ACCEPTED FROM OPEN CALL

ATMoS+: Generalizable Threat Mitigation in SDN Using Permutation Equivariant and Invariant Deep Reinforcement Learning

Hauton Tsang, Iman Akbari, Mohammad A. Salahuddin, Noura Limam, and Raouf Boutaba

In this article, we propose ATMoS+, which extends the RL agent in ATMoS with a novel deep Q-network architecture. The deep RL agent in ATMoS+ leverages permutation-invariant and permutation-equivariant set functions to relax previous assumptions on the number of network hosts and their ordering.

Abstract

Software-defined networking creates new opportunities for automated network security management by providing a global network view and a standard interface for configuring network policies. Previously, we proposed a general framework, called ATMoS, for autonomous threat mitigation using reinforcement learning (RL) in software-defined networks. Using a suitable set of host simulations and based on observations from an arbitrary network monitoring infrastructure, ATMoS can autonomously mitigate threats by moving hosts between a set of virtual networks that embody different network policies. In this article, we propose ATMoS+, which extends the RL agent in ATMoS with a novel Deep Q-Network architecture. The deep RL agent in ATMoS+ leverages permutation-invariant and permutation-equivariant set functions to relax previous assumptions on the number of network hosts and their ordering. We showcase that the proposed deep RL agent is scalable and generalizes to an arbitrary-sized network without additional retraining, scales with the number of hosts, and accommodates several different types of threat alerts.

Introduction

Despite the constantly growing cyber-threat landscape and data breaches for enterprises of all sizes, manual security management remains a de facto standard. On the other hand, recent threat vectors have become more complex and stealthier than ever. They can rapidly evolve to conceal their activities, change behavior over time, and adapt to network dynamics. This adds to the complexity of threat monitoring and response for attacks from advanced actors, such as advanced persistent threats (APTs). Hence, there is a dire need for automation in threat detection and mitigation.

Threat mitigation can be defined as isolating malicious from benign network hosts, and preventing malicious actors from carrying out their operations while ensuring that the benign hosts remain unaffected. This can be easily accomplished in software-defined networking (SDN), which centralizes network control plane functions into dedicated controllers. SDN controllers can add, modify, and delete flow rules in network switches, effectively controlling the entire network. Deploying virtual networks (VNs) within SDNs is also becoming more common, such as in data centers and enterprise networks. Combining SDN with VNs makes mitigating threats more straightforward. By pre-defining different network policies for each VN, an external application can simply send commands to the controller to switch the VN of a host to isolate malicious hosts, achieving threat mitigation in an elegant manner. In contrast, it is very difficult to centrally manage a traditional network. Therefore, SDN and VNs are key enabling technologies to automate threat mitigation.

The final piece is an algorithm to place hosts in the correct VN. This can be accomplished using reinforcement learning (RL), a machine learning (ML) [1, 2] technique that deals with the problem of sequential decision making based on the notion of learning a good behavior by interacting with an environment. While numerous research efforts have focused on ML-based threat detection [3], automated threat mitigation remains relatively uncharted. Previously, we proposed a novel threat mitigation framework, called ATMoS [4], which is based on deep RL in an SDN, and demonstrated its plausibility in a proof-of-concept implementation.

In this article, we extend ATMoS by focusing on the framework's most crucial aspects: scalability and practicality. Notably, we relax assumptions on the number of hosts and their ordering in the training and target networks. This allows a trained deep RL agent to be deployed in networks with different or changing numbers of hosts, which is frequently the case in real-world production environments. To accomplish this, we propose ATMoS+, which addresses these aspects by creating a new architecture incorporating *permutation-invariant* set function, also known as *deep sets* [5], and *permutation-equivariant* set function [6].

Our main contribution is the deep RL agent in ATMoS+, which:

- Is robust to change in input ordering, allowing accommodation for real-world environments with changing host identifiers
- Is scalable as the number of neural network trainable parameters are not dependent on the number of network hosts
- Generalizes to arbitrary-sized networks,

Digital Object Identifier:
10.1109/MCOM.009.2100389

The authors are with the University of Waterloo.

0163-6804/21/$25.00 © 2021 IEEE

allowing for deployment in real-world environments with dynamic numbers of hosts
- Performs well in larger networks with several different types of threat alerts

The source code of ATMoS+ is available online [7].

Background and Related Work

Supervised and unsupervised ML have been leveraged for threat hunting, detection, and mitigation (e.g., [8, 9]). However, the use of RL for cybersecurity is relatively new, especially when it comes to active mitigation. The primary advantage of RL is *sequential* decision making, making it more powerful than supervised learning. However, defining threat mitigation as an RL problem is far from trivial, and there are many considerations on what should constitute various RL components. In this section, we provide a brief context on RL and its application to threat mitigation.

