IZI: Intelligent Zero-Trust Network for IoT

Report 4: Traffic generation, network function virtualization and RL training environment

Mikhail Zolotukhin, Pyry Kotilainen and Timo Hämäläinen
Faculty of Information Technology, University of Jyväskylä, Finland.*

Abstract

With the recent progress in the development of low-budget sensors and machine-to-machine communication, the Internet-of-Things has attracted considerable attention. Unfortunately, many of today's smart devices are rushed to market with little consideration for basic security and privacy protection, making them easy targets for various attacks. Once a device has been compromised, it can become the starting point for accessing other elements of the network at the next stage of the attack, since traditional IT security castle-and-moat concept implies that nodes inside the private network trust each other. For these reasons, IoT will benefit from adapting a zero-trust networking model which requires strict identity verification for every person and device trying to access resources on a private network, regardless of whether they are located within or outside of the network perimeter. Implementing such model can however become challenging, as the access policies have to be updated dynamically in the context of constantly changing network environment. Thus, there is a need for an intelligent enhancement of the zero-trust network that would not only detect an intrusion on time, but also would make the most optimal real-time crisis-action decision on how the security policy should be modified in order to minimize the attack surface and the risk of subsequent attacks in the future. In this research project, we are aiming to implement a prototype of such defense framework relying on advanced technologies that have recently emerged in the area of software-defined networking and network function virtualization. The intelligent core of the system proposed is planned to employ several reinforcement machine learning agents which process current network state and mitigate both external attacker intrusions and stealthy advanced persistent threats acting from inside of the network environment.

1 Introduction

Increasing computing and connectivity capabilities of smart devices in conjunction with users and organizations prioritizing access convenience over security makes such devices valuable asset for cyber criminals. The intrusion detection in IoT is limited due to lack of efficient malware signatures caused by diversity of processor architectures employed by different vendors [1]. In addition to that, owners use mostly manual workflows to address malware-related incidents and therefore they are able to prevent neither attack damage nor potential attacks in the future. Furthermore, since not all devices support over-the-air security updates, or updates without downtime, they might need to be physically accessed or temporarily pulled from production. Thus, many connected smart devices may remain vulnerable and potentially infected for long time resulting in a material loss of revenue and significant costs incurred by not only device owners, but also users and organizations targeted by the attackers as well as network operators and service providers. A potential solution to these and other emerging challenges in IoT is employing zero-trust networking model, that implies that all data traffic generated must be untrusted, no matter if it has been generated from the internal or external network [2].

In this research, we aim to design and implement an intelligent zero-trust networking solution capable of detecting attacks initiated by both external attackers and smart devices from the inside, adapt detection models under constantly changing network context caused by adding new applications and services or discovering new vulnerabilities and attack vectors, make an optimal set of real-time crisis-action decisions on how the network security policy should be modified in order to reduce the ongoing attack surface and minimize the risk of subsequent attacks in the future. These decisions that may include permitting, denying, logging, redirecting, or instantiating certain traffic between end-points under consideration, are based on behavioral patterns observed in the network and log data obtained from multiple intrusion and

*This project has received funding from the European Union's Horizon 2020 research and innovation programme under the NGI_TRUST grant agreement no 825618

anomaly detectors and deployed on the fly with the help of cutting-edge cloud computing technologies such as software-defined networking and network function virtualization. Our implementation of the decision making mechanism in the system proposed is planned to rely on recent advances in reinforcement learning (RL), machine learning paradigm in which software agents automatically determine the ideal behavior within a specific context by continually making value judgments to select good actions over bad. RL algorithms can be used to solve very complex problems that cannot be solved by conventional techniques as they aim to achieve long-term results correcting the errors occurred during the training process.

Recent advent of cutting-edge technologies such as cloud computing, mobile edge computing, network virtualization, software-defined networking (SDN) and network function virtualization (NFV) have changed the way in which network functions and devices are implemented, and also changed the way in which the network architectures are constructed. More specifically, the network equipment or device is now changing from closed, vendor specific to open and generic with SDN technology, which enables the separation of control and data planes, and allows networks to be programmed by using open interfaces. With NFV, network functions previously placed in costly hardware platforms are now implemented as software appliances located on low-cost commodity hardware or running in the cloud computing environment. In this context, the network security service provision has shifted toward replacing traditional proprietary middle-boxes by virtualized and cloud-based network functions in order to enable automatic security service provision. Thus the idea is to implement RL agents which have observe the current state of the network and take optimal decisions on network policy changes implemented in the form of SDN flows and/or security middle box modifications in order to minimize the effect of the attack.

The purpose of this document is to describe the implementation process of the defense framework proposed. The rest of the document is organized as follows. In Section 2, we evaluate various deep learning algorithms needed for implementation. The potential solution for the traffic generation problem is shown in Section 3. Section 4 outlines implementation of SDN flows and security VNFs. The system prototype work is discussed in Section 5. Some preliminary results are provided in Section 6. Section 7 concludes the report and outlines future work.

2 Deep learning algorithms

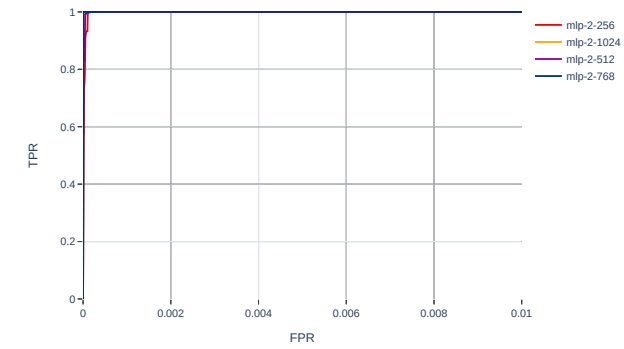
First, we studied basic deep learning models and evaluated them using realistic network traffic. A deep neural network consists of multiple layers of nonlinear processing units. The main idea behind deep learning is using the first layers to find compact low-dimensional representations of high-dimensional data whereas later layers are responsible for achievement of the task given, e.g. regression or categorical classification. All the neurons of the layers are activated through weighted connections. In order the network being capable to approximate a nonlinear transformation, a non-linear activation function is applied to the neuron output. The learning is conducted by calculating error in the output layer and backpropagating gradients towards the input layer. In regular deep neural network layer, each neuron in a hidden or output layer is fully connected to all neurons of the previous layer with the output being calculated by applying the activation function to the weighted sum of the previous layer outputs. Such layers have few trainable parameters and therefore learn fast compared to more complicated architectures.

To evaluate deep learning model capabilities to detect intrusions we use network packet captures from CICIDS2018 [12] dataset. It contains 560 Gb of traffic generated during 10 days by 470 machines. The dataset in addition to benign samples includes following attacks: infiltration of the network from inside, HTTP denial of service, web, SSH and FTP brute force attacks, attacks based on known vulnerabilities. We concentrate on the intrusion detection based on the analysis of network traffic flows. A flow is a group of IP packets with some common properties passing a monitoring point in a specified time interval: IP address and port of the source and IP address and port of the destination. Resulting flow measurements provide us an aggregated view of traffic information and drastically reduce the amount of data to be analyzed. After that, two flows such as the source socket of one of these flows is equal to the destination socket of another flow and vice versa are found and combined together. This combination is considered as one conversation between a client and the server. A conversation can be characterized by following four parameters: source IP address, source port, destination IP address and destination port.

