along with a simulation configuration. The generated models can then undergo thorough network analysis to ensure correctness.

7 DISCUSSION

This section highlights some of the problems encountered during the research and possible flaws with the methodology.

7.1 Initial design problems

During the development of this tool, many problems were encountered. The first of which is the complexity of the source code of the dependencies used. Pox is noticeably a research-oriented tool. There is limited source code documentation and examples on how to produce a real-world SDN controller application are lacking. With some experimentation using Pox and MiniNet to see how the traffic flows, a controller implementation was built.

Another problem not anticipated is the handling of ARP packets throughout the network. Address Resolution Protocol (ARP) is a protocol used to locate MAC addresses by IP addresses. In the OSI networking model, the different network layers are stacked [3]. This is because higher layers depend on lower layers. For example, if a

host sends an ICMP Echo Request packet to another, it must first find the MAC address to send the packet to. The resolution of IP addresses to MAC addresses, is what the ARP protocol is used for.

However, the proper handling of ARP packets without prior experience in software-defined networking is non-trivial. ARP packets are usually handled by low-level mechanisms in most networking stacks and one often has no requirement to manually handle them. The solution to disallowing some packets without a secure connection, but still allowing traffic to flow as soon as a secure connection has been made, is to always handle and forward ARP packets throughout the local network. The local network for this paper, is the communication between host, switch and external host.

7.2 Software dependencies

The current tool requires a number of software components. Because of the dependencies between these different components, there is a struggle when single components receive updates. For example, STS depends on Pox. When Pox receives an update, STS must first be adapted in order for the Pox updates to become usable. Moreover, there is a difference in required Python version between STS and Pox. Pox runs on a newer version of Python (version 3), while STS remains on an outdated version (python 2.7). This makes the integration between these tools complex and debugging even more so.

In my opinion, the best solution here is to reduce the software dependency requirement and develop a tool which unites the functions of the currently separate components. However, this is an enormous task and labour-intensive.

7.3 Maintainability

Because of the use of multiple different components in the open-source community, there is a risk of projects becoming unmaintained. It seems this is already the case for a few of the widely used SDN controllers: Pox and Floodlight. Moreover, STS is a tool developed specifically for the SDNRacer paper, not a tool born from real-world necessity. Due to the lack of maintainers, STS is also no longer under active development.

Because these tools are built recently and software-defined networking is a novel domain, this is no issue for this tool currently. However, as time passes and software components evolve, this tool will also become unusable. The ONF is still working on new drafts for the OpenFlow specification at the time of writing. The current tool can only support OpenFlow 1.0 because of component dependencies and the complex nature of the project. However, OpenFlow 1.0 will not always stay relevant.

7.4 The simulation configuration requirement

This tool works by simulating the network and building a DyNetKAT model based on a simulation given. However, this poses a few challenges. This means the correctness of the network is only as good as the given simulation. It could be the case that a simulation is too simple to register any faults with the network, which could be later discovered when the network is operational. One could argue here that writing unit tests has the same flaw. A unit test is only as good as the strictness of the written test and the original code may contain issues despite passing the test.

Simulations in the SDN Troubleshooting System have to be written by hand. However, at the time of writing the website where STS is explained in detail is not accessible. Moreover, the documentation in the published repository on GitHub and within the code itself is lacking. Therefore, writing a simulation with STS is a complex and arduous procedure that takes quite a time investment. Especially the requirement for custom control flows to simulate complicated network functions.

Due to the complicated nature of STS simulation control flows combined with time constraint, stateful experiments could not be included in this paper. However, stateless experiments have been executed by adapting the 'Fuzzer' configuration. Due to the irrelevant nature of stateless examples compared with the running example, these are not included in this paper.

The solution here could be hidden in the software validation world. Static code analysis, or the analysis of the source code itself, could also be a solution in the software-defined networking domain. It could be useful to perform analysis on the source code of the controllers and perform the checks for validation there.

Another issue with the simulation configuration requirement, is that it is complex to write as STS contains many undocumented parts. There are few high level description of components within STS and documentation of code components is also lacking. Therefore, the use of the tool described in the paper is complex due to the simulation configuration requirement.

8 FUTURE WORK

With this tool, further research can be done into network validation. A possible next step could be to automatically validate the LTS that can be made using the DyNetKAT model. There is also an automation opportunity for automatically generating an LTS based on the DyNetKAT model.

Much like the developments in the software validation domain, perhaps it could be useful to look into static code analysis of controllers within software-defined networks. This could provide a novel field within software-defined network validation and pave the way for wide adoption.

ACKNOWLEDGMENTS

I would like to thank Georgiana Caltais for supervising this project. She has been a great help for the duration of the process and I value our collaboration. Additionally, I would like to thank Can Olmezoglu for his assistance with the model generation and for being there as general support. Finally, I would like to thank Peter Lammich for chairing the Software Technology and Formal Methods track.

REFERENCES

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: semantic foundations for networks. en. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, San Diego California USA, (Jan. 2014), 113–126. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535862.

[2] Georgiana Caltais, Hossein Hojjat, Mohammad Mousavi, and Hunkar Can Tunc. 2021. DyNetKAT: An Algebra of Dynamic Networks. arXiv:2102.10035 [cs]. (May 2021). DOI: 10.48550/arXiv.2102.10035.

[3] J.D. Day and H. Zimmermann. 1983. The OSI reference model. *Proceedings of the IEEE*, 71, 12, (Dec. 1983), 1334–1340. Conference Name: Proceedings of the IEEE. DOI: 10.1109/PROC.1983.12775.

[4] [SW], Floodlight OpenFlow Controller (OSS) June 20, 2023. URL: https://github.com/floodlight/floodlightRetrieved June 20, 2023 from.

[5] Bryan Ford, Pyda Srisuresh, and Dan Kegel. 2006. Peer-to-peer communication across network address translators. (Mar. 18, 2006). arXiv: cs/0603074. DOI: 10.48550/arXiv.cs/0603074.

[6] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. 2016. SDNRacer: concurrency analysis for software-defined networks. en. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Santa Barbara CA USA, (June 2016), 402–415. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908124.