Reinforcement Learning

RL comprises an agent, a set of actions, an environment, and a reward function. The agent *observes* the environment to read its *state*. Based on its observations and internal state, the agent chooses an *action* out of a set of all possible actions. Once the action is carried out in the environment, it alters the environment's state and the agent receives a *reward*, which is used to adjust its internal state for future actions. The goal is to produce the highest expected *cumulative* reward, which allows the RL agent to master a *sequential* decision making problem. At each point in time, the RL agent does not simply realize the highest immediate reward, but rather foresees possibilities created in future steps and makes decisions accordingly.

One of the most basic RL algorithms is Q-learning. In Q-learning, the goal is to estimate the expected cumulative reward, or *Q-value*, for taking each possible action a in a given state s. To estimate these Q-values, the agent executes actions to explore various states, and based on the received reward, updates its Q-value estimation for the relevant state-action pairs. This process is typically governed by a discount factor (γ), which sets the trade-off in prioritizing *future rewards* over immediate rewards, and the exploration rate (ε), which controls how often the agent takes random actions to explore new possibilities, and this decreases as training progresses. Once training is complete, the RL agent only needs to pick the action with the highest Q-value for each state to achieve the highest expected reward. However, Q-learning requires memory space for storing Q-values of every state-action pair, making it infeasible for complex problems. To solve this, a neural network function can be used to estimate the Q-values instead. These solutions are known as deep RL algorithms.

Deep Q-Network (DQN) is one such algorithm, which approximates the Q-value estimation table using a neural network. It takes the current state as input and outputs Q-values for each action. The action with the highest Q-value is taken at each state, and once the reward is obtained, the neural network weights are updated accordingly. Over the years, many variations of the basic DQN algorithm have been introduced, such as Double DQN and Dueling DQN [10]. Other examples of deep RL algorithms include Deep Deterministic Policy Gradient (DDPG), Proximal Policy Optimization (PPO), and Asynchronous Advantage Actor-Critic (A3C) [10], which are extensively used in RL literature.

RL for Threat Mitigation in SDN

The application of RL to threat mitigation is an active field of research. SDN facilitates threat mitigation by providing a centralized network view and allowing dynamic update of network policies.

Liu *et al.* [11] proposed a framework for detecting and mitigating distributed denial of service (DDoS) attacks using the DDPG algorithm. Their deep RL agent uses statistical information gathered from each SDN switch to generate a vector of bandwidth limits for each host. Normally, in an RL algorithm, the number of actions is fixed. However, in this case, there are an infinite number of possible bandwidth limits, and the use of DDPG allows the agent to choose one of these infinite possible actions. Nevertheless, this framework is DDoS-specific, and it is difficult to generalize to other types of attacks.

Zolotukhin *et al.* [12] have proposed a general framework for attack mitigation that incorporates anomaly detection as well as signature-based intrusion detection system (IDS) alerts. They evaluated the use of both the DQN and PPO algorithms. The input to their deep RL agent is a list of statistics for each traffic flow, such as number of unique ports, number of alerts detected, and number of requests in the flow. The agent can then take actions based on these traffic flows. However, the specific actions supported by the agent are not elaborated.

Han *et al.* [13] investigated improving the robustness of using RL for threat mitigation by incorporating adversarial training. They showed that RL-based approaches can be susceptible to vulnerabilities. For example, an attacker may compromise the network observer to send a false network state to the agent, but the agent can learn to mitigate these effects using adversarial training techniques.

One of the main issues with these works is that the *sequential* aspect of the mitigation problem, which is the primary motivation for using RL, is often missing. Defending against stealthy and complex threat vectors (e.g., APTs) entails observing a host's behavior over a relatively long time period and studying how different decisions affect a host's behavior.

Automated Threat Mitigation in SDN: ATMoS

In this article, we extend the ATMoS framework, which assumes that a number of hosts in an SDN environment have been compromised. The goal is to identify these hosts and impose appropriate policies, via the SDN controller, to block their operations. However, there are benign hosts within the network as well, and their operations should not be affected by the imposed policies. For a realistic environment, it is also assumed that the security monitoring systems in place may result in false positive alerts.

The ATMoS framework leverages multiple VNs with different network policies, which are deployed on top of the existing network prior

The primary advantage of RL is sequential decision making, causing it to be more powerful than supervised learning. However, defining threat mitigation as an RL problem is far from trivial, and there are many considerations on what should constitute various RL components.

The deep RL agent in ATMoS is based on a static neural network, that is, the number of hosts and their ordering must remain constant in training and deployment. This limits the transferability of the agent to arbitrary networks. Ideally, we should be able to train the agent in one network, such as a staging environment, and deploy it anywhere.