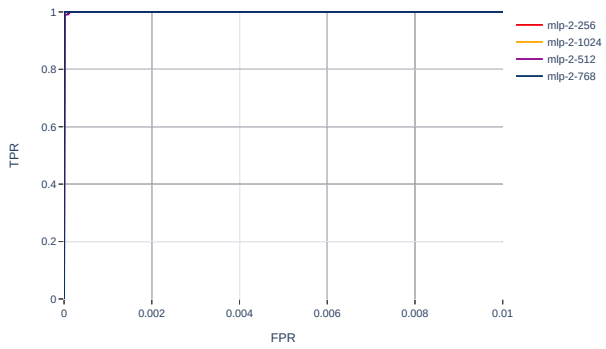
For each such conversation, at each time window, or when a new packet arrives, we extract the following information: flow duration, total number of packets in forward and backward direction, total size of the packets in forward direction, minimum, mean, maximum, and standard deviation of packet size in forward and backward direction and overall in the flow, number of packets and bytes per second, minimum, mean, maximum and standard deviation of packet inter-arrival time in forward and backward direction and overall in the flow, total number of bytes in packet headers in forward and backward direction, number of packets per second in forward and backward direction, number of packets with different TCP flags, backward-to-forward number of bytes ratio, average number of packets and bytes transferred in bulk in the forward and backward direction, the average number of packets in a sub flow in the forward and backward direction, number of bytes sent in initial window in the forward and backward direction, minimum, mean, maximum and standard deviation of time the flow is active, minimum, mean, maximum and standard deviation of time the flow is

idle. In addition, for the traffic transferred in plain HTTP, we also extract frequencies of 1-grams for first 256 ASCII codes for each packet in the flow. All the features can have different scale and therefore they are supposed to be standardized.

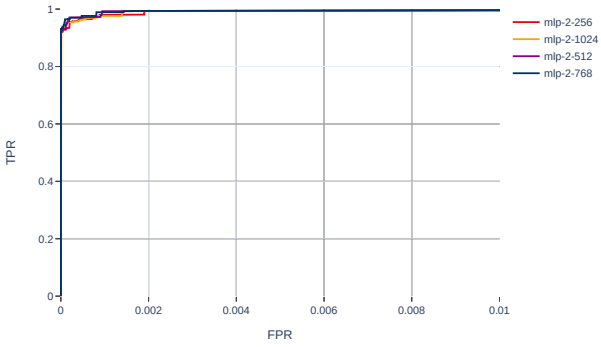
In our numerical experiments, we process raw packet capture files. First, we extract necessary packet features, then combine separate packets into conversations and, after that, we extract conversation features. It is worth noticing, that every time a new packet is transferred during the conversation or a certain time period (one second in our case) passes, we recalculate the conversation features and add a new data sample for the updated conversation. The idea behind that is that we attempt to evaluate how well the deep learning methods can detect intrusions in real time not when the conversation is over. Some results are presented on the Figures 1 and 2.



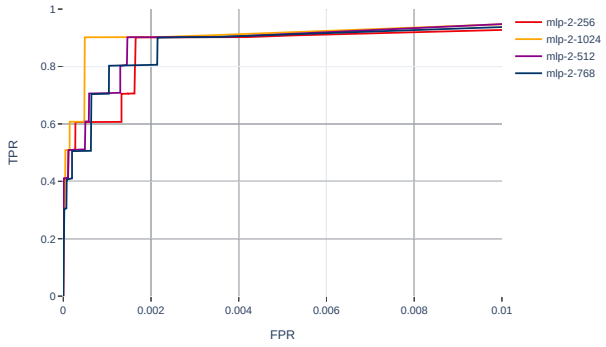
(a) DoS with GoldenEye.



(b) DoS with Hulk.



(c) Web bruteforce attacks.

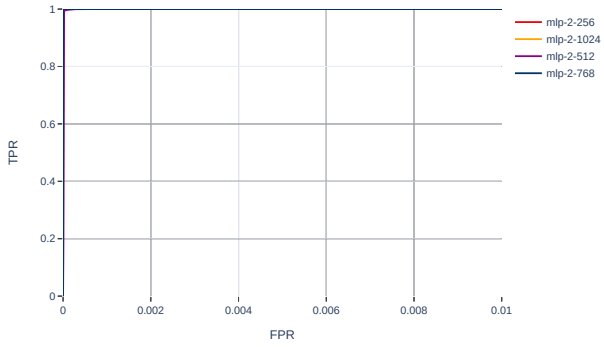


(d) Botnet attack.

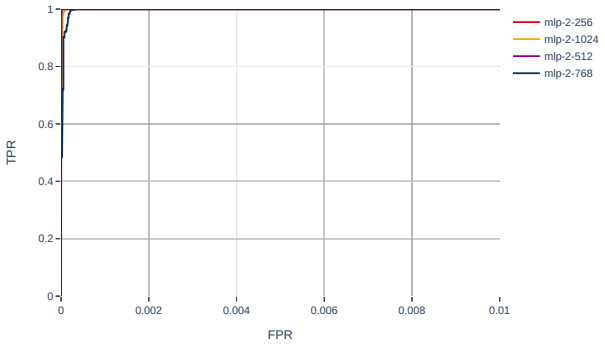
Figure 1: Dependence of TPR on FPR for different intrusion detection with deep learning models applied to flow features.

As one can notice from the figures, basic neural networks allow us to detect malicious connections without many false alarms. Results for the classification models slightly vary in terms of true and false positive rates depending on the architecture. It is also worth noticing that increasing the number of trainable parameters does not improve accuracy of the models significantly. In the sense of efficiency, simple MLPs look the most promising solution. It is worth noticing that we also experimented with more complicated neural network layers, e.g. residual [14] and self-attention [15], but for our classification task those do not provide any increase in the detection accuracy. We also tested unsupervised models such as autoencoders, however we did not manage to obtain good results using those.

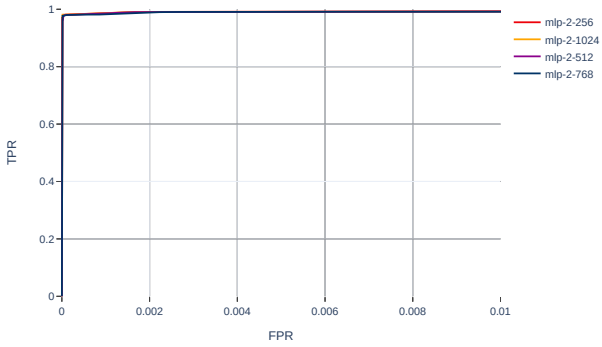
Second, we got familiar with the reinforcement learning approach. Reinforcement learning is a machine learning paradigm in which software agents and machines automatically determine the ideal behavior within a specific context by continually making value judgments to select good actions over bad. A reinforcement learning problem can be modeled as Markov Decision Process (MDP) that includes three following components: a set of agent states and a set of its actions, a transition probability function which evaluates the probability of making a transition from an initial state to the next state taking a certain action, and an immediate reward function which represents the reward obtained by the agent for a particular state transition. If the transition probability function is known, the agent can compute the solution



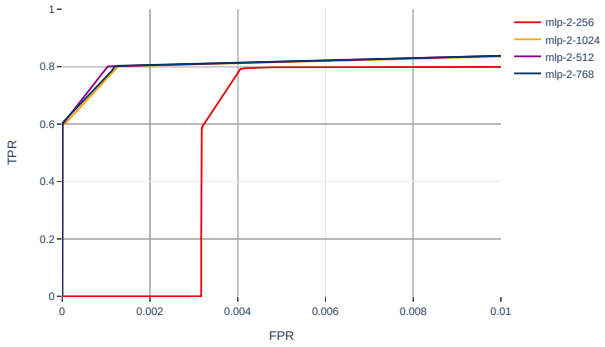
(a) DoS with GoldenEye.



