IZI: Intelligent Zero-Trust Network for IoT

Report 2: Deep learning algorithm development (part 1)

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

Abstract

Artificial intelligence and deep learning are revolutionizing almost every industry with a seemingly endless list of applications ranging from object recognition for systems in autonomous vehicles to helping doctors detect and diagnose cancer. This list includes multiple branches of the field of cyber-security that include intrusion detection, malware classification, network traffic analysis and anomaly detection. In this project, we are aiming to employ state-of-art deep learning algorithms for developing a network defense framework that would be able to detect attacks in high-speed network environments, adapt detection models under constantly changing network context caused by adding new applications and services. The key component of the framework is supposed to rely on multiple reinforcement learning agents that evaluate the state of the network environment under consideration and make the most optimal real-time crisis-action decision on how the network security policy should be modified in order to minimize the risk of subsequent attacks in the future. These decisions are partially based on the information provided by deep classification and anomaly detection models that are deployed on virtual security middle boxes. Finally, the training of the agents is planned to be implemented with the help of deep generation models that generate artificial network traffic based on the real user behavioral patterns. In this report, we briefly outline the deep learning algorithms required for carrying out each of the tasks mentioned and provide some preliminary results using publicly available network traffic datasets and simulation environments.

1 Introduction

As mentioned in the first report, the key component of the defense system proposed consists of multiple reinforcement learning agents that reside on top of the SDN and NFV controllers and they are 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. In general, 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 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

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

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.

RL algorithms require both value and policy function to be approximated based on sample data collected during interaction with the environment [1]. This approximation problem can be solved by representing those functions in form of deep multi-layer neural networks. 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, however they may suffer when dealing with spatio-temporal data such as images and time-series.

Most of the time, convolutional neural networks (CNNs) are employed for automatic extraction of low-Level features such as edges, color, gradient orientation in image related problems [2]. The main building block of CNN is the convolutional layer which calculates an integral that expresses the amount of overlap of the layer's filter as it is shifted over the input data. Similarly to the previous case, the integral value is passed through an activation function to account for non-linearity in data. As a rule, multiple convolutions are performed on the input, each using a different filter. Resulting feature maps are then stuck together and become the final output of the convolution layer. CNNs can be employed to handle data of any dimension, since the result of the convolution operation is always a scalar. After a convolution operation, pooling can be performed to reduce the dimensionality of the output. The most common type of pooling is max pooling which takes the max value in the pooling window. Contrary to the convolution operation, pooling has no parameters. It slides a window over its input, and simply takes the max value in the window. CNNs usually consist of several convolutional layers mixed with few pooling layers followed by standard fully-connected layers. Stacking multiple convolutional layers allows one to learn both basic features as well as higher level representations to recognize objects in different shapes and positions.

Temporal dependencies in the data can be extracted with the help of recurrent neural networks (RNNs). In distinction to a fully-connected layer, a recurrent layer assumes that input data samples are time-series. To accommodate this fact, each recurrent layer has its own internal state the value of which is calculated based on the state value of the previous sample. The output of the recurrent layer is essentially an activation of the weighted sum of the previous layer outputs added to the weighted sum of the previous state values. During the learning process, derivatives are backpropagated through time, all the way to the beginning or to a certain point. All the derivatives multiply the same weight matrix which may result in either infinite or vanishing update values. While gradient exploding can be fixed by straight-forward clipping [3], dealing with gradient vanishing requires an intelligent control over the state via forget gates [4, 5]. It is worth mentioning that convolutional layers can also be employed to extract features from temporal data. For example, in context of reinforcement learning, DeepMind's Q-network that teaches itself to play Atari games, stacks last four frames of the historical data to produce an input for the CNN [8].

Recent studies propose a new architecture that replaces RNNs with attention mechanism [6]. An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key. In particular, the transformer relies on an encoder-decoder (autoencoder) structure, both of which, encoder and decoder, consist of several feed-forward and multi-head attention layers. In general, an autoencoder consists of an input layer, several hidden layers, and an output layer. The objective of the network is for the output layer to be exactly the same as the input layer despite the information bottleneck caused by the hidden layers. The process of going from the first layer to the hidden layer is called encoding. The process of going from the hidden layer to the output layer is called decoding. In this learning process, an autoencoder essentially learns the format rules of the input data. This property allows one to apply autoencoders for anomaly detection. This process usually involves two main steps. First, train data is fed to an autoencoder until it is well trained to reconstruct the expected output with minimum error. Second, the same data is fed again to the trained autoencoder and the error term of each reconstructed data point is measured. In theory, a well-trained autoencoder essentially learns how to reconstruct an input that follows a certain format, giving a badly formatted data point to a well-trained autoencoder will most likely result in something that is quite different from the expected input, and a large error term. This mechanism is used to classify anomalies by comparing the reconstruction error of an unknown sample to the error threshold observed during the training.

