Towards Self-Defense of Non-Stationary Systems

Stefano Iannucci, *Member, IEEE*, Andrea Montemaggio, Byron Williams, *Member, IEEE*

Abstract—One of the major trends in research on Intrusion Response Systems is to use a model of the system to be protected and/or a model of the attacker to predict the evolution of the system and of the strategy of the attacker. However, very often, modeled systems exhibit a non-stationary behavior due to changes in their configuration, in the software base and in the users behavior. If not properly captured by the system model, such a non-stationary behavior could lead to divergences between the expected and the actual behaviors, thus invalidating the model-based approach. In this paper, we introduce a model-free technique for self-defense of non-stationary systems based on Q-Learning. We experimentally show that the proposed approach is able to effectively capture the dynamics of the underlying system and quickly adapts to changes in the environment.

Index Terms—Intrusion Response System, Autonomic Security Management

1 Introduction

Securing an organization's IT infrastructure against cyber-attacks is an activity of paramount importance. This endeavor can be challenging due to system size and attack surface of the network (i.e., the numerous potential sources of exploit attempts). Organizations with limited cyber-security expertise face additional risk. Determining if an attack has taken place, responding to it appropriately and in a timely manner is a hard problem even for the most advanced cyber-security professionals. Identifying anomalies, characterizing malicious behavior, detecting malware, and responding appropriately are all activities seasoned cyber-security professionals are challenged with.

In the past decade, there has been considerable research focused on Intrusion Detection [17]. While important, these results leave the system administrator the task of manually responding to the detected attacks. However, any non-automated attempts at system defense could cause delays, which in turn provide the attackers more time to reach their objectives [4]. For this reason, we envisage an Autonomic Security Management (ASM) system designed to cover both detection and response, and we discuss in this paper the methodology underlying the response selection mechanism in presence of a system with non-stationary behavior.

The ASM structure, outlined in Figure 1, is composed of two main parts: a controller that implements the self-protection algorithms with the supporting modules and the controlled system. The former is represented by the Autonomic Security Management block, while the latter is represented by the Enterprise System block. The architecture of the ASM follows the Monitor, Analyze, Plan, Execute (MAPE, [12]) loop for autonomic systems. The Monitor phase leverages existing monitoring tools already deployed in the Enterprise System, and produces streams of events that are forwarded to the *Analyze* phase. The latter is in charge of (i) filtering the incoming streams, (ii) correlating them and (iii) deciding which events constitute a threat for the system. When a threat is found, the *Plan* phase is activated. The *Plan* is in charge of dynamically composing a sequence of actions that are designed to protect the system in response to the detected threat. The *Execute* phase is activated afterwards, along with the optional component *Decision Support Tool* that presents the decision to the system administrator and will wait for acknowledgement. Subsequently, all the defense actions are pushed to the proper delivery queue and executed on the enterprise system.

- S. Iannucci is with the Department of Computer Science and Engineering, Mississippi State University, Starkville, MS.
- A. Montemaggio is with the Center for Cyber Innovation, Mississippi State University, Starkville, MS.
- B. Williams is with the Department of Computer & Information Science & Engineering, University of Florida, Gainesville, FL.