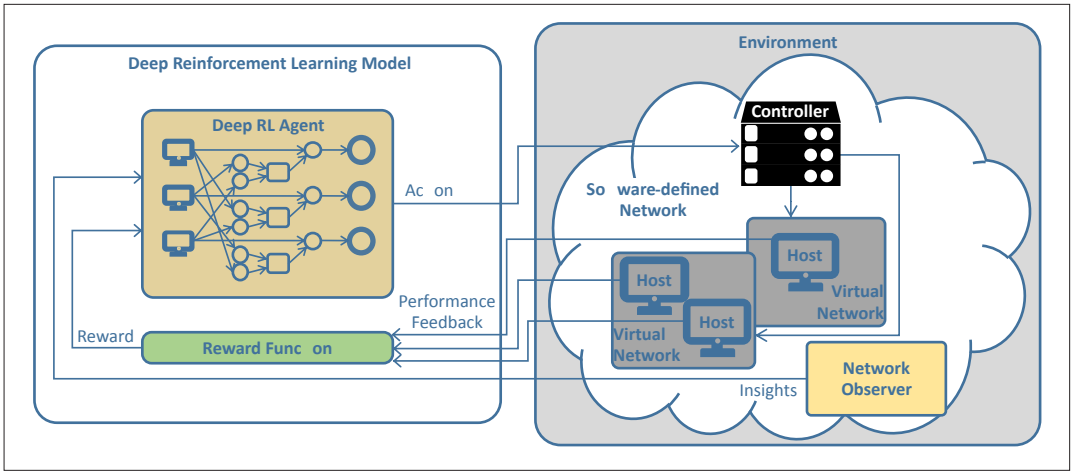

FIGURE 1. ATMoS architecture.

to running the agent. These VNs are designed by domain experts to embody different *security levels* or *potential policies toward network hosts*. For instance, a network can have two VNs, where one VN passively monitors traffic while the other blocks traffic upon alerts from the security monitoring systems. While different VN designs can be explored in the future, the core idea is to steer the RL agent's actions to place a network host in a particular VN in order to control the size of the action space.

ATMoS leverages deep RL to decide on which VN each network host should be placed. The deep RL agent receives the alerts from an arbitrary security monitoring system as input, and accordingly decides whether it should change a host's VN. Using a suitable reward function, the agent learns to place the malicious and benign hosts in their appropriate VNs.

The architecture of ATMoS is depicted in Fig. 1. The security monitoring system is depicted as the *network observer*, which monitors the network traffic and produces alerts in real time. The network observer transmits the alert data from the environment to the deep RL agent in a standard format. The network observer is assumed to accurately construct alerts matching user-defined rules. However, the rules themselves may be overly sensitive, as is often the case in real-world environments.

The agent is implemented using a DQN, with a neural network model consisting of two dense fully connected layers with rectified linear unit (ReLU) activation functions. It receives the alerts and the current VN placement of all the hosts, and decides whether to pick a host and move it to a different VN in a predefined order or do nothing.

Many different performance feedback metrics can be used to train ATMoS in production, including response time and host uptime metrics. Alternatively, simulated malicious and benign hosts can be directly labeled so that the reward depends on the placement of hosts in VNs. Thus, by defining the appropriate reward function and VNs, ATMoS can be applied not just to mitigate DoS attacks, but other threat vectors, even APTs. After the RL agent is trained based on simulations, it can be deployed in the target network, where the hosts are real users, and it is unknown whether they are malicious or benign.

ATMoS+

Motivation

The deep RL agent in ATMoS is based on a static neural network, that is, the number of hosts and their ordering must remain constant in training and deployment. This limits the transferability of the agent to arbitrary networks. Ideally, we should be able to train the agent in one network, such as a staging environment, and deploy it anywhere. Therefore, to facilitate cross-network use cases, we propose ATMoS+, an extension to ATMoS, with a novel deep RL agent architecture that supports training and deployment on arbitrary-sized networks.

The DQN model in ATMoS is not readily generalizable when the number of network hosts changes. Since the model learns host-specific weights that capture the characteristics of particular hosts, the model must be retrained to learn the appropriate weights for new hosts. Furthermore, the input and output shapes of the DQN model also depend on the number of hosts. Therefore, modifying the number of hosts requires changes to the shape of the input and output layers before the model undergoes retraining. Since additional training is required, changing VNs of randomly selected hosts occurs when the agent is exploring new policies. In a production environment, servers can be added or removed frequently, so the agent will need to be retrained frequently as well. This causes VNs to be repeatedly toggled for random hosts, which is impractical. Additionally, both the training time and the neural network complexity grow super-linearly with the number of hosts.