(b) DoS with Hulk.



(c) Web bruteforce attacks.



(d) Botnet attack.

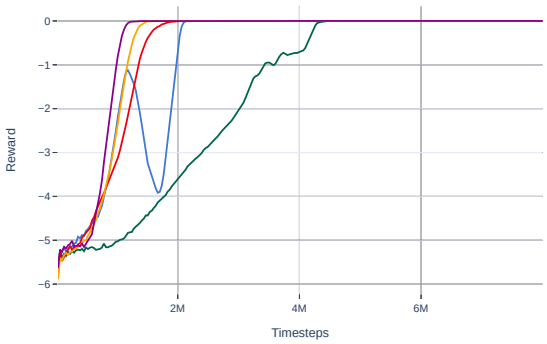
Figure 2: Dependence of TPR on FPR for different intrusion detection with deep learning models applied to payload features.

before executing any action in the environment. However, in real-world environment, the agent often knows neither how the environment will change in response to its action nor what immediate reward it will receive for executing the action. It is not enough to only account the immediate reward of the current state, the far-reaching rewards should also be taken into consideration. Most of the time RL algorithms focus on the optimization of infinite-horizon discounted model, implying that the rewards that come sooner are more probable to happen, since they are more predictable than the long term future reward.

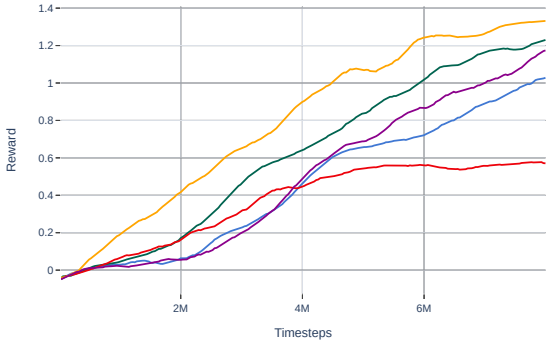
There are three main approaches for the reinforcement learning: value-based, policy-based and model-based. In value-based RL, the goal is to maximize the value function which is essentially a function that evaluates the total amount of the reward an agent can expect to accumulate over the future, starting at a particular state. The agent then uses this function by picking an action at each step that is believed to maximize the value function. On the other hand, policy-based RL agent attempts to optimize the policy function directly without using the value function. The policy function in this case is the function that defines the action the agent selects at the given state. Finally, model-based approach focuses on sampling and learning the probabilistic model of the environment which is then used to determine the best actions the agent can take. Assuming the model of the environment has been properly learned, model-based algorithms are much more efficient than model-free ones, however, since the agent only learns the specific environment model, it becomes useless in a new environment and requires time to learn another model.

To evaluate performance of different RL algorithms, we use OpenAI gym that has emerged recently as a standardization effort [16]. We run multiple copies of the environment in parallel. The training process is divided into episodes. Each episode lasts a certain fixed amount of time steps, during which one of the tasks is performed by an agent implemented using OpenAI baselines [17]. The tasks include swinging an inverted pendulum up from a random position, moving a bipedal walker through a one-dimensional track, and landing a space ship in a two-dimensional environment. We

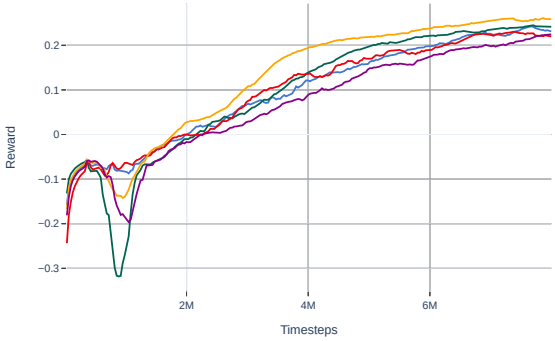
tested three state-of-art RL algorithms A2C [18], ACKTR [19] and PPO [20] in several virtualized environments. We concentrated on these algorithms as they can be applied for both discrete and continuous environments. In our experiments, PPO consistently provides good results in terms of both average reward and convergence speed. We run several experiments with different network architectures. The results on Figure 3 show that the network with one shared layer followed by two separate streams for policy and value function looks the most promising architecture variant. We also experimented with using shared LSTM layer for both policy and value function, but the results showed that much more steps is required for the algorithm convergence in this case, which can be critical in case of more complicated environment that requires more time per iteration.



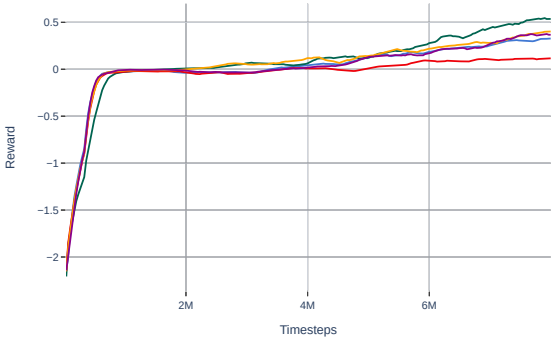
(a) Pendulum.



(b) Continuous mountain car.



(c) Bipedal walker.



(d) Lunar lander.

Figure 3: Performance of PPO2 with different policy and value network architectures in several OpenAI gym environments. Architectures: shared MLP (blue), one separate layer for value (green), one separate MLP layer for policy (red), one shared MLP layer (yellow), separate MLPs (purple)

3 Traffic generation

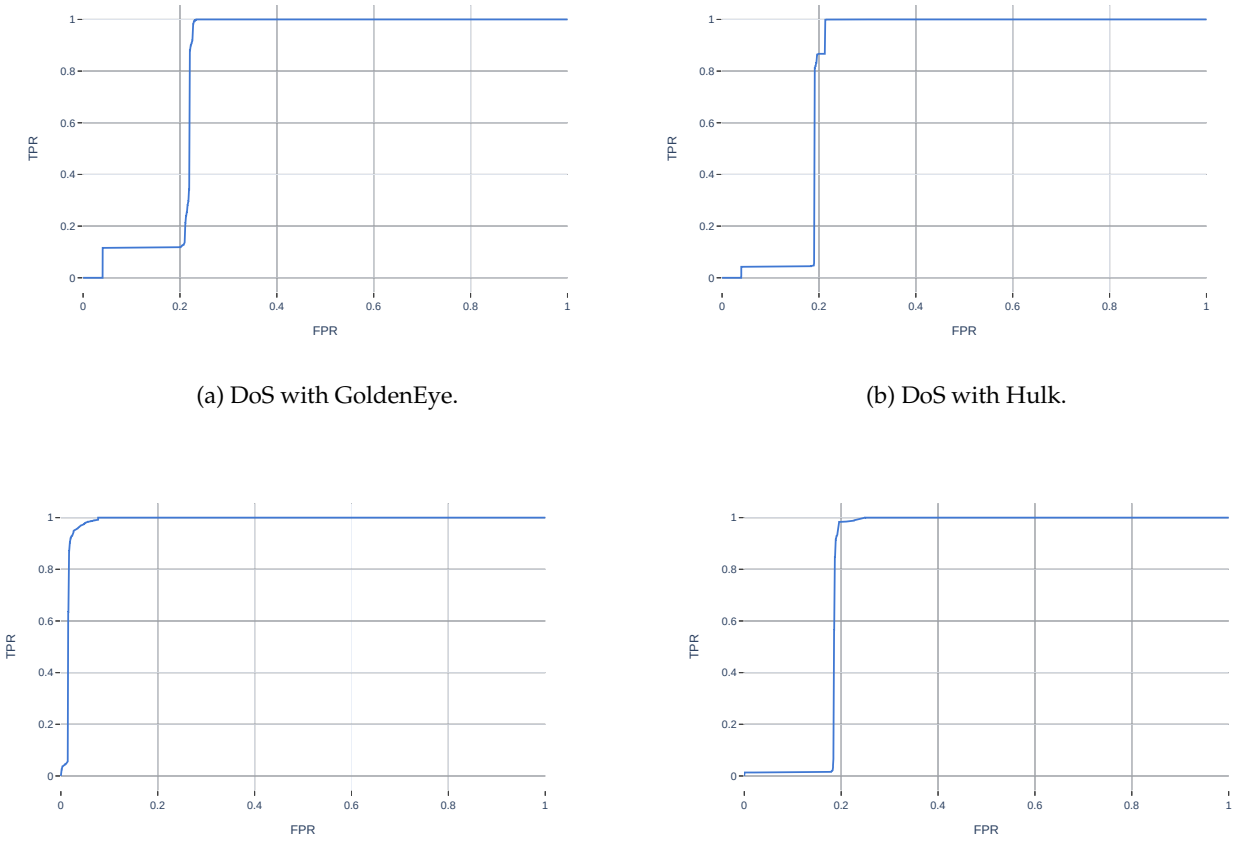
Next, we experimented with generative adversarial models. The idea was to build generative adversarial networks (GANs) which would allow us generate realistic network traffic in order to train the reinforcement learning agents in the simulation environment [21]. In GANs, the discriminator generates an estimate of the probability that a given sample is real or generated. The discriminator is supplied with a set of samples which include both real and generated ones and it would generate an estimate for each of these inputs. The error between the discriminator output and the actual labels would then be measured by cross-entropy loss. GAN can be extended to a conditional model if both the generator and discriminator are conditioned on some extra information [22]. We can perform the conditioning by feeding this information into the both the discriminator and generator as additional input layer.