The last but not the least deep learning mechanism is worth mentioning in the introduction is residual neural networks [7]. Such networks aim to alleviate the vanishing gradient problem that is typical in deep learning architectures: as the gradient is back-propagated to earlier layers, repeated multiplication makes the gradient infinitely small. The core idea of residual networks is introducing a so-called identity shortcut connection that skips one or several layers. Stacking the resulting residual blocks on top of each other allows one to build a very deep network architecture which does not suffer from vanishing gradients. Having residual blocks with the shortcut also makes it very easy for one of the blocks to learn an identity function. This means that additional blocks can be stuck with little risk of harming the performance. The original paper that introduced residual networks used two different blocks: identity block and convolutional block. The difference is the later has additional convolution, batch normalization and activation layers in the main path and convolution accompanied by the batch normalization in the shortcut.

Speaking of the reinforcement learning, deep Q-Network (DQN) proposed in [8] presents the first deep value-based RL model to learn control policies directly from high-dimensional sensory input. In particular, the original DQN algorithm used the images shown on the Atari emulator as input, using convolution neural network to process image data. The value function of each action at each time step, Q-function, is evaluated using Bellman equation [9] that is proven to converge to an optimal value [10]. DQN uses a deep neural network as the function approximation to estimate the value function. The network is trained by minimizing the loss function, which is essentially the difference between the value of Q-function predicted in that particular time step and the target value function that is evaluated using the real reward value obtained from the environment. There are two main issues with using deep Q-networks. Fist, deep learning assumes data samples to be independent, however, the training data for the Q-network are collected by the sequence correlated states which led out by actions chosen. Second, the collected data distributions are non-stationary, since the agent keeps learning new strategies at every iteration. To overcome these issues, two following mechanisms can be employed: experience replay and freezing the target. The later requires constructing two neural networks with the same structure, but different weight values. One of these networks is updated online at each training step, while weights of the another one are kept frozen for a fixed amount of iterations. To calculate value of the loss function during training, Q-function value is predicted with the first neural network, while the target is evaluated using the second one. Weights of the target network are periodically updated with weight values of the online network. Since the evaluation of the target value uses the old parameters whereas the predicted value uses the current parameters, this can break the data correlation efficiently. Moreover, this mechanism allows one to avoid policy oscillations caused by rapid changes of the Q-function. The second mechanism, experience replay, is the method which aims to to break correlations between data samples by accumulating a buffer of experiences from many previous episodes and training the agent by sampling mini-batches of experiences randomly from this buffer uniformly at random.

DQN is proven to be powerful tool that could deal with problems involving low-dimensional, discrete observations and actions. However, in real world engineering applications, not only the dimensionality of both states and actions can be high, both the states and actions are often continuous. When there are a finite number of discrete actions, Q-function maximization poses no problem, because we can just compute the Q-values for each action separately and directly compare them, whereas when the action space is continuous, solving the resulting optimization problem can be non-trivial. In principle, DQN can be used to solve such problems using a set of applicable tools ranging from adaptive discretization and function approximation approaches to macro-actions or options [1]. However, this approach by design can lead to non-optimal solutions. Deep Deterministic Policy Gradient (DDPG) has been developed specifically for dealing with environments that operate in continuous action spaces [11]. Similarly to DQN, DDPG uses the Bellman equation to learn the Q-function which is in turn used to derive and learn the optimal policy. In addition to the value-function in DDPG, the second neural network that represents the agent's policy is employed to learn a deterministic policy which for every given state of the environment returns the action that maximizes the Q-function. Assuming the Q-function is differentiable with respect to action, gradient ascent is performed with respect to policy parameters only to find the action that maximizes the Q-value. Both DQN tricks experience replay and freezing the target can be also used with DDPG. The only difference is that in DQN the target network is just copied over from the main network every some-fixed-number of steps, whereas in DDPG-style algorithms, the target network is updated once per main network update by Polyak averaging. There is also difference in the exploration process. DQN often relies on ϵ-greedy policy meaning that ϵ percent of the time action is chosen completely at random. Value ϵ is usually set to decay over time to allow the algorithm to concentrate more on exploiting the best strategies that it has found. On the other hand, DDPG explores by adding a mean-zero Gaussian noise to the actions or policy parameters at the training stage.