Moreover, the DQN model in ATMoS can become sensitive to the host ordering; that is, it is susceptible to binding to host IDs. The model comprises dense neural network layers that take a vector of alert observations from all hosts as the input in a fixed order. Hence, neurons corresponding to different hosts in the neural network can have a different set of weights, so the model can, for example, learn that the second host is always malicious. However, if the network host order changes, the neurons will no longer correspond to the correct hosts, confusing the deep RL agent. To alleviate this, the host order can be randomized every training episode. However, this comes at the cost of a much larger number of

training episodes, substantially increasing training time.

These issues call for enhancing the neural network architecture such that it allows for a *dynamic* number of network hosts. The neural network should also be robust to the order in which a host's data appears in the input feature vector. In ATMoS+, we address these issues by using *set functions* that treat inputs as unordered sets. This prevents the DQN model from implicitly depending on host ordering. Furthermore, due to the structure of set functions, they can easily adapt to a different (i.e., higher or lower) number of hosts. ATMoS+ leverages the same architecture as ATMoS, as depicted in Fig. 1, but includes the new deep RL agent to improve generalization.

Set Functions

There are two major types of set functions: permutation-equivariant and permutation-invariant. In permutation-equivariant set functions, the input ordering is directly correlated with the output. Although there is no particular ordering in the function's input, the output for the corresponding input is always in the same relative position. For example, the output corresponding to the second input will always be the second output. If the first and second inputs are swapped, the first and second outputs swap as well, as shown in Fig. 2.

In permutation-invariant set function, the ordering of the input has no effect on the output. In this case, unlike permutation-equivariant function, the outputs of a permutation-invariant function are tied to the general state of all the inputs, not to a specific input. This is also illustrated in Fig. 2, where the output does not change when the inputs are swapped.

Both permutation-equivariant and permutation-invariant set functions can be approximated using respective neural network architectures [5, 6]. Both of these neural networks rely on functions that we denote as P and R. The P function creates a high-dimensional summary vector from the data of a particular input. These summary vectors are then pooled together using a pooling function, such as the element-wise maximum, average, or summation. The same P function is used on all inputs, so once the pooling operation is complete, it is impossible to infer the original order of the hosts. Finally, an R function takes this pooled vector as an input to produce the desired output. Both the P and R functions can be approximated using neural networks without any specific architectural constraints. We use the P and R functions to model permutation-equivariant and permutation-invariant set functions as follows.

Permutation-Invariant Modeling: To model a permutation-invariant set function with a neural network, the P function is applied to all the inputs, and the resulting vectors are pooled together. Finally, the R function is applied to the pooled result to obtain a permutation-invariant output.

Permutation-Equivariant Modeling: To model a permutation-equivariant set function with a neural network, each input must be considered individually. For a particular input, the P function is applied to all the other inputs and pooled together. Then both the data from the current input and the pooled vector are fed to the R function. The output of the R function is the output for the corresponding input. The outputs of the R function for all the inputs is the output of the model.

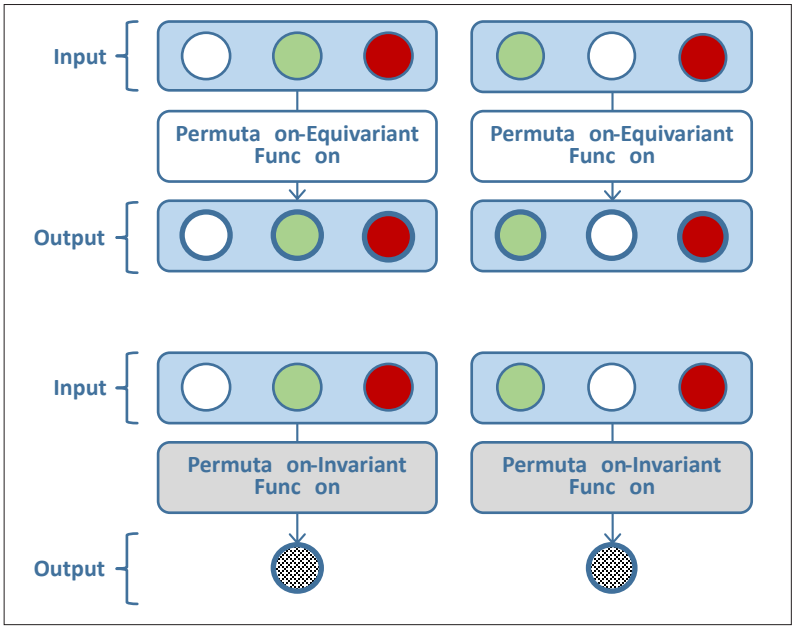

FIGURE 2. Input, output ordering property of permutation-equivariant and permutation-invariant set functions.