Cross-entropy loss fails in some cases and not point in the right direction in other cases. This may lead to mode collapse when the generator only learns a small subset of the possible realistic samples which the discriminator cannot recognize.

One potential solution for this problem is using Wasserstein distance metric [23]. The Wasserstein metric looks at the distribution of each variable in the real and generated samples, and determines how far apart the distributions are for real and generated data. The Wasserstein metric looks at how much effort, in terms of mass times distance, it would take to push the generated distribution into the shape of the real distribution.

We tested conditional Wasserstein GANs for constructing network traffic packets. We trained two types of such models. The first generates inter-arrival time between two consecutive packets, payload size and TCP window size, the second generates n-gram distribution for the payload of the packet. Features for the condition include direction (request or reply) and TCP flags. In the generator network, a random noise vector is concatenated with the condition and the result is fed to an MLP, output of which is a feature vector for the next packet. The discriminator network also takes features extracted from the previous packets of the flow as an input. The second input is the feature vector generated by the generator. The generator produces packets that are closer to the real ones extracted from the datasets while the discriminator network tries to determine the differences between real and fake packets. The ultimate goal is to have two generative networks that can produce traffic which resembles the realistic one.

We trained such GANs separately for different attacks presented in the dataset and the normal HTTP traffic. Figures 4 and 5 show the results of applying the classifiers trained in the previous stage to the traffic generated with GANs. As one can see, the results for models which are trained with flow features are more or less inline with the ones obtained using the real data. However, such approach does not really work in case of the classifiers which are trained with n-grams of ASCII symbols extracted from packet payloads.



(a) DoS with GoldenEye.

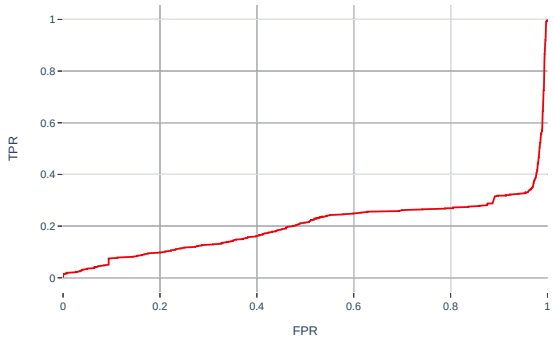
(b) DoS with Hulk.

(c) Web bruteforce attacks.

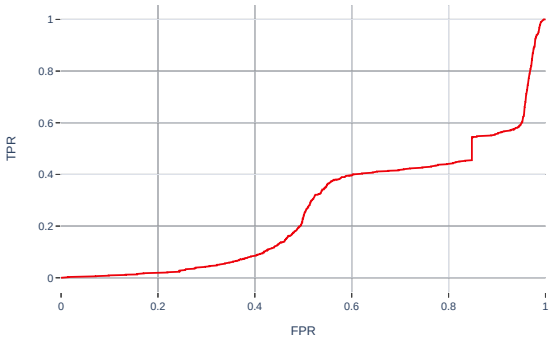
(d) Botnet attack.

Figure 4: Dependence of TPR on FPR for different intrusion detection with deep learning models applied to flow features extracted from fake traffic generated with Wasserstein GANs.

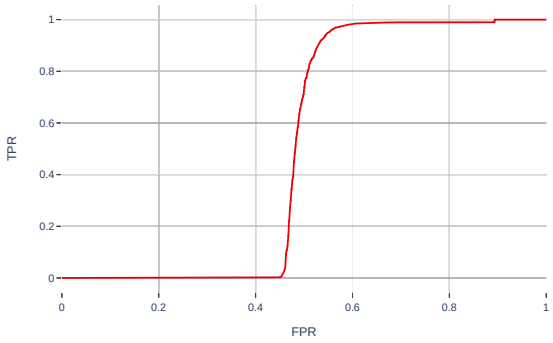
We implement an application for traffic generation in form of a Docker container. Docker allows users to package an application with all of its dependencies into a standardized unit for software development. Unlike virtual machines, containers do not have high overhead and hence enable more efficient usage of the underlying system and resources.



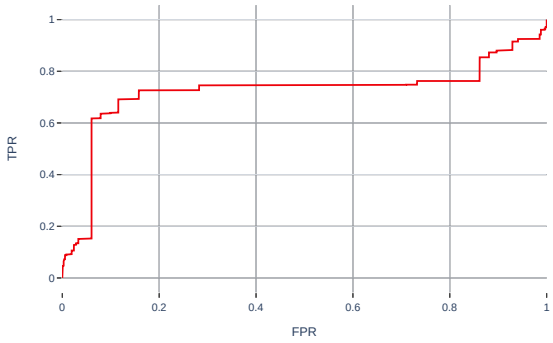
(a) DoS with GoldenEye.



(b) DoS with Hulk.



(c) Web bruteforce attacks.



(d) Botnet attack.

Figure 5: Dependence of TPR on FPR for different intrusion detection with deep learning models applied to payload features extracted from fake traffic generated with Wasserstein GANs.

The container includes client and server application implemented with Scapy module which is able to forge or decode packets of a wide number of protocols. We set the first ECN bit of each packet generated with the generator trained using malicious traffic to one in order to be able to provide ground truth labels for the AI in order to calculate the reward. Generator models of the trained GANs are first converted into a compressed .tflite format and added to the application. This has been done in order to deploy the trained model without installing the entire Tensorflow library.