Finally, in order to train the reinforcement learning agents a simulation environment is supposed to be constructed since we cannot deploy, train and test those agents in a real production network. Traffic in such environment can be generated with the help of various data generation methods. There are two main approaches to solve this problem using deep learning structures; teacher forcing and generative adversarial networks (GANs). A tractable implementation may be to attempt to generate continuous valued parameter vectors which are then converted to concrete network packets, as the generation of packet features directly to form a legal packet would likely be infeasible. Both GANs and teacher forcing have significant challenges when applied to our task. The main issue is the non-differentiability of the conversion to a packet, which is a problem also encountered in text generation, when converting output to discrete word. In the context of GANs, the gradient information from the discriminator cannot be propagated through the conversion to a packet to train the generator network. The applicable remedies from text generation approaches are to use a reinforcement learning approach, which does not depend on propagation of gradients, and instead treats the output of the discriminator as the reward signal, or to work with the produced continuous values directly, without conversion to a concrete packet. The latter approach requires that real network packets used for training be converted to the corresponding continuous valued parameter vector which would decode into the packet. This in turn might mean the use of an autoencoder to discover the suitable embedding. The teacher forcing approach suffers similarly from the conversion to a concrete packet, and working directly with the continuous valued vectors again necessitates the conversion of the real packets used for training to the corresponding continuous valued vectors.

The purpose of this document is to provide background information for the deep learning algorithms we are planning to use for implementation of the defense framework proposed in the project. The rest of the document is organized as follows. In Section 2, we implement and test several deep learning solutions for the problem of online intrusion detection. Reinforcement learning methods are outlined in Section 3. Section 4 concludes the paper and outlines future work.

2 Classification and anomaly detection

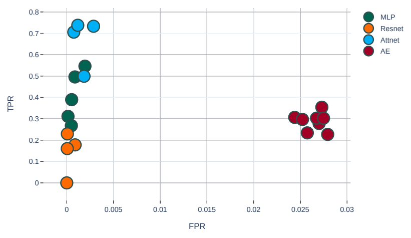
To evaluate deep learning model capabilities to detect intrusions we use network packet captures from CICIDS2018 [15] and UNSW-NB15 [16] datasets. The former contains 560 Gb of traffic generated during 10 days by 470 machines. The dataset in addition to benign samples includes following attacks: infiltration of the network from inside, HTTP denial of service, web, SSH and FTP brute force attacks, attacks based on known vulnerabilities. The latter contains 100 Gb of traffic generated during two days by three servers, two routers and multiple clients. This dataset includes the following types of attacks: fuzzers, backdoors, DoS, exploits, reconnaissance, shellcodes and worms. In case of TCP and UDP traffic, 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. We also extract frequencies of 1-grams for first 256 ASCII codes for each packet in the flow, however we did not notice any improvements using these features in the preliminary experiments. All the features can have different scale and therefore they are supposed to be standardized.

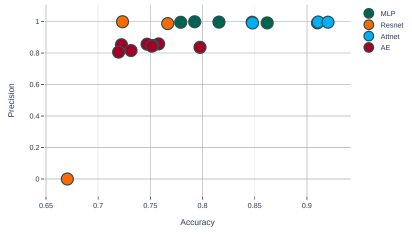
In our numerical experiments, we process raw packet capture files. First, we extract necessary packet features, then combine separate packets into conversations and, after that, we extract conversation features. It is worth noticing, that every time a new packet is transferred during the conversation 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. We test four following neural network architectures:

- MLP: multiple interconnected dense layers with batch normalization and dropout
- AttNet: multiple self-attention layers
- ResNet: multiple residual identity blocks
- AutoEncoder (anomaly detection): multiple interconnected dense layers

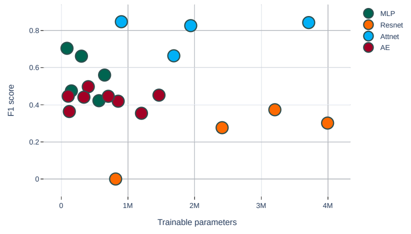
We evaluate each architecture by testing various numbers of layers and nodes in each layer. In each test, we calculate the following metrics: true positive and false positive rates, accuracy and precision, the number of trainable parameters and F1 score. Results are presented on the Figures below.