By integrating these neural network architectures into the deep RL agent in ATMoS+, we obtain a DQN model that treats hosts as a set and is not influenced by any host order in particular. Furthermore, the neural network architectures can easily be extended to networks of any size, since all hosts share the neural network parameters.

Set Functions in ATMoS+

The DQN model in ATMoS+, however, is neither a permutation-equivariant nor permutation-invariant set function. It is nearly a permutation-equivariant set function, as each input has a corresponding output. If we swap the ordering of the hosts, we would like to output corresponding Q-values associated with each input host. However, the number of inputs does not exactly correspond to the number of outputs, because there is one extra output that does not belong to any host, which is the "do nothing" action. The Q-value for "do nothing" is dependent on the state of all the hosts, as nothing needs to be done if and only if all the hosts are in the correct VNs. Therefore, the Q-value of the "do nothing" action corresponds to a permutation-invariant set function. Thus, to implement the DQN model, both permutation-equivariant and permutation-invariant set functions are necessary.

The neural network architecture of the DQN model is shown in Fig. 3. The input has two parts: vector of alert observations for each host, and current VNs of the hosts. The functions P_1 and P_2 correspond to the P functions used in the permutation-equivariant and permutation-invariant neural networks, respectively. They are both approximated using a neural network consisting of two dense layers with 12 and 16 neurons using ReLU activation functions. These P functions take a vector of alerts from one host and the current VN of that host as the input. We use element-wise maximum as the pooling function.

Similarly, the functions R_1 and R_2 correspond to the R functions used in the permutation-equiv-

ariant and permutation-invariant neural networks, respectively. R_1 consists of a dense layer with eight neurons (ReLu activation) followed by a dense layer with a single neuron with linear activation. The network input consists of the corresponding pooled vector concatenated with the VN and alert information of the host. R_2 consists of a single dense layer with one neuron using a linear activation function. Its input is the pooled vector from the permutation-invariant functions. The number of neurons and activation functions in the neural networks are chosen based on trial and error.

Additional Hosts after Training

Note that the P function for all the P_1 nodes in Fig. 3 are the same. These P functions are a shared layer between all three inputs. This is also the case for P_2, R_1, and R_2. In other words, all the P_1 nodes have exactly the same weights, all the P_2 nodes have exactly the same weights, and so on. This makes it trivial to alter the neural network to accommodate a larger or smaller number of hosts.

To add a host to the neural network, we simply extend both the permutation-invariant and permutation-equivariant functions. To extend the latter, we set the alert observations and VN status of the new host as the input to P_1 and obtain a summary vector. This summary vector can then be added as an input to the pooling function of all the other inputs. Next, we take the summary vectors of all the other hosts and set them as the input to the pooling function of the new host to obtain a pooled vector. Finally, we set the pooled vector, the alert observations, and VN status of the host as the input to R_1. The output of R_1 is the output corresponding to the new host.

To extend the permutation-invariant function, we map the data from the new host to P_2, which is then added as an input to the pooling function. To remove a node, these steps can be carried out in reverse. Therefore, in addition to the DQN model being independent of the host input position, it can also be deployed in a network with a different number of hosts than the training network.

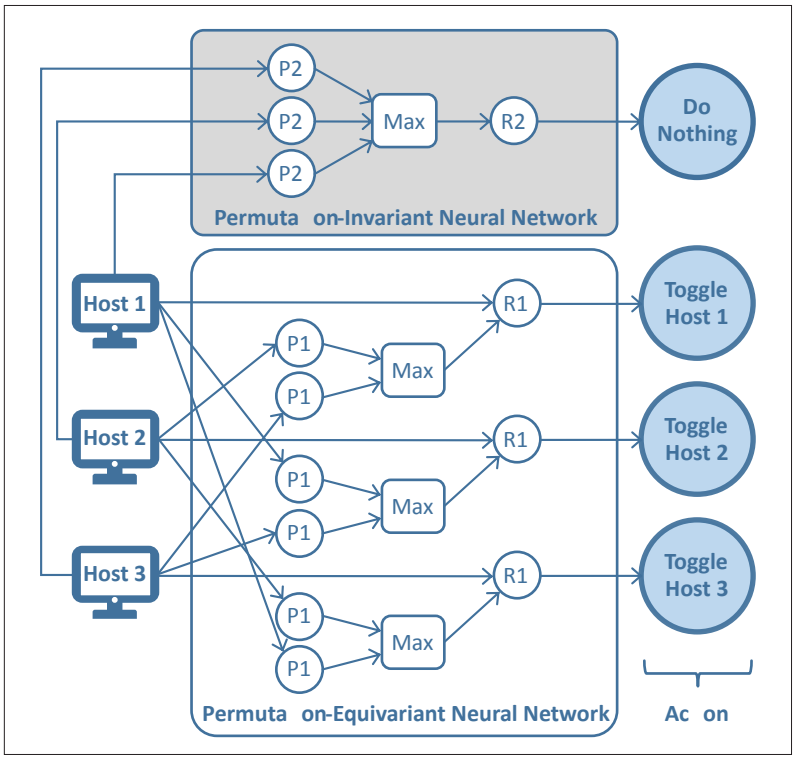

FIGURE 3. DQN's neural network architecture: P_1, P_2, and R_1, R_2 correspond to two different P and R functions for the permutation-invariant and permutation-equivariant functions.