4 SDN and NFV boxes

The main purpose of the SDN controller in our defense framework, which is to transform the security intent of the AI core to SDN flows and push them to the switches, can be implemented as an internal module of an existing SDN controller or an external application that uses RESTful APIs exposed by one or more plugins existing in the controller framework. There are many open-source controller options currently available, that can be modified in order to be used to redirect traffic between devices under protection and virtual security appliances. According to several SDN controller surveys, OpenDayLight [24] is one of the most featured controllers that are able to run on different platforms. Being under the partnership of well-known network providers and research communities, they have a clear development plan and good documentation. Even though Java-based OpenDayLight is inferior in performance compared to the controllers implemented in C in terms of throughput, they perform on similar level in terms of latency [25], which is alongside with high modularity and proper documentation makes it the most optimal choice to serve as an SDN controller in the defense system proposed.

We installed the following features on the controller:

- odl-openflowplugin-flow-services (for access to standard applications)

- odl-restconf-all (to push flows into the controller via rest API)
- odl-openflowplugin-nxm-extensions (to push flows with NXM matches and actions)

After that, we implemented a simple application for receiving information from operational data store of the controller and manipulating its config data store, which include the following functions:

- push a flow into a switch table
- find existing tables on a switch
- find existing flows on a switch
- delete flow from a switch table
- delete entire table from a switch

The basic functions allow us to setup basic network configurations using the following actions:

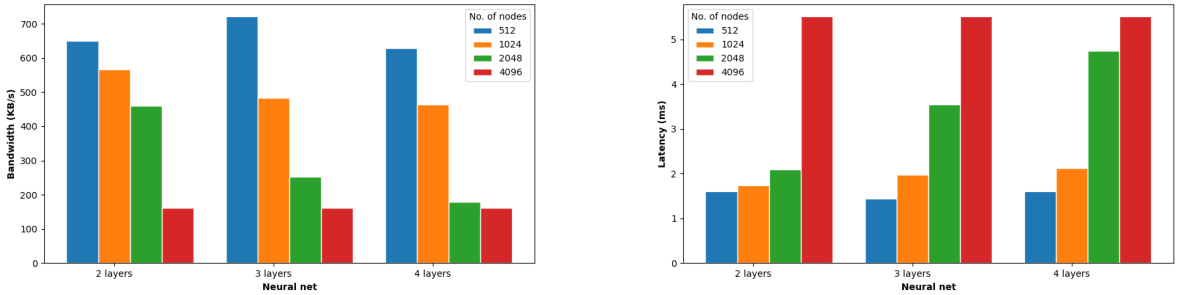
- resubmit packet with certain Ethernet protocol to another table
- reply to an ARP request with a MAC address
- output an ARP packet with certain target protocol address to a port
- output an IP packet with certain destination to a port
- output an IP packet with certain source to a port and resubmit to another table
- modify ECN of an IP packet with certain destination to a port
- modify ECN of an IP packet with certain destination to a port and resubmit to another table

The purpose of the last two actions is to change the second ECN bit of a packet to one when it arrives at the first switch and change it back to zero when it sent from the last switch. The idea is to account for packets which are dropped in the environment in order to calculate the impact of the defense framework.

Concerning the virtual security functions, there are many open-source intrusion detection and packet inspection software available that can be implemented as security middle boxes for timely attack detection and mitigation. We implement our own security middle box in order to use deep learning models trained. For that purpose, we first install OpenVSwitch on an Ubuntu virtual machine and connect it to our Opendaylight controller. It can then be connected to other network switches via VXLAN tunnels. For intercepting and analyzing network traffic we use Libnetfilter_queue [26] and Iptables firewall rules to gain access to network packets and the ability to reject or accept these packets for forwarding. Python library Netfilterqueue is used to interface with Libnetfilter_queue [27] from a python program. The interceptor program receives a network packet and extracts relevant features from it and uses pre-trained classifier to determine whether it is malicious or not. The malicious flows are flagged by setting a desired bit in the TCP-protocol DSCP field (upper 6 bits of the TOS field) facilitating detection and further actions by downstream devices. All packets are then forwarded regardless of the analysis result.

Similarly to the traffic generation containers we use Tensorflow Lite interpreter in order to avoid installing the entire Tensorflow library. We select the best classifier in terms of the metric selected (e.g. accuracy or AUC) for each attack class tested and copy those models to the VNF. We finally implement a simple API using Flask which allows to manipulate two parameters: classifier model to use for the analysis and the threshold according to which we differentiate normal traffic from malicious one. This is basically a very straight forward implementation of transfer learning approach, i.e. a model developed for a task is reused as the starting point for a model on a second task. We train a model using traffic contained in the dataset and then use all its layers except the last one as a foundation for the classifier inside our VNF. The last layer is essentially one number since we only classify traffic as either normal or malicious. Thus, an intelligent agent can manipulate VNfs implemented by selecting the most optimal combination of the classifier and the value of the threshold.

The effect on network performance can be measured by setting up a three machine virtual network: two virtual machines in separate subnets and a virtual machine running the interceptor program acting as a router between the subnets. The software tool Qperf [28] is used to analyze network performance. A Qperf server is started on one of the test machines and a client running a test against the server was started on the other. All traffic passed through the interceptor machine and through the analysis. The recorded metrics are bandwidth and latency of TCP traffic. The test is repeated several times with the interceptor analyzing packets with several different classifiers. The tested configurations have 2,3 or 4 layers of 512, 1024, 2048 or 4096 nodes. The results are shown in figure 6. These results accompanied with the ROC curves obtained previously allow us to conclude that it is reasonable to use MLP classifiers with less trainable parameters, since increasing the number of trainable parameters does not improve accuracy of the models significantly, however it negatively affects the network performance.



(a) Bandwidth through the interceptor.

(b) Latency through the interceptor.

Figure 6: Network performance of the interceptor with varying number of layers and nodes in each layer.

5 System overview

In order to implement the system prototype, the following actions have to be carried out:

- Extract necessary packet features from the traffic in the selected datasets
- Group packet features extracted into conversations
- Generate classification datasets for training models to detect intrusions based on features extracted from the conversations and/or features extracted from the payload
- Compile and train several classification models
- Select the best models in terms of the metric selected for detection of different classes of intrusions
- Convert the classifiers trained into .tflite format and store them for VNF implementation
- Use the features extracted previously to generate datasets for realistic flow and payload generation
- Train generative models using these datasets
- Create Docker container with the resulting generative models accompanied by a packet crafting scripts
- Push the container to a Docker repository

As a result, we require link to the traffic generation container and a directory with intrusion detection classifiers.

After that, we can start setting up the training environment using Vagrant. Vagrant [29] is an open-source program which allows for automatic building and managing virtual machines. Vagrant uses existing hypervisors, in our case Qemu/KVM through Libvirt, to deploy and run the machines. Vagrant manages these machines through SSH connections and can provide access to files for the virtual machines through NFS shares. We use Vagrant to configure the VMs required for the system implementation as follows:

ODL:

- Download specified version of Opendaylight and extract files
- Modify its configuration file to include necessary features
- Create and enable systemd service to run the controller on the system startup

OVS:

- Install OpenVSwitch and set Opendaylight as the controller
- Install Docker and enable remote access to the Docker service

IDS:

- Install OpenVSwitch and set Opendaylight as the controller
- Copy trained classifiers and scripts for packet interception and manipulation to the VM
- Enable IP forwarding and add the forwarding rule to Iptables to forward traffic through Nfqueue
- Create and enable systemd service to run the interception script on the system startup

MON:

- Install OpenVSwitch
- Copy simple script which monitors all the traffic sent and received
- Create and enable systemd service to run the script on the system startup

We use Ubuntu 18.04 as the base image, we install build essentials and python3 if needed. Switches are connected between each other with VXLAN tunnels. It is worth noticing that switch of the monitor is not controlled by Opendaylight, it just acts as a "sink" for the network traffic in order to provide information on the network state for an RL agent. We use OpenAI gym [16] to implement the frontend for the virtualized environment. The RL agent is implemented using OpenAI baselines [17]. The resulting environment is shown in Figure 7.