[7] [SW], Mininet: Rapid Prototyping for Software Defined Networks June 21, 2023. URL: https://github.com/mininet/mininetRetrieved June 21, 2023 from.

[8] Atica Mohammed, Rasha Alsarraj, and Asmaa Albayati. 2020. VERIFICATION AND VALIDATION OF a SOFTWARE: a REVIEW OF THE LITERATURE. *Iraqi Journal for Computers and Informatics*, 46, (June 30, 2020), 40–47. DOI: 10.25195/ijci.v46i1.249.

[9] [SW], ONOS : Open Network Operating System June 16, 2023. URL: https://github.com/opennetworkinglab/onosRetrieved June 20, 2023 from.

[10] [SW] N. O. X. Repo, POX June 20, 2023. URL: https://github.com/noxrepo/pox Retrieved June 20, 2023 from.

[11] Timon Sloane. 2013. Software-defined networking: the new norm for networks. Open Networking Foundation. (May 2, 2013). Retrieved June 20, 2023 from https://opennetworking.org/sdn-resources/whitepapers/software-defined-networking-the-new-norm-for-networks/.

[12] [SW], ucb-sts/sts Nov. 16, 2021. URL: https://github.com/ucb-sts/stsRetrieved June 20, 2023 from.

[13] [SW], VerCors Verification Toolset June 21, 2023. URL: https://github.com/utwente-fmt/vercorsRetrieved June 21, 2023 from.

A SOURCE CODE

A.1 POX Stateful Firewall implementation

```
import pox.openflow.libopenflow_01 as of
import pox.lib.packet as pkt

from pox.core import core
from pox.lib.revent import EventMixin
from pox.openflow import PacketIn, ConnectionUp

log = core.getLogger()


class FirewalledLearningSwitch(EventMixin):
    def __init__(self, connection):
        self.mac_table = {}
        self.open_secure_conns = {}

        self.connection = connection
        self.connection.addListeners(self)

        self.idle_timeout = 10
        self.hard_timeout = 30
        self.mac_clear_timeout = 60

    def clear_mac_table(self):
        self.mac_table.clear()

    def schedule_clear_mac_table(self):
        self.clear_mac_table()
        core.callDelayed(self.mac_clear_timeout, self.schedule_clear_mac_table)

    def _handle_PacketIn(self, event):
        packet = event.parsed

        def flood():
            msg = of.ofp_packet_out()
            msg.actions.append(of.ofp_action_output(port=of.OFPP_FLOOD))
            msg.data = event.data
            msg.in_port = event.port
            self.connection.send(msg)

        def drop():
            msg = of.ofp_flow_mod()
            msg.match = of.ofp_match.from_packet(packet)
            msg.idle_timeout = self.idle_timeout
            msg.hard_timeout = self.hard_timeout
            msg.buffer_id = event.ofp.buffer_id
            self.connection.send(msg)

        def accept():
```

```
            msg = of.ofp_flow_mod()
            msg.match = of.ofp_match.from_packet(packet)
            msg.idle_timeout = self.idle_timeout
            msg.hard_timeout = self.hard_timeout
            msg.buffer_id = event.ofp.buffer_id
            msg.data = event.ofp

            if packet.dst not in self.mac_table:
                flood()
                return

            port = self.mac_table[packet.dst]
            if port == event.port:
                drop()

            msg.actions.append(of.ofp_action_output(port=port))

        # Always accept ARP.
        if packet.type == packet.ARP_TYPE:
            flood()
            return

        # Create a match object so we can extract details from the packet.
        match = of.ofp_match.from_packet(packet)
        self.open_secure_conns.setdefault(packet.src, [])
        self.open_secure_conns.setdefault(packet.dst, [])

        # Check if this is an IP packet with TCP
        if match.dl_type == 0x800 and match.nw_proto == 0x6:
            # Port 6000 opens the secure connection.
            if match.tp_dst == 6000:
                self.open_secure_conns[packet.src].append(packet.dst)
                self.open_secure_conns[packet.dst].append(packet.src)
                log.info("%s opened a secure connection." % (packet.src))
            # 7000 closes it.
            elif match.tp_dst == 7000:
                self.open_secure_conns[packet.src].remove(packet.dst)
                self.open_secure_conns[packet.dst].remove(packet.src)
                log.info("%s closed the secure connection." % (packet.dst))

        # Check if the destination is an allowed destination.
        allowed_destinations = self.open_secure_conns[packet.src]
        if packet.dst in allowed_destinations:
            accept()
        else:
            drop()


class FirewallController(EventMixin):
    def __init__(self, transparent):
        self.listenTo(core.openflow)
```

```
        self.transparent = transparent
        core.openflow.addListeners(self, "all")

    def _handle_ConnectionUp(self, event):
        # Delete all mods
        event.connection.send(of.ofp_flow_mod(command=of.OFPFC_DELETE))

        # Create a learning firewalled switch from the connection.
        FirewalledLearningSwitch(event.connection)


def launch(transparent=False):
    core.registerNew(FirewallController, transparent)
```

A.2 STS experiment configuration

```
from config.experiment_config_lib import ControllerConfig
from sts.topology import StarTopology
from sts.control_flow.interactive import Interactive
from sts.input_traces.input_logger import InputLogger
from sts.simulation_state import SimulationConfig
from sts.happensbefore.hb_logger import HappensBeforeLogger

# Use POX as our controller
start_cmd = ('''./pox.py '''
             '''openflow.of_01 --address=__address__ --port=__port__ '''
             '''eyedevelop.stateful_firewall''')
controllers = [ControllerConfig(start_cmd, cwd="pox/", address="0.0.0.0")]

topology_class = StarTopology
topology_params = "num_hosts=2"

results_dir = "experiments/eyedevelop_stateful_firewall"

simulation_config = SimulationConfig(controller_configs=controllers,
                                     topology_class=topology_class,
                                     topology_params=topology_params,
                                     hb_logger_class=HappensBeforeLogger,
                                     hb_logger_params=results_dir)

control_flow = Interactive(simulation_config,
                           input_logger=InputLogger())
```