(a) FPR vs. TPR

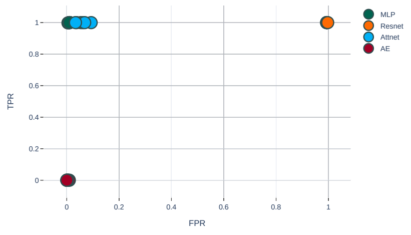


(b) Accuracy vs. precision

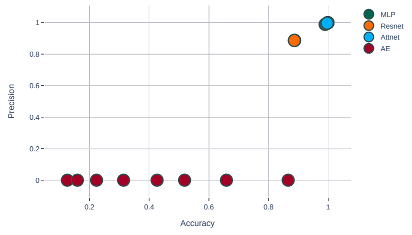


(c) Trainable parameters vs. F1 score

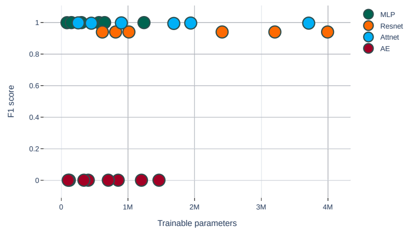
Figure 1: Performance evaluation of malicious traffic detection deep learning algorithms using HTTP/HTTPS traffic. Attacks include protocol and application DoS attacks, cross-site scripting, password bruteforcing, SQL injections, and communication between infected devices and C&C.



(a) FPR vs. TPR

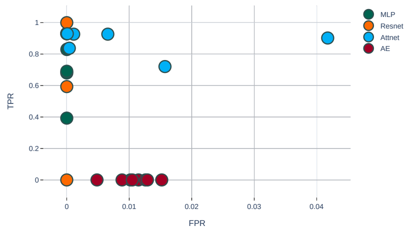


(b) Accuracy vs. precision

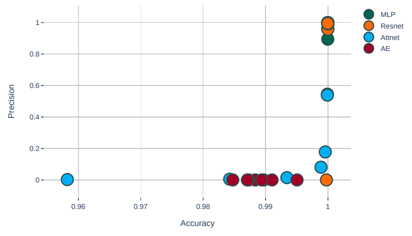


(c) Trainable parameters vs. F1 score

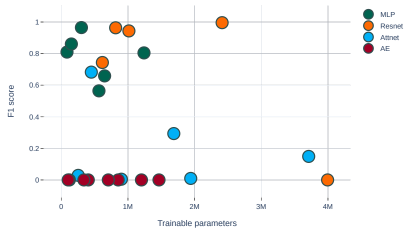
Figure 2: Performance evaluation of malicious traffic detection deep learning algorithms using SSH traffic. The only attack type performed against SSH servers is password bruteforcing.



(a) FPR vs. TPR



(b) Accuracy vs. precision



(c) Trainable parameters vs. F1 score

Figure 3: Performance evaluation of malicious traffic detection deep learning algorithms using the traffic other than HTTP/HTTPS and SSH. This group includes conversations generated during the infiltration attack.

As one can notice from Figure 1, all neural networks except autoencoders allow us to detect malicious HTTP connections without many false alarms. Results for the classification models vary in terms of true positive rate with self-attention-based neural networks providing the most accurate results. 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. Speaking of the autoencoders, poor results are not surprising as those models detect intrusions via unsupervised learning. However, some anomalies detected by the autoencoders turn out to be intrusions and therefore such models can be implemented in form of security middle boxes in order to detect unknown attacks. Figure 2 shows that almost 100 % of malicious SSH connections are successfully detected using either MLP or self-attention layers. However, autoencoders cannot find any anomalous samples. Finally, infiltration attacks can be detected using residual networks as one can see from Figure 3.