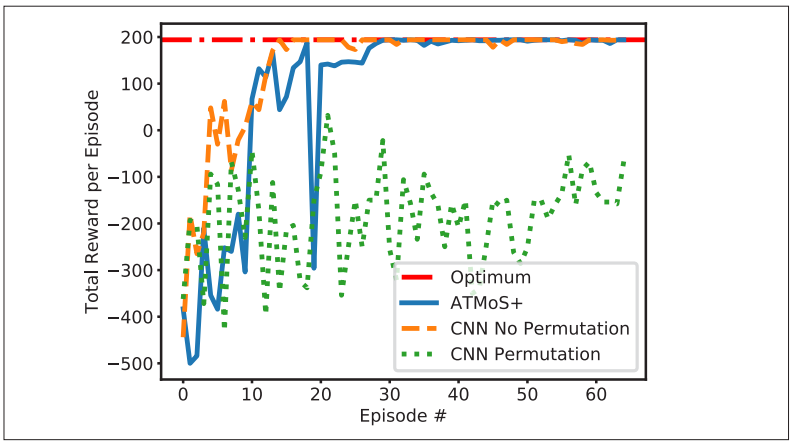

FIGURE 4. Training with permuted CNN vs. non-permuted CNN vs. ATMoS+.

Evaluation

To evaluate ATMoS+, we set up an SDN with an OpenDaylight controller on Containernet, a network emulator that uses Docker containers as hosts. OpenDaylight's Virtual Tenant Network plugin is used to implement the VNs. We use Open vSwitch as the switches in the SDN data plane. Finally, the benign and malicious hosts are implemented using a set of scripts running within the Docker containers. The deep RL agent itself is implemented using Tensorflow Keras.

Training Convergence: Set Functions vs. Non-Set Functions

We trained ATMoS+ on a network of 10 hosts, that is, six benign and four malicious hosts. We leverage two VNs in our evaluation:

- Low-security VN with passive monitoring, denoted as security level 1
- High-security VN with active interception, denoted as security level 2

The reward function is chosen as the sum of the VN security levels of all the malicious hosts subtracted by the sum of the VN security levels of all the benign hosts. The input to the RL agent is a one-hot encoded vector of four types of alerts for each host:

- A SYN flood detector
- A false positive alert that can be triggered by excessive pinging
- An error-based SQL injection detector
- An alert triggered by a flood of HTTP traffic

The agent's exploration rate (ε) and discount factor (γ) parameters are set to 0.1 and 0.5, respectively. The alerts from the host are obtained as a one-hot encoded vector of the last 20 alerts. The training runs for 100 steps per episode for 65 episodes. After the end of each episode, the hosts are reset; that is, they are moved to the low-security VN.

The training convergence of the deep RL agent with the DQN model and four alerts is depicted in Fig. 4. This training convergence is

compared against the best-performing non-set function model, that is, a convolutional neural network (CNN) with a kernel size of 3. The CNN model converges slightly faster than the DQN model. We attribute this to the CNN learning the malicious hosts based on the position of each host in the input. However, if the positions of hosts are rearranged after every episode, the CNN model fails to learn the optimal placement of hosts. This demonstrates that the set function model in ATMoS+ is robust to permutations in host ordering.

Additional Alert Types

We also evaluated the scalability of training the DQN model in ATMoS+ with a varying number of alerts up to a maximum of 12. Seven alerts are network-level alerts, and include detection of SYN attacks, abnormal HTTP traffic, and excessive pinging. The remaining five alerts are application-level alerts, which detect SQL injection, directory traversal, and buffer overflow exploits. We deliberately increased the sensitivity of some alerts to generate false positives. We also implemented three additional Docker containers: a docker container running DVWA, a deliberately insecure web application, a malicious container executing SQL injection, and a malicious container executing directory traversal and buffer overflow. The result in Fig. 5 shows that the deep RL agent in ATMoS+ still converges with the addition of new alerts. However, the convergence time is longer for 8 and 12 alerts in comparison to 4 alerts. We attribute this to the increase in the DQN model parameters, which increases with the number of alerts. We have also evaluated different combinations of four alerts, and found that ATMoS+ performs consistently across all combinations.