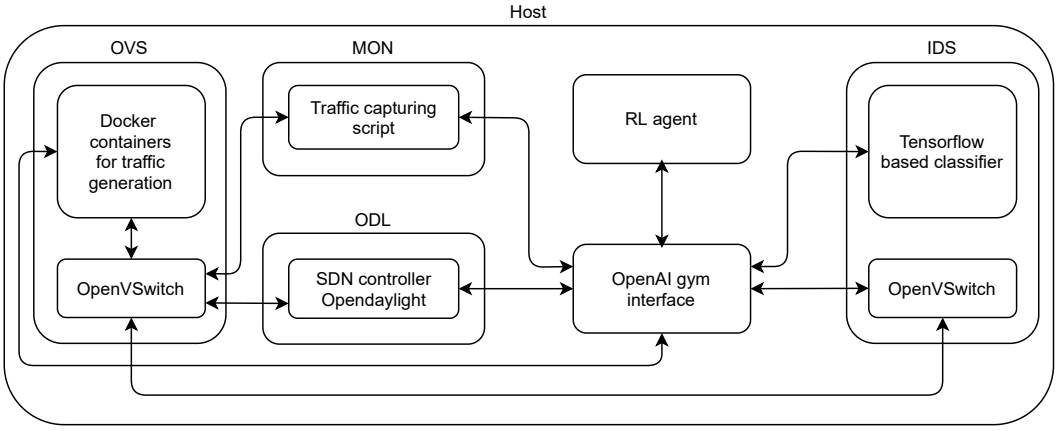


Figure 7: The environment for training RL agent implemented using Vagrant.

The agent observes the following statistics of the environment:

- Packet and byte counts sent from one subnet to another for each pair of subnets in the environment
- One-hot encoded indexes of the classifiers deployed in the security middle boxes
- Threshold values used in the classifiers

As one can notice we require to account for traffic transferred between internal subnets in the network. All external hosts can be classified as belonging to the same subnet.

Action set of the RL agent includes:

- Change the classifier model index for a certain VNF
- Change the threshold for a certain VNF
- Redirect traffic between a certain pair of subnets having certain DSCP label to a certain middle box
- Block traffic between a certain pair of subnets having certain DSCP label

For the reward function calculation we utilize our flow monitor VM. Since all malicious traffic packets generated have the first ECN bit equal to 1 and all packet which are received on the first switch have the second ECN bit equal to one, we cam calculate percentages of the normal and malicious traffic which is blocked by the environment. The reward function is then calculated as a weighted sum of these two metrics.

6 Preliminary results

Since we have not yet managed to carry out experiments in the environment described above, we implement another environment using SDN controller Opendaylight and Docker as shown in Figure 8). We use OpenAI gym to implement the frontend for the virtualized environment. Both virtualized devices and security VNFs are created as Docker containers connected to each other via Openflow-enabled OpenVSwitches. Each copy of the environment is deployed in a dedicated virtual machine with applications belonging to different devices running in different containers. At each VM, there is a simple web server which is used to obtain necessary information features and to enter actions from a RL

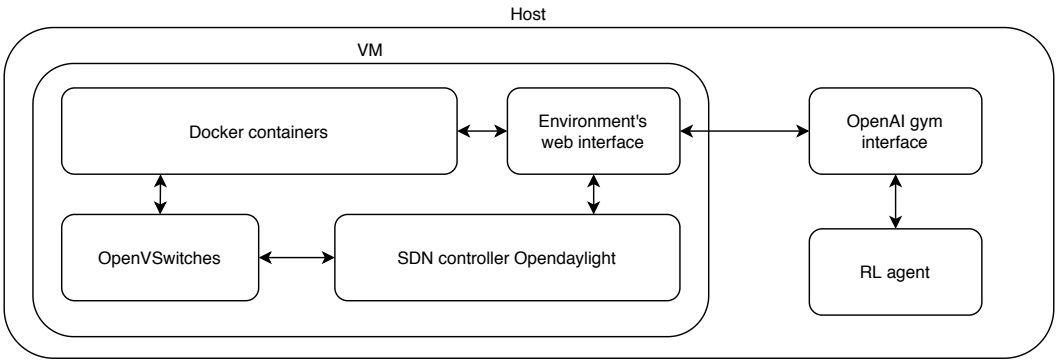


Figure 8: The environment for training RL agent implemented in Docker containers.

agent. The actions are implemented in form of SDN flows which result in redirecting certain traffic to one of the security containers.

We implemented several types of such containers: traditional signature-based IDS Snort, custom machine learning based anomaly detection IDS and honeypots that are essentially virtual machines with open SSH and Telnet ports and several user accounts with very easy-to-guess passwords. For each VNF, we implemented simple web interface that allows one to request their logs over TCP protocol.

Signature-based IDS. First, we launch Snort appliance that uses the latest sets of community rules. However, it turns out, that those sets do not include rules for some basic intrusions, e.g. SSH password brute force attack and DNS tunneling. For this reason, we implemented a simple set of custom rules by analyzing legitimate network traffic generated in the network environment under consideration. We find the maximum amount of packets of a particular type sent to a particular host per second and generate a list of rules that throw an alert if one of such thresholds is exceeded. This turns Snort into a basic anomaly-based detection system, since its rules are configured based on the normal behavior patterns. We implement two types of Snort middle boxes: one uses community rules and another relies on our custom rules.

Anomaly-based IDS. In order to detect anomalous payloads, we rely on centroid-based clustering. A small set of the normal traffic collected in advance before the RL agent training starts is used to divide 2-grams of the packet payloads into several clusters with SOM algorithm. For each cluster, we calculate its radius as sum of the mean distance between the centroid and the samples of the cluster and the standard deviation of these distance values multiplied by a configurable parameter. During the inference phase, we evaluate resulting clusters using a mixed set of normal and malicious traffic. If the distance between a new sample and its nearest cluster centroid exceeds the threshold, the sample is marked as anomalous.

Honeypot. We launch a standard Linux box, install SSH and Telnet server on it and add several users with default name-password combinations used by Mirai botnet [30]. Other services that connect to the network are disabled. Thus, if the honeypot appliance attempts to establish a connection with an external host, we mark this host as suspicious.

Firewall. We use SDN flows to block suspicious connections by installing rules to drop packets from a particular source host to a particular destination socket that use a particular IP protocol. We push these SDN flows to the switches with two different timeout values: zero, i.e. no timeout constraint, and one, i.e. the rule lasts only one second to block the connection in the current time window. We also implement pass rule that removes all SDN flows that affect certain network traffic from the controller storage.