B GENERATED FILES

B.1 HappensBefore logging output


```
{"eid": 1, "type": "HbMessageSend", "mid_in": 1, "mid_out": [2], "msg_type": "OFPT_HELLO", "
    dpid": 1, "controller_id": ["127.0.0.1", 6633], "msg": "AQAACAAAAAE="}
{"eid": 4, "type": "HbMessageHandle", "pid_in": null, "pid_out": [], "mid_in": 3, "mid_out":
    [], "msg_type": "OFPT_HELLO", "operations": ["{\"eid\": 5, \"type\": \"TraceSwitchNoOp\",
    \"dpid\": 1, \"t\": \"1688208880.312565\"}"], "dpid": 1, "controller_id": ["127.0.0.1",
    6633], "packet": null, "in_port": "None", "msg": "AQAACAAAAAE="}
```

```
{"eid": 10, "type": "HbMessageHandle", "pid_in": null, "pid_out": [], "mid_in": 4, "mid_out":
    [5], "msg_type": "OFPT_FEATURES_REQUEST", "operations": ["{\"eid\": 11, \"type\": \"
    TraceSwitchNoOp\", \"dpid\": 1, \"t\": \"1688208880.41347\"}"], "dpid": 1, "controller_id
    ": ["127.0.0.1", 6633], "packet": null, "in_port": "None", "msg": "AQUACAAAAAM="}
{"eid": 13, "type": "HbMessageSend", "mid_in": 5, "mid_out": [6], "msg_type": "
    OFPT_FEATURES_REPLY", "dpid": 1, "controller_id": ["127.0.0.1", 6633], "msg": "
    AQYAgAAAAAMAAAAAAAAAAQAAAGQBAAAAAAAA7wAAB/8
    AAQAAAAABAWV0aDEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAgAAAABAmV0aDIAAAAAAAAAAAAAAAAAAAAAA
    ="}
{"eid": 20, "type": "HbMessageHandle", "pid_in": null, "pid_out": [], "mid_in": 7, "mid_out":
    [], "msg_type": "OFPT_SET_CONFIG", "operations": ["{\"eid\": 21, \"type\": \"
    TraceSwitchNoOp\", \"dpid\": 1, \"t\": \"1688208880.514586\"}"], "dpid": 1, "controller_id
    ": ["127.0.0.1", 6633], "packet": null, "in_port": "None", "msg": "AQkADAAAAAUAAACA"}
{"eid": 24, "type": "HbMessageHandle", "pid_in": null, "pid_out": [], "mid_in": 8, "mid_out":
    [], "msg_type": "OFPT_FLOW_MOD", "operations": ["{\"eid\": 26, \"type\": \"
    TraceSwitchFlowTableWrite\", \"dpid\": 1, \"flow_table\": [], \"flow_mod\": \"
    AQ4ASAAAAAcAEAAfAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAwAAAACAAP
    ///////wAA\", \"t\": \"1688208880.51521\"}"], "dpid": 1, "controller_id": ["127.0.0.1",
    6633], "packet": null, "in_port": "None", "msg": "
    AQ4ASAAAAAcAEAAfAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAwAAAACAAP
    ///////wAA"}
{"eid": 29, "type": "HbMessageHandle", "pid_in": null, "pid_out": [], "mid_in": 9, "mid_out":
    [10], "msg_type": "OFPT_BARRIER_REQUEST", "operations": ["{\"eid\": 30, \"type\": \"
    TraceSwitchBarrier\", \"dpid\": 1, \"t\": \"1688208880.515531\"}"], "dpid": 1, "
    controller_id": ["127.0.0.1", 6633], "packet": null, "in_port": "None", "msg": "
    ARIACAAAAAk="}
{"eid": 32, "type": "HbMessageSend", "mid_in": 10, "mid_out": [11], "msg_type": "
    OFPT_BARRIER_REPLY", "dpid": 1, "controller_id": ["127.0.0.1", 6633], "msg": "ARMACAAAAAk
    ="}
{"eid": 35, "type": "HbControllerHandle", "mid_in": 11, "mid_out": [12]}
{"eid": 40, "type": "HbControllerSend", "mid_in": 12, "mid_out": [13]}
{"eid": 38, "type": "HbMessageHandle", "pid_in": null, "pid_out": [], "mid_in": 13, "mid_out":
     [], "msg_type": "OFPT_FLOW_MOD", "operations": ["{\"eid\": 41, \"type\": \"
    TraceSwitchFlowTableWrite\", \"dpid\": 1, \"flow_table\": [], \"flow_mod\": \"
    AQ4ASAAAAAsAEAAfAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAwAAAACAAP
    ///////wAA\", \"t\": \"1688208880.617202\"}"], "dpid": 1, "controller_id": ["127.0.0.1",
    6633], "packet": null, "in_port": "None", "msg": "
    AQ4ASAAAAAsAEAAfAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAwAAAACAAP
    ///////wAA"}
{"eid": 44, "type": "HbHostSend", "pid_in": 14, "pid_out": [15], "hid": 1, "packet": "
    EjRWeAECEjRWeAEBCABFAAAcBfAAAEABe/h7ewEBe3sBAggArFNLrAAA", "out_port":
    "12:34:56:78:01:01"}
{"eid": 46, "type": "HbPacketHandle", "pid_in": 15, "pid_out": [16], "mid_out": [17], "
    operations": ["{\"eid\": 49, \"type\": \"TraceSwitchFlowTableRead\", \"dpid\": 1, \"packet
    \": \"EjRWeAECEjRWeAEBCABFAAAcBfAAAEABe/h7ewEBe3sBAggArFNLrAAA\", \"in_port\": 1, \"