3 Reinforcement learning

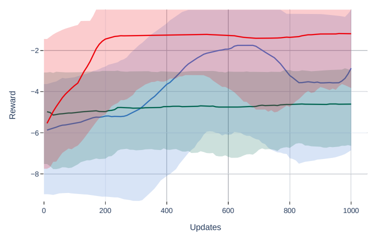
This section presents performance results of the state-of-art RL algorithms highlighted in the previous section evaluated in several virtualized environments imitating different tasks varying from balancing a pendulum to mitigating simple network intrusions. We first perform tests in simple Python-based environments to evaluate performance of RL algorithms. After that, we show how the algorithms tested can be applied to solve network security problems using more sophisticated custom environment we implemented using OpenDayLight [17] and Docker [18].

To evaluate performance of different RL algorithms, we use OpenAI gym that has emerged recently as a standardization effort [19]. 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 [20]. 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. Table 1 shows more detailed information about the OpenAI gym environments used for the evaluation.

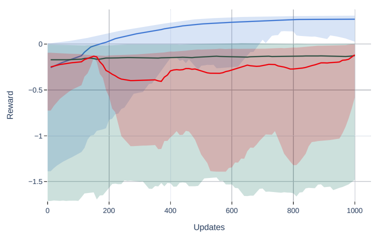
Table 1: OpenAI gym environments.

Parameter	Environment		
	Pendulum	Bipedal walker	Lunar lander
Number of environment copies	16		
Time steps per episode / per environment	125 / 2500		
State space	[-1, 1] x [-1, 1] x [-8, 8]	24 * $(-\infty, \infty)$	8 * $(-\infty, \infty)$
Action space	[-2, 2]	4 * [-1, 1]	2 * [-1, 1]
Reward function	$(-\infty, 0)$	[-100, 300]	[-100, 100]

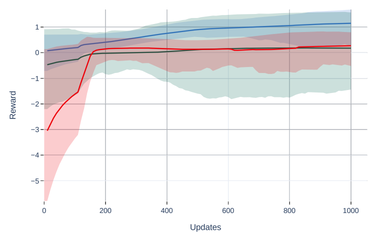
We evaluate three following RL algorithms: DDPG, A2C, and PPO. Evaluation results are presented in in Figures 4. The first observation we can make is that RL agents are capable of learning to act in all the environments tested. Moreover, PPO consistently provides good results in terms of both average reward and convergence speed.



(a) Pendulum.



(b) Bipedal walker.



(c) Lunar lander.

Figure 4: Performance of DDPG (red), A2C (green) and PPO (blue) in several OpenAI gym environments.

In addition, we implement our own network security environment using SDN controller Opendaylight and Docker as shown in Figure 5). We use OpenAI gym [19] 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 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.

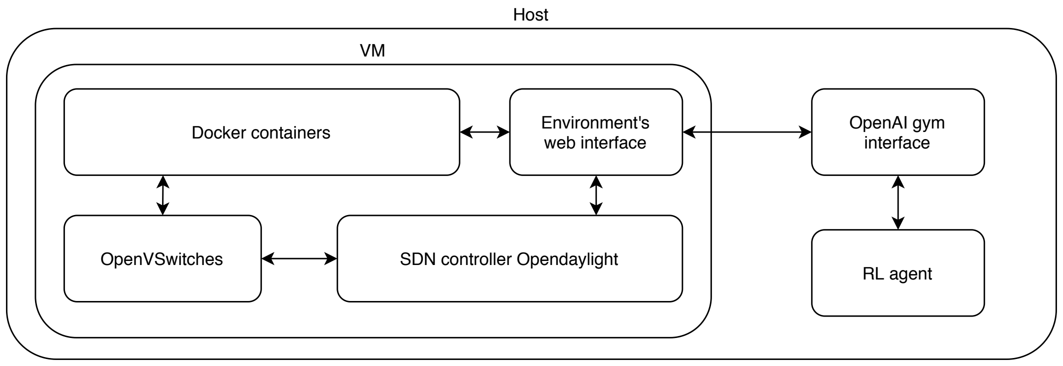


Figure 5: The RL agent training ground.

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 [21]. 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 [22]. 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.