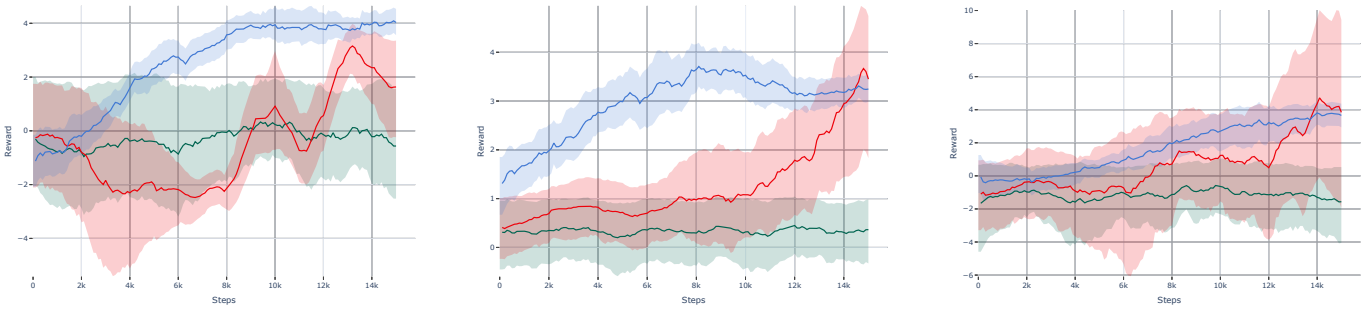
We initialize flow tables of each SDN switch with one single flow that forwards packets to the next table, until the last table is reached. SDN flows to drop packets or redirect them to a particular security appliance are pushed to the dedicated flow table with priority higher than default forwarding rule. The last table outputs packet to the physical port of its destination as well as mirrors packet to a special patch port, on which we run a flow collector. For each network activity from a particular source host to a particular destination socket we extract a set of features that include destination port, packet size, TCP flag counts, and several others. Furthermore, security alerts are requested from the corresponding appliances and added to the feature vectors.

In order to evaluate the framework proposed, we designed a simple Python application that sends a random text message to one of the external data servers. Every arbitrary amount of time (between one and three seconds) each device connects to a randomly selected update server and requests several files from it. Some devices are accessed via SSH by external entities imitating the administration process. DNS queries are resolved with the help of the internal DNS server. To generate malicious traffic, we implemented a simple Mirai-like malware with three attack capabilities. First, the malware is able to scan its local network looking for open SSH server ports and, in case such server is found, attempts to login to

the server using a short list of user-password combinations. The password brute-forcing is carried out in multiple threads with the list of credentials shuffled randomly in the beginning of the attack. If the correct password has been found, the malware initiates download of its copy to the compromised device from an external server. When the download is complete, the malware initiates a HTTP connection to its C&C server to inform that the attack has been successful. In case the C&C server is not available via HTTP, the malware sends a query with specific domain name to the DNS server. The domain name is essentially an encoded version of the same report that the malware sends to the C&C server via HTTP. The DNS server is configured in advance to forward such queries to the domain that belongs to the attacker. The second attack type performed by the malware uses DNS tunneling to exfiltrate a randomly selected file found on the device to the attacker server using scheme similar to the C&C channel. Finally, once multiple devices are infected with the malware, the attacker performs an application-based slow DDoS attack Slowloris against one of the data servers used by the legitimate application by sending never ending HTTP requests.

We run multiple copies of the environment in parallel. The training process is divided into episodes. Each episode lasts a certain fixed amount of time, during which one of the attacks is performed. The RL agent is implemented using OpenAI baselines. The agent selects one of the actions for each flow that are sent to the environment back-end where they are transformed to SDN rules. The reward for the action is proportional to the number of packets transferred during the most recent time window and it is calculated for each flow separately. The proportion coefficients are positive for legitimate traffic and negative for the malicious one. The exact values of the coefficients are estimated by running the attack without the agent and counting the average number of packets that are sent by the application and the attacker. The coefficients are then calculated as such a way, that the total reward without the agent's intervention is equal to zero.

We first train the RL agent using DQN and PPO algorithms with multi-layer perceptron (MLP) as both policy and value function to detect and mitigate the attacks mentioned. In case of DQN, first 80 % of the episodes are used for ϵ-greedy exploration with ϵ value decreasing from one to zero. For PPO, we collect data from entire episode to calculate cumulative rewards and advantages for each unique host-to-socket tuple, before dividing the resulting dataset to mini-batches which are used to train both the critic and the actor network. With DQN, for each such tuple, we store the state vector, the action taken by the agent, the reward and the feature vector of the same tuple in the next time window, only if there is still active connection. Otherwise, the state is marked as the terminate for this particular tuple. Figure 9 shows the evolution of the reward function throughout the training episodes for both DQN and PPO in case of three attacks mentioned. As one can notice, both algorithms are able to identify and block malicious connections reducing the number of malicious flows to minimum and subsequently increasing the reward value.



(a) Infection stage. (b) Exfiltration stage. (c) Slowloris stage.

Figure 9: Performance of DQN (red) and PPO (blue) during different attacks. Green line corresponds to "do nothing" policy.

7 Conclusion

The main contribution of this research is developing a proof-of-concept of an intelligent network defense system which relies on SDN and NFV technologies and allows customers to detect and mitigate attacks performed against their smart devices by letting an artificial intelligent agent control network security policy. On infrastructure level, the defense framework proposed includes cloud compute servers in order to emulate elements of real infrastructure as well as launch security appliances. In order to forward traffic from the network under protection to these appliances as well as connect the appliances to each other, the system relies on SDN capabilities that include global visibility of the network state and run-time manipulation of traffic forwarding rules. The key component of the defense system proposed is a reinforcement learning agent that resides on top of the SDN and NFV controllers and is responsible for manipulating security policies depending on the current network state. In particular, the agents processes traffic flowing through edge switches as well as log reports from security appliances deployed and manipulates the network traffic by instructing the SDN controller to pass, forward or block certain connections. We have not managed to complete the working prototype planned, however,

we conducted some experiments using a simplified version of the environment. We used this version to evaluate two state-of-art reinforcement learning algorithms for mitigating three basic network attacks against a small virtual network environment.

In addition to time constraints, there were other challenges we faced during the project. Those are mostly related to the traffic generation procedure. We managed to implement simple traffic generation application, it however does not really represent the realistic traffic and therefore its usage is limited in a real world scenario. A potential solution would be to use real devices and applications and generate traffic using those. In the future, we are planning to continue this research by conducting experiments in the environment prototype, as well as testing various reinforcement learning algorithms for different attack scenarios. We are also aiming to improve the scalability of the framework proposed and evaluate the system performance for bigger network environments. We are also going to implement adversarial module for the traffic generators which would allow for spoofing a neural-network-based intrusion detection system by manipulating flow parameters. Finally, we are going to test the working prototype of the network defense system developed during the project in a non-SDN enterprise network environment.

References

[1] M. Alhanahnah, Q. Lin, Q. Yan, N. Zhang, and Z. Chen. Efficient signature generation for classifying cross-architecture IoT malware. Proc. of IEEE Conf. on Communications and Network Security, pp. 1–9, 2018.

[2] J. Kindervag. No More Chewy Centers : Introducing The Zero Trust Model Of Information Security. pp. 1–15, 2010.

[3] B. Osborn, J. McWilliams, B. Beyer, M. Saltonstall. BeyondCorp; Design to Deployment at Google, login, vol. 41, no. 1, 2016.

[4] T. Yu, S. Fayaz, M. Collins, V. Sekar, and S. Seshan. Psi: Precise security instrumentation for enterprise networks. Proc. of the 24th Network and Distributed System Security Symposium, 2017.

[5] L. Fernandez Maimo, L. Perales Gomez, F. J. Garcia Clemente, et al. A Self-AdaptiveDeep Learning-Based System for Anomaly Detection in 5G Networks. IEEE Access, vol. 6, pp. 7700–7712, 2018.

[6] O. E. David, N. S. Netanyahu. DeepSign: Deep learning for automatic malware signature generation and classification. Proc. of IEEE IJCNN, pp. 1–8, 2015.

[7] Z. Fang, J. Wang, B. Li, S. Wu, Y. Zhou and H. Huang. Evading Anti-malware Engines with Deep Reinforcement Learning. IEEE Access, 2019.