Deploying Trained Agents in Arbitrary-Sized Networks

The ATMoS+ agent was tested in networks with arbitrary sizes. The procedure to reconstruct the DQN model for different network sizes was described previously. We trained the deep RL agent on a small network with 10 hosts and evaluated its performance by reconstructing the DQN model for larger networks with sizes ranging from 20 to 100 hosts, with 5 percent of the hosts being malicious. Indeed, the agent can theoretically be deployed in much larger networks.

For each network, we execute the deep RL agent for 100 steps. The agent starts off in a baseline state, where all the hosts are placed in the low-security VN. Then we run the RL agent purely on the greedy policy, that is, choosing the action that maximizes the reward on every step. Figure 6 shows how ATMoS+ places the hosts in the optimal VN configuration immediately. Since the reward function we use changes domain depending on network size, for simplicity we transformed the reward function to show the number of misplaced hosts instead. This demonstrates that the deep RL agent with the DQN model in ATMoS+, which is based on set functions, generalizes to arbitrary-sized networks without retraining.

Conclusion

In this article, we propose ATMoS+, an extension to ATMoS, which leverages a novel DQN with permutation-invariant and permutation-equivariant set functions to facilitate threat mitigation across arbitrary-sized SDNs. We showcase that the deep RL agent in ATMoS+ is scalable, accommodates a larger number of alerts, and generalizes to an arbitrary-sized network without additional retraining. This opens the door to future applications, where a pre-trained deep RL agent could be deployed to mitigate threats from many different networks.

An important future direction is to enable the DQN model to learn long-term dependencies for each host. For example, if a malicious host ceases malicious activity for a long time, the DQN model is unable to retain this knowledge, as it relies solely on alerts detected from the host's activity to determine malice. Furthermore, the agent in ATMoS+ uses a one-hot encoded vector of the last 20 alerts to make decisions, which may result in alerts being missed if more than 20 alerts occurred within the last time step. Although this number can be increased to adapt to alert frequencies in different environments, a generalizable alert representation could be explored that incorporates information from all alerts, regardless of how frequently alerts occur. ATMoS+ must also be extended to mitigate zero-day attacks. This could potentially be accomplished by integrating the output of an anomaly detector as an alert type.

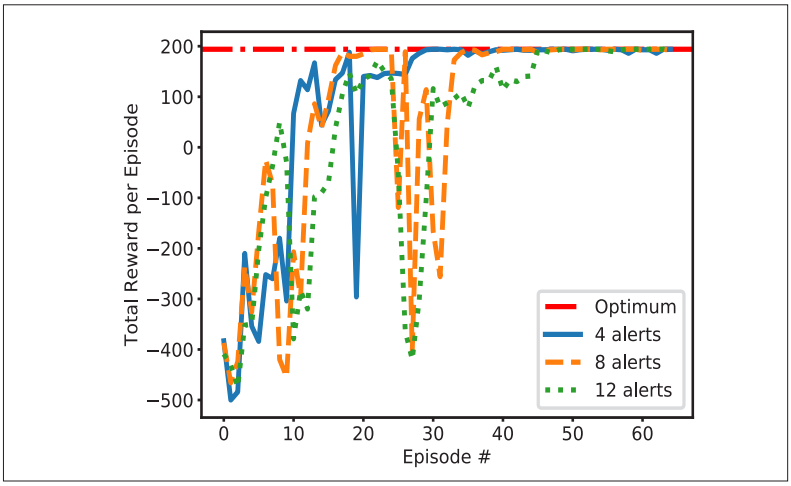

FIGURE 5. Training ATMoS+ with a varying number of alerts.

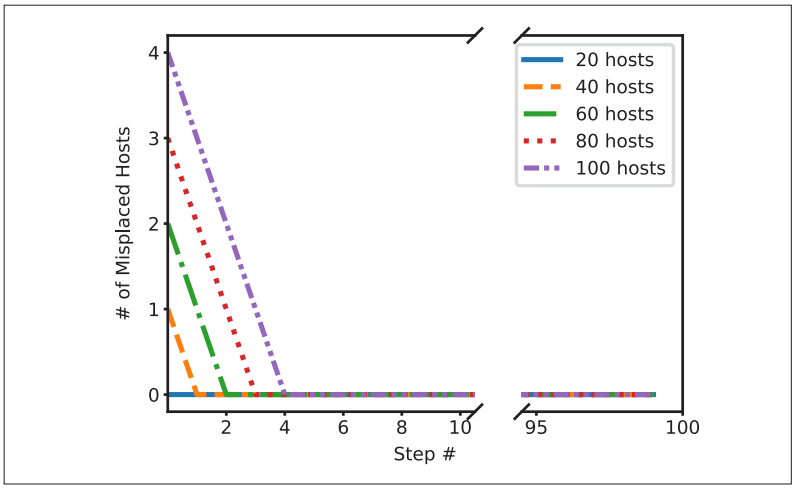

FIGURE 6. Threat mitigation for varying network sizes.