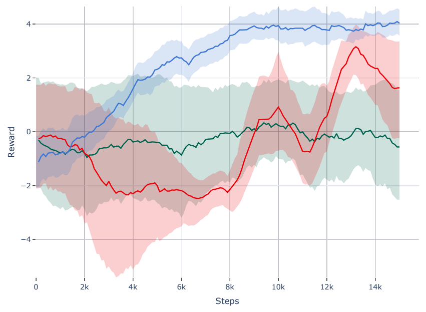
Firewall. We use SDN flows to block suspicious connections by installing rules to drop packets from a particular source host to a particular destination socket that use a particular IP protocol. We push these SDN flows to the switches with two different timeout values: zero, i.e. no timeout constraint, 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.

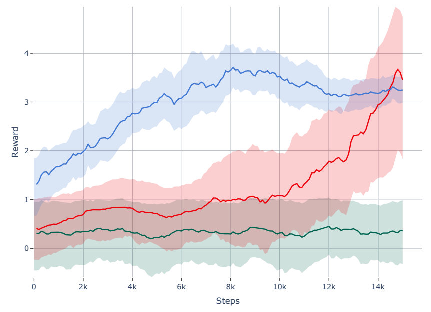
In order to evaluate the framework proposed, we designed a simple Python application that sends a random text message to one of the external data servers. Every arbitrary amount of time (between one and three seconds) each device connects to a randomly selected update server and requests several files 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 [20]. 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.

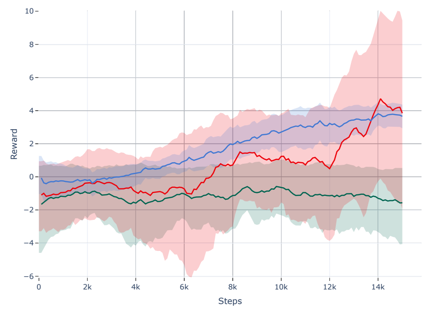
We first train the RL agent using DQN and PPO algorithms with multi-layer perceptron (MLP) as both policy and value function to detect and mitigate the attacks mentioned. In case of DQN, first 80 % of the episodes are used for ϵ-greedy exploration with ϵ value decreasing from one to zero. For PPO, we 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 6 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 6: Performance of DQN (red) and PPO (blue) during different attacks. Green line corresponds to ”do nothing” policy.

4 Conclusion

For this report, we first test several deep learning solutions for the problem of online intrusion detection. After that, several state-of-art reinforcement learning methods are tested using simple environments as well as our custom Docker framework. The results show that such approach can be employed to decrease impact of the attacks in a small private network. All the Python code can be found in [23]. In the next month report, we will present more tests and implementations for both intrusion detection and intrusion mitigation. In addition, we will focus on the problem of realistic network traffic generation with machine learning.

References

[1] J. Kober, J Andrew Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. The International Journal of Robotics Research, 32(11):1238–1274, 2013.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. Proc. of the 25th International Conference on Neural Information Processing Systems (NIPS), Vol. 1, 2012.

[3] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. Proc. of Int. Conference on Machine Learning, pp. 1310–1318, 2013.

[4] S. Hochreiter and J. Schmidhuber. Long short-term memory. Neural Computation, 1997.

[5] F. Gers, N. Schraudolph, and J. Schmidhuber. Learning precise timing with LSTM recurrent networks. Journal of Machine Learning Research, 3:115–143, 2002.

[6] 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.

[7] 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.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. arXiv:1312.5602, 2013.

[9] R. E. Bellman. Dynamic Programming. Dover Publications, Inc., New York, NY, USA, 2003.

[10] C. Watkins and P. Dayan. Q-learning. Machine learning, 8(3-4):279–292, 1992.

[11] T. Lillicrap, J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971, 2015.

[12] 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.

[13] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel. Trust Region Policy Optimization. arXiv preprint arXiv:1502.05477, 2015.

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

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

[16] 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.

[17] 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.

[18] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. Linux journal, 2014(239), 2014.

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

[20] 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.

[21] M. Zolotukhin, T. Hämäläinen, A. Juvonen. Growing Hierarchical Self-organising Maps for Online Anomaly Detection by using Network Logs. Proc. of WEBIST, pp. 633–642, 2012.

[22] 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.

[23] M. Zolotukhin, P. Kotilainen and T. Hämäläinen. LearningAlgorithms. GitLab, https://gitlab.jyu.fi/izi/learningalgorithms, 2020.