    flow_table\": [], \"flow_mod\": null, \"touched_flow_bytes\": null, \"t\":
    \"1688208884.234906\"}", "{\"eid\": 50, \"type\": \"TraceSwitchBufferPut\", \"dpid\": 1,
    \"packet\": \"EjRWeAECEjRWeAEBCABFAAAcBfAAAEABe/h7ewEBe3sBAggArFNLrAAA\", \"in_port\": 1,
    \"buffer_id\": 1, \"t\": \"1688208884.234935\"}"], "dpid": 1, "packet": "
    EjRWeAECEjRWeAEBCABFAAAcBfAAAEABe/h7ewEBe3sBAggArFNLrAAA", "in_port": 1}
```

```
{"eid": 52, "type": "HbMessageSend", "mid_in": 17, "mid_out": [18], "msg_type": "
    OFPT_PACKET_IN", "dpid": 1, "controller_id": ["127.0.0.1", 6633], "msg": "
    AQoAPAAAAAEAAAABADwAAQAAEjRWeAECEjRWeAEBCABFAAAcBfAAAEABe/h7ewEBe3sBAggArFNLrAAA"}
{"eid": 55, "type": "HbControllerHandle", "mid_in": 18, "mid_out": [19]}
{"eid": 60, "type": "HbControllerSend", "mid_in": 19, "mid_out": [20]}
{"eid": 58, "type": "HbMessageHandle", "pid_in": 16, "pid_out": [], "mid_in": 20, "mid_out":
    [], "msg_type": "OFPT_FLOW_MOD", "operations": ["{\"eid\": 61, \"type\": \"
    TraceSwitchFlowTableWrite\", \"dpid\": 1, \"flow_table\": [], \"flow_mod\": \"
    AQ4ASAAAAA0AAAABAAASNFZ4AQESNFZ4AQL//
    wAACAAAAQAAe3sBAXt7AQIACAAAAAAAAAAAAAAAAAAAKAB6AAAAAAAH//wAA\", \"t\":
    \"1688208884.336893\"}", "{\"eid\": 62, \"type\": \"TraceSwitchBufferGet\", \"dpid\": 1,
    \"packet\": \"EjRWeAECEjRWeAEBCABFAAAcBfAAAEABe/h7ewEBe3sBAggArFNLrAAA\", \"in_port\": 1,
    \"buffer_id\": 1, \"t\": \"1688208884.337028\"}", "{\"eid\": 63, \"type\": \"
    TraceSwitchPacketDrop\", \"dpid\": 1, \"packet\": \"EjRWeAECEjRWeAEBCABFAAAcBfAAAEABe/
    h7ewEBe3sBAggArFNLrAAA\", \"in_port\": 1, \"flow_table\": [\"
    AQ4ASAAAAAMAAAABAAASNFZ4AQESNFZ4AQL//
    wAACAAAAQAAe3sBAXt7AQIACAAAAAAAAAAAAAAAAAAAKAB6AAAAAAAH//wAA\"], \"t\":
    \"1688208884.337051\"}"], "dpid": 1, "controller_id": ["127.0.0.1", 6633], "packet": "
    EjRWeAECEjRWeAEBCABFAAAcBfAAAEABe/h7ewEBe3sBAggArFNLrAAA", "in_port": 1, "msg": "
    AQ4ASAAAAA0AAAABAAASNFZ4AQESNFZ4AQL//
    wAACAAAAQAAe3sBAXt7AQIACAAAAAAAAAAAAAAAAAAAKAB6AAAAAAAH//wAA"}
{"eid": 66, "type": "HbHostSend", "pid_in": 21, "pid_out": [22], "hid": 1, "packet": "
    EjRWeAECEjRWeAEBCABFAAAcBfIAAEABe/Z7ewEBe3sBAggAKnXNiQAB", "out_port":
    "12:34:56:78:01:01"}
{"eid": 72, "type": "HbAsyncFlowExpiry", "mid_in": null, "mid_out": [], "operations": ["{\"eid
    \": 73, \"type\": \"TraceSwitchFlowTableEntryExpiry\", \"dpid\": 1, \"flow_table\": [\"
    AQ4ASAAAAAUAAAABAAASNFZ4AQESNFZ4AQL//
    wAACAAAAQAAe3sBAXt7AQIACAAAAAAAAAAAAAAAAAAAKAB6AAAAAAAH//wAA\"], \"flow_mod\": \"
    AQ4ASAAAAAAAAAABAAASNFZ4AQESNFZ4AQL//
    wAACAAAAQAAe3sBAXt7AQIACAAAAAAAAAAAAAAAAAAAKAB6AAAAAAAH//wAA\", \"t\":
    \"1688208894.865789\", \"duration_sec\": 10, \"duration_nsec\": 528722047, \"reason\": \"
    OFPRR_IDLE_TIMEOUT\"}"], "dpid": 1}
{"eid": 68, "type": "HbPacketHandle", "pid_in": 22, "pid_out": [23], "mid_out": [24], "
    operations": ["{\"eid\": 75, \"type\": \"TraceSwitchFlowTableRead\", \"dpid\": 1, \"packet
    \": \"EjRWeAECEjRWeAEBCABFAAAcBfIAAEABe/Z7ewEBe3sBAggAKnXNiQAB\", \"in_port\": 1, \"
    flow_table\": [], \"flow_mod\": null, \"touched_flow_bytes\": null, \"t\":
    \"1688208894.866222\"}", "{\"eid\": 76, \"type\": \"TraceSwitchBufferPut\", \"dpid\": 1,
    \"packet\": \"EjRWeAECEjRWeAEBCABFAAAcBfIAAEABe/Z7ewEBe3sBAggAKnXNiQAB\", \"in_port\": 1,
    \"buffer_id\": 1, \"t\": \"1688208894.866244\"}"], "dpid": 1, "packet": "
    EjRWeAECEjRWeAEBCABFAAAcBfIAAEABe/Z7ewEBe3sBAggAKnXNiQAB", "in_port": 1}
{"eid": 78, "type": "HbMessageSend", "mid_in": 24, "mid_out": [25], "msg_type": "
    OFPT_PACKET_IN", "dpid": 1, "controller_id": ["127.0.0.1", 6633], "msg": "
    AQoAPAAAAAIAAAABADwAAQAAEjRWeAECEjRWeAEBCABFAAAcBfIAAEABe/Z7ewEBe3sBAggAKnXNiQAB"}
{"eid": 81, "type": "HbControllerHandle", "mid_in": 25, "mid_out": [26]}
{"eid": 86, "type": "HbControllerSend", "mid_in": 26, "mid_out": [27]}
```

```
{"eid": 84, "type": "HbMessageHandle", "pid_in": 23, "pid_out": [], "mid_in": 27, "mid_out":