Acknowledgments

This work is supported in part by the Royal Bank of Canada, and in part by NSERC CRD Grant 530335.

References

[1] R. Boutaba *et al.*, "A Comprehensive Survey on Machine Learning for Networking: Evolution, Applications and Research Opportunities," *J. Internet Services and Applications*, vol. 9, no. 1, 2018, pp. 1–99.
[2] S. Ayoubi *et al.*, "Machine Learning for Cognitive Network Management," *IEEE Commun. Mag.*, vol. 56, no. 1, Jan. 2018, pp. 158–65.
[3] P. Mishra *et al.*, "A Detailed Investigation and Analysis of Using Machine Learning Techniques for Intrusion Detection," *IEEE Commun. Surveys & Tutorials*, vol. 21, no. 1, 2018, pp. 686–728.
[4] I. Akbari *et al.*, "ATMoS: Autonomous Threat Mitigation in SDN Using Reinforcement Learning," *Proc. IEEE/IFIP Network Operations and Management Symp.*, 2020.
[5] M. Zaheer *et al.*, "Deep Sets," *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates, Inc., 2017.
[6] A. Sannai *et al.*, "Universal Approximations of Permutation Invariant/Equivariant Functions by Deep Neural Networks," arXiv preprint arXiv:1903.01939, 2019.
[7] "ATMoS+ Source Code"; https://github.com/ATMoS-Waterloo/ATMoS/ tree/atmosplus, accessed Oct. 14, 2021.
[8] N. N. Tuan *et al.*, "A Robust TCP-Syn Flood Mitigation Scheme Using Machine Learning Based on SDN," *Proc. Int'l. Conf. Info. and Commun. Technology Convergence*, 2019, pp. 363–68.
[9] S. S. Bhunia and M. Gurusamy, "Dynamic Attack Detection and Mitigation in IoT Using SDN," *Proc. Int'l. Telecommun. Networks and Applications Conf.*, 2017.
[10] V. François-Lavet *et al.*, "An Introduction to Deep Reinforcement Learning," *Foundations and Trends in Machine Learning*, vol. 11, no. 3–4, 2018, pp. 219–354.
[11] Y. Liu *et al.*, "Deep Reinforcement Learning Based Smart Mitigation of DDoS Flooding in Software-Defined Networks," *Proc. IEEE Int'l. Wksp. Computer Aided Modeling and Design of Commun. Links and Networks*, 2018.
[12] M. Zolotukhin *et al.*, "Reinforcement Learning for Attack Mitigation in Sdn-Enabled Networks," *Proc. IEEE Conf. Network Softwarization*, 2020, pp. 282–86.
[13] Y. Han *et al.*, "Reinforcement Learning for Autonomous Defence in Software-Defined Networking," *Decision and Game Theory for Security*, Springer, 2018, pp. 145–65.

Biographies

Hauton Tsang is a graduate student at the University of Waterloo. He received his B.Sc. degree in computer science from Hong Kong University of Science and Technology. His research interests are focused on applying machine learning in the field of cybersecurity.

Iman Akbari received his B.Sc. from Sharif University in software engineering. He is currently a graduate student at the University of Waterloo. His research mostly revolves around the intersection of AI, cybersecurity, and network management with a focus on automation and scalability.

Mohammad A. Salahuddin is a research assistant professor of computer science at the University of Waterloo. He received his Ph.D. degree in computer science from Western Michigan University in 2014. His current research interests include the Internet of Things, content delivery networks, network softwarization, network security, and cognitive network management. He serves as a TPC member for IEEE conferences, and is a reviewer for various journals and magazines.

Noura Limam received her M.Sc. and Ph.D. degrees in computer science from the University Pierre & Marie Curie, Paris VI, in 2002 and 2007, respectively. She is currently a research assistant professor of computer science at the University of Waterloo. She is on the Technical Program Committees and Organization Committees of several IEEE conferences. Her contributions are in the area of network and service management. Her current research interests are in network softwarization and cognitive network management.

Raouf Boutaba [F] received his M.Sc. and Ph.D. degrees in computer science from Sorbonne University in 1990 and 1994, respectively. He is currently a University Chair Professor and the director of the David R. Cheriton School of Computer Science at the University of Waterloo. He also holds an INRIA International Chair in France. He is the founding Editor-in-Chief of *IEEE Transactions on Network and Service Management* (2007–2010) and the current Editor-in-Chief of the *IEEE Journal on Selected Areas in Communications*. He is a Fellow of the Engineering Institute of Canada, the Canadian Academy of Engineering, and the Royal Society of Canada.