[8] R. Vanickis, P. Jacob, S. Dehghanzadeh and B. Lee. Access Control Policy Enforcement for Zero-Trust-Networking. Proc. of the 29th Irish Signals and Systems Conference (ISSC), Belfast, pp. 1-6, 2018.

[9] M. Samaniego and R. Deters. Zero-Trust Hierarchical Management in IoT. Proc. of IEEE International Congress on Internet of Things (ICIOT), pp. 88-95, 2018.

[10] A. Sendi, H. Louafi, W. He, and M. Cheriet. Dynamic optimal countermeasure selection for intrusion response system. IEEE Transactions on Dependable and Secure Computing, vol. 15, no. 5, pp. 755–770, 2016.

[11] M. Landauer, M. Wurzenberger, F. Skopik, G. Settanni and P. Filzmoser. Dynamic log file analysis: An unsupervised cluster evolution approach for anomaly detection. Computers & Security, vol. 79, pp. 94–116, 2018.

[12] I. Sharafaldin, A. Lashkari, and A. Ghorbani. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. Proc of the 4-th International Conference on Information Systems Security and Privacy (ICISSP), pp. 108–116, 2018.

[13] N. Moustafa. Designing an online and reliable statistical anomaly detection framework for dealing with large high-speed network traffic. PhD diss., University of New South Wales, Canberra, Australia, 2017.

[14] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. Proc. of the IEEE conference on computer vision and pattern recognition, pp. 770–778, 2016.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. Proc. of the 31st International Conference on Neural Information Processing Systems (NIPS'17). pp. 6000–6010, 2017.

[16] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang and Wojciech Zaremba. OpenAI Gym.arXiv:1606.01540, 2016.

[17] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov. OpenAI Baselines. GitHub, https://github.com/openai/baselines, 2017.

[18] V. Mnih, A. Badia, M. Mirza, A. Graves, T.Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. arXiv preprint arXiv:1602.01783, 2016.

[19] Y. Wu, E. Mansimov, S. Liao, R. Grosse and J. Ba. Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation. arXiv preprint arXiv:1708.05144, 2017.

[20] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.

[21] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. Advances in neural information processing systems, pp. 2672–2680, 2014.

[22] M. Mirza and S. Osindero. Conditional Generative Adversarial Nets. arXiv preprint arXiv:1411.1784, 2014.

[23] M. Arjovsky, S. Chintala and L. Bottou. Proc. of the 34th International Conference on Machine Learning, Vol. 70, pp 214–223, 2017.

[24] J. Medved, R. Varga, A. Tkacik and K. Gray. OpenDaylight: Towards a Model-Driven SDN Controller architecture. Proc. of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks, pp. 1–6, 2014.

[25] O. Salman, I. H. Elhajj, A. I. Kayssi, and A. Chehab. SDN controllers: A comparative study. Proc. of the 18th Mediterranean Electrotechnical Conference (MELECON), pp. 1–6, 2016.

[26] The netfilter.org "libnetfilter_queue" project. Retrieved December 1, 2020, from https://www.netfilter.org/projects/libnetfilter_queue/index.html.

[27] NetfilterQueue - PyPi. Retrieved December 1, 2020, from https://pypi.org/project/NetfilterQueue/.

[28] Github - linux-rdma/qperf. Retrieved December 1, 2020, from https://github.com/linux-rdma/qperf.

[29] Vagrant by HashiCorp. Retrieved November 19, 2020, from https://www.vagrantup.com.

[30] M. Antonakakis, T. April, M. Bailey, et al. Understanding the mirai botnet. Proc. of the 26th USENIX Conference on Security Symposium (SEC). pp. 1093–1110, 2017.

A Deliverable: deep learning algorithms implemented in the form of scripts and software libraries.

Link: github.com/mizolotu/izi_algorithms

Description: Python code for carrying out experiments with network traffic datasets as well as testing OpenAI RL algorithms.

Usage:

1. Add a packet labelling function into file `data_processing/label_packets.py`
2. Extract basic packet features from raw PCAP files:

   ```
   python3 process_data.py -m read.pcap -i <input directory> -o <output_directory>
   ```

3. Group packets into flows:

   ```
   python3 process_data.py -m group_packets -i <input directory>
   -o <output_directory>
   ```

4. Extract flow features:

   ```
   python3 process_data.py -m summarize_groups -i <input directory>
   -o <output_directory>
   ```

5. Create datasets for flow classification:

   ```
   python3 create_datasets.py -m tcp_classification -i <input directory>
   -o <output_directory>
   ```

6. Create datasets for packet generation:

   ```
   python3 create_datasets.py -m tcp_generation -i <input directory>
   -o <output_directory>
   ```

7. Train intrusion detection classifiers:

   ```
   python3 detect_intrusions.py -i <directory with datasets> -o <output_directory>
   -l <attack type>
   ```

8. Train generative networks:

   ```
   python3 generate_traffic.py -i <directory with datasets> -o <output_directory>
   -l <traffic type>
   ```

9. Plot ROC curves

```
python3 plot_rocs.py -i <directory with models> -o <output_directory>
-l <traffic type>
```

10. Select best classfiers and transform them into .tflite format:

```
python3 prepare_models.py -t tcp_classification -i <directory with models>
-o <output_directory>
```

11. Transform generators from GAN models into .tflite format:

```
python3 prepare_models.py -t tcp_generation -i <directory with models>
-o <output_directory>
```

Other functionality

12. Test A2C, ACKTR and PPO algorithms:

```
python3 test_agents.py -e <environment index> -a <algorithm index>
-o <output directory>
```

13. Plot rewards:

```
python3 plot_rewards.py -i <directory with results> -o <output directory>
-e <environment>
```

B Deliverable: a software module to generate realistic network traffic of different kind including both benign and malicious flows.

Link: github.com/mizolotu/izi_traffic

Description: Simple traffic generation application implemented using Scapy and other Python modules built as a docker container. It requires having generators trained with GAN algorithm.

Usage:

1. Build docker container:

```
sudo docker build . -t <tag_name>
```

2. Push the container to the docker hub:

```
sudo docker push <tag_name>
```

3. Create a client-server couple of containers as follows:

```
sudo docker run -dt --name server <tag_name>
sudo docker run -dt --name client <tag_name>
```

4. Start traffic generation using the new containers:

```
sudo docker exec -dt server "python3 servber.py -m <mode> -t <traffic type>"
sudo docker exec -dt server "python3 client.py -m <mode> -t <traffic type>
-r <server's ip>"
```

5. Test traffic generated against classifiers trained in the previous stage:

```
python3 test_traffic_against_classifier.py -i <generated pcap files>
-m <classifier models>
```

C Deliverable: a virtualized network environment for training software RL-based security agents (includes API modules for an Opendaylight SDN controller and custom security virtual middle-boxes).

Link: github.com/mizolotu/izi_vagrant

Description: Simple virtual environment for RL agent training, includes an application for pushing certain SDN flows and deep learning based security middle boxes for flow labeling.

Usage:

1. Create VMs using vagrant and install necessary packages:

   ```
   sudo python3 create_vms.py
   ```

   Connect OVS instances running on those VMs using VXLAN tunnels:

   ```
   sudo python3 connect_switches.py
   ```

   Specify networks in a json file similar to test1.json and put it to directory scenarios/network. Setup networks as follows:

   ```
   sudo python3 setup_networks.py -s <scenario_file>
   ```

   Provision default SDN flows and IDS ip settings:

   ```
   sudo python3 provision_sdn.py
   ```

   Specify traffic in a json file similar to test1.json and put it to directory scenarios/traffic. Start training an RL agent:

   ```
   sudo python3 train_agent.py
   ```