    [], "msg_type": "OFPT_FLOW_MOD", "operations": ["{\"eid\": 87, \"type\": \"
    TraceSwitchFlowTableWrite\", \"dpid\": 1, \"flow_table\": [], \"flow_mod\": \"
    AQ4ASAAAAA8AAAABAAASNFZ4AQESNFZ4AQL//
    wAACAAAAQAAe3sBAXt7AQIACAAAAAAAAAAAAAAAAAAAKAB6AAAAAAAH//wAA\", \"t\":
    \"1688208894.86866\"}", "{\"eid\": 88, \"type\": \"TraceSwitchBufferGet\", \"dpid\": 1, \"
    packet\": \"EjRWeAECEjRWeAEBCABFAAAcBfIAAEABe/Z7ewEBe3sBAggAKnXNiQAB\", \"in_port\": 1, \"
    buffer_id\": 1, \"t\": \"1688208894.868766\"}", "{\"eid\": 89, \"type\": \"
    TraceSwitchPacketDrop\", \"dpid\": 1, \"packet\": \"EjRWeAECEjRWeAEBCABFAAAcBfIAAEABe/
    Z7ewEBe3sBAggAKnXNiQAB\", \"in_port\": 1, \"flow_table\": [\"
    AQ4ASAAAAAcAAAABAAASNFZ4AQESNFZ4AQL//
    wAACAAAAQAAe3sBAXt7AQIACAAAAAAAAAAAAAAAAAAAKAB6AAAAAAAH//wAA\"], \"t\":
    \"1688208894.868788\"}"], "dpid": 1, "controller_id": ["127.0.0.1", 6633], "packet": "
    EjRWeAECEjRWeAEBCABFAAAcBfIAAEABe/Z7ewEBe3sBAggAKnXNiQAB", "in_port": 1, "msg": "
    AQ4ASAAAAA8AAAABAAASNFZ4AQESNFZ4AQL//
    wAACAAAAQAAe3sBAXt7AQIACAAAAAAAAAAAAAAAAAAAKAB6AAAAAAAH//wAA"}
{"eid": 92, "type": "HbHostSend", "pid_in": 28, "pid_out": [29], "hid": 2, "packet": "
    EjRWeAEBEjRWeAECCABFAAAcBfQAAEABe/R7ewECe3sBAQgApbBSTQAC", "out_port":
    "12:34:56:78:01:02"}
{"eid": 94, "type": "HbPacketHandle", "pid_in": 29, "pid_out": [30], "mid_out": [31], "
    operations": ["{\"eid\": 97, \"type\": \"TraceSwitchFlowTableRead\", \"dpid\": 1, \"packet
    \": \"EjRWeAEBEjRWeAECCABFAAAcBfQAAEABe/R7ewECe3sBAQgApbBSTQAC\", \"in_port\": 2, \"
    flow_table\": [\"AQ4ASAAAAAkAAAABAAASNFZ4AQESNFZ4AQL//
    wAACAAAAQAAe3sBAXt7AQIACAAAAAAAAAAAAAAAAAAAKAB6AAAAAAAH//wAA\"], \"flow_mod\": null, \"
    touched_flow_bytes\": null, \"t\": \"1688208895.0681\"}", "{\"eid\": 98, \"type\": \"
    TraceSwitchBufferPut\", \"dpid\": 1, \"packet\": \"EjRWeAEBEjRWeAECCABFAAAcBfQAAEABe/
    R7ewECe3sBAQgApbBSTQAC\", \"in_port\": 2, \"buffer_id\": 1, \"t\":
    \"1688208895.068255\"}"], "dpid": 1, "packet": "EjRWeAEBEjRWeAECCABFAAAcBfQAAEABe/
    R7ewECe3sBAQgApbBSTQAC", "in_port": 2}
{"eid": 100, "type": "HbMessageSend", "mid_in": 31, "mid_out": [32], "msg_type": "
    OFPT_PACKET_IN", "dpid": 1, "controller_id": ["127.0.0.1", 6633], "msg": "
    AQoAPAAAAAMAAAABADwAAgAAEjRWeAEBEjRWeAECCABFAAAcBfQAAEABe/R7ewECe3sBAQgApbBSTQAC"}
{"eid": 103, "type": "HbControllerHandle", "mid_in": 32, "mid_out": [33]}
{"eid": 108, "type": "HbControllerSend", "mid_in": 33, "mid_out": [34]}
```

```
{"eid": 106, "type": "HbMessageHandle", "pid_in": 30, "pid_out": [], "mid_in": 34, "mid_out":
    [], "msg_type": "OFPT_FLOW_MOD", "operations": ["{\"eid\": 109, \"type\": \"
    TraceSwitchFlowTableWrite\", \"dpid\": 1, \"flow_table\": [\"
    AQ4ASAAAAAsAAAABAAASNFZ4AQESNFZ4AQL//
    wAACAAAAQAAe3sBAXt7AQIACAAAAAAAAAAAAAAAAAAAKAB6AAAAAAAH//wAA\"], \"flow_mod\": \"
    AQ4ASAAAABEAAAABAAASNFZ4AQISNFZ4AQH//
    wAACAAAAQAAe3sBAnt7AQEACAAAAAAAAAAAAAAAAAAAKAB6AAAAAAAH//wAA\", \"t\":
    \"1688208895.070892\"}", "{\"eid\": 110, \"type\": \"TraceSwitchBufferGet\", \"dpid\": 1,
    \"packet\": \"EjRWeAEBEjRWeAECCABFAAAcBfQAAEABe/R7ewECe3sBAQgApbBSTQAC\", \"in_port\": 2,
    \"buffer_id\": 1, \"t\": \"1688208895.071077\"}", "{\"eid\": 111, \"type\": \"
    TraceSwitchPacketDrop\", \"dpid\": 1, \"packet\": \"EjRWeAEBEjRWeAECCABFAAAcBfQAAEABe/
    R7ewECe3sBAQgApbBSTQAC\", \"in_port\": 2, \"flow_table\": [\"
    AQ4ASAAAAA0AAAABAAASNFZ4AQESNFZ4AQL//
    wAACAAAAQAAe3sBAXt7AQIACAAAAAAAAAAAAAAAAAAAKAB6AAAAAAAH//wAA\", \"
    AQ4ASAAAAA8AAAABAAASNFZ4AQISNFZ4AQH//
    wAACAAAAQAAe3sBAnt7AQEACAAAAAAAAAAAAAAAAAAAKAB6AAAAAAAH//wAA\"], \"t\":
    \"1688208895.071097\"}"], "dpid": 1, "controller_id": ["127.0.0.1", 6633], "packet": "
    EjRWeAEBEjRWeAECCABFAAAcBfQAAEABe/R7ewECe3sBAQgApbBSTQAC", "in_port": 2, "msg": "
    AQ4ASAAAABEAAAABAAASNFZ4AQISNFZ4AQH//
    wAACAAAAQAAe3sBAnt7AQEACAAAAAAAAAAAAAAAAAAAKAB6AAAAAAAH//wAA"}
```