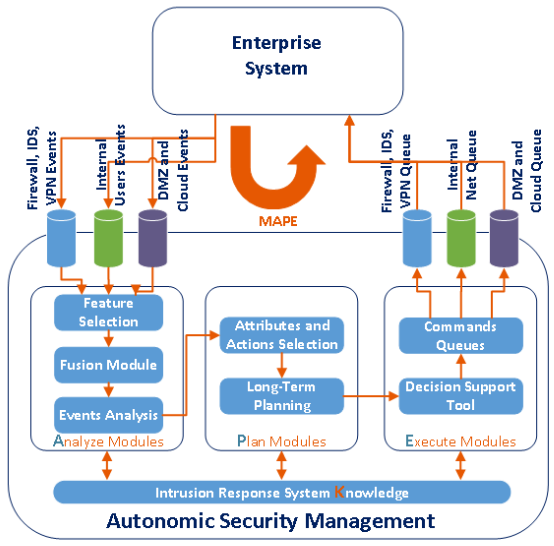

Fig. 1: Organization Network secured with ASM

To the best of our knowledge, most of the works proposed in the field of automatic intrusion response deal either with the model of an attacker (e.g., [13]), or with the model of a defender (e.g., [5]) or, in some case, they propose a joint model (e.g., [18]). However, none of them address the problem of a dynamic environment. In a previous work [8], we introduced an approach based on planning over a Markov Decision Process (MDP [2], [15]) for the computation of optimal system defense strategies. However, even though the MDP-based approach is in theory sufficient to protect a system that behaves exactly as modeled, the effects of the defense actions executed on the system might change over time. Furthermore, new actions might become available and others might disappear. These situations can arise due to different reasons, including shifts in system configurations, software updates, and changes in user behavior. One way to address this issue is to monitor the execution of the actions and to update the parameters of the model, for instance by using filters like Exponential Weighted Moving Average (EWMA, [7]). However, this approach would require the re-execution of a computationally expensive planning every time a change is detected. With the adoption of certain heuristics as described in [9], and considering the massively parallel implementation proposed in [10], this scenario would be feasible for small to medium sized systems. However, it remains infeasible for larger systems due to the exponential nature of the problem.

For these reasons, we investigate the implementation of a self-adaptive planner that leverages a model-free reinforcement learning approach [15]. The main idea is to let the planner automatically evolve as the target system does without the need to run a computationally expensive planning phase when a change to the system occurs. However, one of the main issues with model-free agents is the time needed to learn from the environment or, in other words, in a discrete setting, the time steps needed to converge to a near-optimal solution. In order to accelerate the training phase, we plan on using a hybrid approach, where the agent is first trained on a simulated environment based on an approximated model of the system (developed using prior expertise obtained from experimentation). When its cumulative reward is satisfactory, it is detached from the simulator and attached to the real system. The proposed approach has a twofold advantage over the model-based one: on one hand, it will avoid the computationally expensive planning, thus reducing the time-to-protection; on the other hand, by supporting and automatically evolving approximated models, it will not require any additional intervention from the system administrator to change the initial model. Experimental results show that an agent starting with zero knowledge can effectively capture the dynamics of the underlying system and can quickly react to events, such as, the addition, removal, and change of configuration of the defense actions.

The paper is organized as follows: Section 2 introduces the mathematical framework that we use to model the system behavior and describes the reference system model; in Section 3 we discuss the testbed and the experimental results; related work is discussed in Section 4. Finally, in Section 5 we draw the conclusions and hint at future work.

2 Methodology and Model

We use a common mathematical framework for reinforcement learning: the Markov Decision Process (MDP). MDP is a probabilistic approach to model the stochastic behavior of a system. It can be formalized by the tuple $\langle S, A, P, T, R, \gamma \rangle$. The symbol S represents the state space that an agent can navigate and $s_k \in S$ represents the agent state at discrete time k. A is the finite set of actions available to the agent to navigate the state space. Specifically, by executing at time k an action $a \in A$ in the current state $s_k \in S$, the agent moves to a successor state $s_{k+1} \in S$. The transition dynamics from the current to the next state are given by the transition probability function P. This function specifies for each source state $s_k \in S$, for each destination state $s_{k+1} \in S$, and for each action $a \in A$, the value $P(s_k, a, s_{k+1})$, that is, the probability that by executing the action a in state s at time k, the resulting state will be s_{k+1}. $T \subseteq S$ is the subset of the final states of the MDP that corresponds to the system's *secure region*.

Every time an action is executed, the MDP agent is rewarded with a bonus (or penalized with a cost), according to the reward function R. That is, $R_k = R(s_k, a, s_{k+1})$ represents the reward that the agent will earn (or the cost the agent will pay) for executing at time k the action a in state s_k and being taken to some state s_{k+1}. MDPs are also characterized by a discount factor, γ, which specifies the preference of short-term rewards over long-term ones.