B.2 DyNetKAT model

```
{
  "channels": [
    "escalateSwitch2",
    "escalateSwitch1"
  ],
  "program": "Controller6633 || Switch1Iteration0 || Switch2Iteration0",
  "recursive_variables": {
    "Controller6633": "( escalateSwitch1 ! 1 ; Controller6633 ) +o ( escalateSwitch2 ! 1 ;
        Controller6633 )",
    "Switch1Iteration0": "\"(( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304514 .
        _nw_src = 33651579 . _dl_vlan = 65535 . _nw_dst = 33651579 . dpid = 1 . _dl_src =
        20015998304514 . _tp_src = 8 ) )\" ; Switch1Iteration0 +o ( escalateSwitch1 ? 1 ;
        Switch1Iteration1 )",
    "Switch1Iteration1": "\"(( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304514 .
        _nw_src = 33651579 . _dl_vlan = 65535 . _nw_dst = 33651579 . dpid = 1 . _dl_src =
        20015998304514 . _tp_src = 8 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst =
        20015998304514 . _nw_src = 33717115 . _dl_vlan = 65535 . _in_port = 1 . _nw_dst =
        33651579 . dpid = 1 . _dl_src = 20015998304770 . _tp_src = 8 . port <- 2 ) )\" ;
        Switch1Iteration1 +o ( escalateSwitch1 ? 1 ; Switch1Iteration2 )",
    "Switch1Iteration2": "\"(( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304514 .
        _nw_src = 33651579 . _dl_vlan = 65535 . _nw_dst = 33651579 . dpid = 1 . _dl_src =
        20015998304514 . _tp_src = 8 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst =
        20015998304514 . _nw_src = 33717115 . _dl_vlan = 65535 . _in_port = 1 . _nw_dst =
        33651579 . dpid = 1 . _dl_src = 20015998304770 . _tp_src = 8 . port <- 2 ) + (
        _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304514 . _nw_src = 33717115 .
        _dl_vlan = 65535 . _in_port = 1 . _nw_dst = 33651579 . dpid = 1 . _dl_src =
        20015998304770 . _tp_src = 8 . port <- 2 ) )\" ; Switch1Iteration2 +o (
        escalateSwitch1 ? 1 ; Switch1Iteration3 )",
```

```
"Switch1Iteration3": "\"(( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304514 .
    _nw_src = 33651579 . _dl_vlan = 65535 . _nw_dst = 33651579 . dpid = 1 . _dl_src =
    20015998304514 . _tp_src = 8 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst =
    20015998304514 . _nw_src = 33717115 . _dl_vlan = 65535 . _in_port = 1 . _nw_dst =
    33651579 . dpid = 1 . _dl_src = 20015998304770 . _tp_src = 8 . port <- 2 ) + (
    _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304514 . _nw_src = 33717115 .
    _dl_vlan = 65535 . _in_port = 1 . _nw_dst = 33651579 . dpid = 1 . _dl_src =
    20015998304770 . _tp_src = 8 . port <- 2 ) + ( _nw_proto = 1 . _dl_type = 2048 .
    _dl_dst = 20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 . _in_port = 2 .
    _nw_dst = 33717115 . dpid = 1 . _dl_src = 20015998304514 . port <- 1 ) )\" ;
    Switch1Iteration3 +o ( escalateSwitch1 ? 1 ; Switch1Iteration4 )",
"Switch1Iteration4": "\"(( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304514 .
    _nw_src = 33651579 . _dl_vlan = 65535 . _nw_dst = 33651579 . dpid = 1 . _dl_src =
    20015998304514 . _tp_src = 8 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst =
    20015998304514 . _nw_src = 33717115 . _dl_vlan = 65535 . _in_port = 1 . _nw_dst =
    33651579 . dpid = 1 . _dl_src = 20015998304770 . _tp_src = 8 . port <- 2 ) + (
    _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304514 . _nw_src = 33717115 .
    _dl_vlan = 65535 . _in_port = 1 . _nw_dst = 33651579 . dpid = 1 . _dl_src =
    20015998304770 . _tp_src = 8 . port <- 2 ) + ( _nw_proto = 1 . _dl_type = 2048 .
    _dl_dst = 20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 . _in_port = 2 .
    _nw_dst = 33717115 . dpid = 1 . _dl_src = 20015998304514 . port <- 1 ) + ( _nw_proto =
     1 . _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579 . _dl_vlan =
    65535 . _in_port = 2 . _nw_dst = 33717115 . dpid = 1 . _dl_src = 20015998304514 . port
     <- 1 ) )\" ; Switch1Iteration4 +o ( escalateSwitch1 ? 1 ; Switch1Iteration5 )",
"Switch1Iteration5": "\"(( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304514 .
    _nw_src = 33651579 . _dl_vlan = 65535 . _nw_dst = 33651579 . dpid = 1 . _dl_src =
    20015998304514 . _tp_src = 8 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst =
    20015998304514 . _nw_src = 33717115 . _dl_vlan = 65535 . _in_port = 1 . _nw_dst =
    33651579 . dpid = 1 . _dl_src = 20015998304770 . _tp_src = 8 . port <- 2 ) + (
    _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304514 . _nw_src = 33717115 .
    _dl_vlan = 65535 . _in_port = 1 . _nw_dst = 33651579 . dpid = 1 . _dl_src =
    20015998304770 . _tp_src = 8 . port <- 2 ) + ( _nw_proto = 1 . _dl_type = 2048 .
    _dl_dst = 20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 . _in_port = 2 .
    _nw_dst = 33717115 . dpid = 1 . _dl_src = 20015998304514 . port <- 1 ) + ( _nw_proto =
     1 . _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579 . _dl_vlan =
    65535 . _in_port = 2 . _nw_dst = 33717115 . dpid = 1 . _dl_src = 20015998304514 . port
     <- 1 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src =
    33651579 . _dl_vlan = 65535 . _in_port = 2 . _nw_dst = 33717115 . dpid = 1 . _dl_src =
     20015998304514 . _tp_src = 8 . port <- 1 ) )\" ; Switch1Iteration5 +o (
    escalateSwitch1 ? 1 ; Switch1Iteration6 )",
```

```
"Switch1Iteration6": "\"(( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304514 .
    _nw_src = 33651579 . _dl_vlan = 65535 . _nw_dst = 33651579 . dpid = 1 . _dl_src =
    20015998304514 . _tp_src = 8 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst =
    20015998304514 . _nw_src = 33717115 . _dl_vlan = 65535 . _in_port = 1 . _nw_dst =
    33651579 . dpid = 1 . _dl_src = 20015998304770 . _tp_src = 8 . port <- 2 ) + (
    _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304514 . _nw_src = 33717115 .
    _dl_vlan = 65535 . _in_port = 1 . _nw_dst = 33651579 . dpid = 1 . _dl_src =
    20015998304770 . _tp_src = 8 . port <- 2 ) + ( _nw_proto = 1 . _dl_type = 2048 .
    _dl_dst = 20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 . _in_port = 2 .
    _nw_dst = 33717115 . dpid = 1 . _dl_src = 20015998304514 . port <- 1 ) + ( _nw_proto =
     1 . _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579 . _dl_vlan =
    65535 . _in_port = 2 . _nw_dst = 33717115 . dpid = 1 . _dl_src = 20015998304514 . port
     <- 1 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src =
    33651579 . _dl_vlan = 65535 . _in_port = 2 . _nw_dst = 33717115 . dpid = 1 . _dl_src =
     20015998304514 . _tp_src = 8 . port <- 1 ) + ( _nw_proto = 1 . _dl_type = 2048 .
    _dl_dst = 20015998304514 . _nw_src = 33717115 . _dl_vlan = 65535 . _in_port = 1 .
    _nw_dst = 33651579 . dpid = 1 . _dl_src = 20015998304770 . port <- 2 ) )\" ;
    Switch1Iteration6 +o ( escalateSwitch1 ? 1 ; Switch1Iteration7 )",
"Switch1Iteration7": "\"(( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304514 .
    _nw_src = 33717115 . _dl_vlan = 65535 . _in_port = 1 . _nw_dst = 33651579 . dpid = 1 .
     _dl_src = 20015998304770 . _tp_src = 8 . port <- 2 ) + ( _nw_proto = 1 . _dl_type =
    2048 . _dl_dst = 20015998304514 . _nw_src = 33717115 . _dl_vlan = 65535 . _in_port = 1
     . _nw_dst = 33651579 . dpid = 1 . _dl_src = 20015998304770 . _tp_src = 8 . port <- 2
    ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579
    . _dl_vlan = 65535 . _in_port = 2 . _nw_dst = 33717115 . dpid = 1 . _dl_src =
    20015998304514 . port <- 1 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst =
    20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 . _in_port = 2 . _nw_dst =
    33717115 . dpid = 1 . _dl_src = 20015998304514 . port <- 1 ) + ( _nw_proto = 1 .
    _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 .
    _in_port = 2 . _nw_dst = 33717115 . dpid = 1 . _dl_src = 20015998304514 . _tp_src = 8
    . port <- 1 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304514 . _nw_src
     = 33717115 . _dl_vlan = 65535 . _in_port = 1 . _nw_dst = 33651579 . dpid = 1 .
    _dl_src = 20015998304770 . port <- 2 ) )\" ; Switch1Iteration7",
"Switch2Iteration0": "\"(( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 .
    _nw_src = 33717115 . _dl_vlan = 65535 . _nw_dst = 33717115 . dpid = 2 . _dl_src =
    20015998304770 . _tp_src = 8 ) )\" ; Switch2Iteration0 +o ( escalateSwitch2 ? 1 ;
    Switch2Iteration1 )",
"Switch2Iteration1": "\"(( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 .
    _nw_src = 33717115 . _dl_vlan = 65535 . _nw_dst = 33717115 . dpid = 2 . _dl_src =
    20015998304770 . _tp_src = 8 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst =
    20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 . _in_port = 1 . _nw_dst =
    33717115 . dpid = 2 . _dl_src = 20015998304514 . port <- 2 ) )\" ; Switch2Iteration1 +
    o ( escalateSwitch2 ? 1 ; Switch2Iteration2 )",
"Switch2Iteration2": "\"(( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 .
    _nw_src = 33717115 . _dl_vlan = 65535 . _nw_dst = 33717115 . dpid = 2 . _dl_src =
    20015998304770 . _tp_src = 8 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst =
    20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 . _in_port = 1 . _nw_dst =
    33717115 . dpid = 2 . _dl_src = 20015998304514 . port <- 2 ) + ( _nw_proto = 1 .
    _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 .
    _in_port = 1 . _nw_dst = 33717115 . dpid = 2 . _dl_src = 20015998304514 . port <- 2 )
    )\" ; Switch2Iteration2 +o ( escalateSwitch2 ? 1 ; Switch2Iteration3 )",
```

```
"Switch2Iteration3": "\"(( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 .
    _nw_src = 33717115 . _dl_vlan = 65535 . _nw_dst = 33717115 . dpid = 2 . _dl_src =
    20015998304770 . _tp_src = 8 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst =
    20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 . _in_port = 1 . _nw_dst =
    33717115 . dpid = 2 . _dl_src = 20015998304514 . port <- 2 ) + ( _nw_proto = 1 .
    _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 .
    _in_port = 1 . _nw_dst = 33717115 . dpid = 2 . _dl_src = 20015998304514 . port <- 2 )
    + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579 .
    _dl_vlan = 65535 . _in_port = 1 . _nw_dst = 33717115 . dpid = 2 . _dl_src =
    20015998304514 . port <- 2 ) )\" ; Switch2Iteration3 +o ( escalateSwitch2 ? 1 ;
    Switch2Iteration4 )",
"Switch2Iteration4": "\"(( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 .
    _nw_src = 33717115 . _dl_vlan = 65535 . _nw_dst = 33717115 . dpid = 2 . _dl_src =
    20015998304770 . _tp_src = 8 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst =
    20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 . _in_port = 1 . _nw_dst =
    33717115 . dpid = 2 . _dl_src = 20015998304514 . port <- 2 ) + ( _nw_proto = 1 .
    _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 .
    _in_port = 1 . _nw_dst = 33717115 . dpid = 2 . _dl_src = 20015998304514 . port <- 2 )
    + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579 .
    _dl_vlan = 65535 . _in_port = 1 . _nw_dst = 33717115 . dpid = 2 . _dl_src =
    20015998304514 . port <- 2 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst =
    20015998304514 . _nw_src = 33717115 . _dl_vlan = 65535 . _in_port = 2 . _nw_dst =
    33651579 . dpid = 2 . _dl_src = 20015998304770 . _tp_src = 8 . port <- 1 ) )\" ;
    Switch2Iteration4 +o ( escalateSwitch2 ? 1 ; Switch2Iteration5 )",
"Switch2Iteration5": "\"(( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 .
    _nw_src = 33717115 . _dl_vlan = 65535 . _nw_dst = 33717115 . dpid = 2 . _dl_src =
    20015998304770 . _tp_src = 8 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst =
    20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 . _in_port = 1 . _nw_dst =
    33717115 . dpid = 2 . _dl_src = 20015998304514 . port <- 2 ) + ( _nw_proto = 1 .
    _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 .
    _in_port = 1 . _nw_dst = 33717115 . dpid = 2 . _dl_src = 20015998304514 . port <- 2 )
    + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579 .
    _dl_vlan = 65535 . _in_port = 1 . _nw_dst = 33717115 . dpid = 2 . _dl_src =
    20015998304514 . port <- 2 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst =
    20015998304514 . _nw_src = 33717115 . _dl_vlan = 65535 . _in_port = 2 . _nw_dst =
    33651579 . dpid = 2 . _dl_src = 20015998304770 . _tp_src = 8 . port <- 1 ) + (
    _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579 .
    _dl_vlan = 65535 . _in_port = 1 . _nw_dst = 33717115 . dpid = 2 . _dl_src =
    20015998304514 . _tp_src = 8 . port <- 2 ) )\" ; Switch2Iteration5 +o (
    escalateSwitch2 ? 1 ; Switch2Iteration6 )",
```

```
    "Switch2Iteration6": "\"(( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 .
        _nw_src = 33717115 . _dl_vlan = 65535 . _nw_dst = 33717115 . dpid = 2 . _dl_src =
        20015998304770 . _tp_src = 8 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst =
        20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 . _in_port = 1 . _nw_dst =
        33717115 . dpid = 2 . _dl_src = 20015998304514 . port <- 2 ) + ( _nw_proto = 1 .
        _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 .
        _in_port = 1 . _nw_dst = 33717115 . dpid = 2 . _dl_src = 20015998304514 . port <- 2 )
        + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579 .
        _dl_vlan = 65535 . _in_port = 1 . _nw_dst = 33717115 . dpid = 2 . _dl_src =
        20015998304514 . port <- 2 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst =
        20015998304514 . _nw_src = 33717115 . _dl_vlan = 65535 . _in_port = 2 . _nw_dst =
        33651579 . dpid = 2 . _dl_src = 20015998304770 . _tp_src = 8 . port <- 1 ) + (
        _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579 .
        _dl_vlan = 65535 . _in_port = 1 . _nw_dst = 33717115 . dpid = 2 . _dl_src =
        20015998304514 . _tp_src = 8 . port <- 2 ) + ( _nw_proto = 1 . _dl_type = 2048 .
        _dl_dst = 20015998304514 . _nw_src = 33717115 . _dl_vlan = 65535 . _in_port = 2 .
        _nw_dst = 33651579 . dpid = 2 . _dl_src = 20015998304770 . port <- 1 ) )\" ;
        Switch2Iteration6 +o ( escalateSwitch2 ? 1 ; Switch2Iteration7 )",
    "Switch2Iteration7": "\"(( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304770 .
        _nw_src = 33651579 . _dl_vlan = 65535 . _in_port = 1 . _nw_dst = 33717115 . dpid = 2 .
         _dl_src = 20015998304514 . port <- 2 ) + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst
        = 20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 . _in_port = 1 . _nw_dst =
        33717115 . dpid = 2 . _dl_src = 20015998304514 . port <- 2 ) + ( _nw_proto = 1 .
        _dl_type = 2048 . _dl_dst = 20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 .
        _in_port = 1 . _nw_dst = 33717115 . dpid = 2 . _dl_src = 20015998304514 . port <- 2 )
        + ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304514 . _nw_src = 33717115 .
        _dl_vlan = 65535 . _in_port = 2 . _nw_dst = 33651579 . dpid = 2 . _dl_src =
        20015998304770 . _tp_src = 8 . port <- 1 ) + ( _nw_proto = 1 . _dl_type = 2048 .
        _dl_dst = 20015998304770 . _nw_src = 33651579 . _dl_vlan = 65535 . _in_port = 1 .
        _nw_dst = 33717115 . dpid = 2 . _dl_src = 20015998304514 . _tp_src = 8 . port <- 2 ) +
         ( _nw_proto = 1 . _dl_type = 2048 . _dl_dst = 20015998304514 . _nw_src = 33717115 .
        _dl_vlan = 65535 . _in_port = 2 . _nw_dst = 33651579 . dpid = 2 . _dl_src =
        20015998304770 . port <- 1 ) )\" ; Switch2Iteration7"
  }
}
```