The overall behavior of the agent is described by a policy π that specifies a probability distribution such that $\pi : S \times A \to [0, 1]$. That is, $\pi(a|s_k)$ represents the probability that the agent will execute action a while in state s at discrete time k. The objective is to find a policy π^* such that the discounted reward $R_k = \sum_{j=0}^{\infty} \gamma^j R_{k+j+1}$ is maximized.

Optimal and sub-optimal algorithms for solving MDPs have been proposed (e.g., [2], [11]). However, a combination of the *Value Iteration* (VI, [2]) and *Policy Iteration* (PI, [15]) algorithms is commonly used because of their simplicity.

The aforementioned algorithms fall into a category of MDP solvers named *planners* and are *model-based*. These solvers assume complete knowledge of all the elements of the MDP, and some of them (i.e., VI and PI) are able to provide optimal solutions, in terms of achievable cumulative reward. However, in a typical reinforcement learning problem, the transition function P and the reward function R are generally unknown, preventing the problem from being directly solved with a planner. *Model-free* techniques, such as, Q-Learning [16], SARSA [15], and Artificial Neural Networks (ANNs) [14] use an approach based on trial-and-error interactions with a dynamic environment to learn the behavior of the system. In this work, we use Q-Learning with an intention to create a baseline for future comparison of other algorithms.

Q-Learning is a Temporal-Difference approach to estimate the action-value function of an MDP, defined by:

$$Q(s_k, a) \leftarrow Q(s_k, a) + \alpha[R_{k+1} + \gamma \max_a Q(s_{k+1}, a) - Q(s_k, a)]$$

The current state-action value Q, also known as *q-value*, represents the expected discounted return that the agent can earn starting from s_k and executing a. The update rule defined by Q-Learning is constituted by two components: a prior knowledge acquired by interacting with the system up to time k, and the current knowledge, acquired as temporal difference with respect to the previous value. The α coefficient is known as the *learning rate*: the higher is the value of α, the higher the importance that is given to the current learning experience with respect to the past ones. Q-Learning has been shown to converge with probability 1 to Q^*, the optimal state-action value function.

2.1 System Model

The reference model we use throughout this paper, represents a typical deployment of a three-tier web application, with a firewall to divide front-end components from the back-end components. The front-end network hosts the web server (WS), while the back-end network hosts an application server and a database server. Due to space considerations, we only describe the details of the WS, but the same modeling technique and considerations apply to the other components as well.

Each server is characterized by a collection of variables that compose the overall system state. Variables can either refer to server or operating system specific attributes, or to the services that are running on the server. In our case, the WS exposes two services: Apache `httpd` and the FTP server `vsftpd`. The WS is characterized by the following variables (we do not list the variables associated with `httpd`): `isWebServerOn`, `isWebServerUnderAttack` identify respectively whether the WS is on and under attack, `isFSCorrupted`, `CPULoad` represent whether any filesystem needs repair and the current CPU load, `isVsftpdStarted`, `isVsftpdVulnerable`, `isVsftpdNewVersionAvailable`, `isVsftpdUnderAttack` identify respectively whether the FTP server is started, vulnerable, up to date, and under attack. In the domain of automated intrusion response, with the objective of simplifying the formulation of the system model, it is common to consider the actions independently from the state where they are executed [9]. In this paper, we use a simplified version of the reward function that only depends on the executed actions, that is, for all states $s_0, s_1 \in S$ and actions $a \in A$ we have that $R(s_0, a, s_1) = \hat{R}(a)$ holds, and:

$$\hat{R}(a) = -w_T \frac{T(a)}{T_{max}} - w_C \frac{C(a)}{C_{max}} - w_{Conf} Conf(a) - w_I I(a) - w_A A(a)$$

where $w_T, w_C, w_{Conf}, w_I, w_A \in [0, 1]$ reflect the importance of, respectively, execution time $T(a)$, cost $C(a)$, confidentiality $Conf(a)$, integrity $I(a)$, availability $A(a)$, and optimization criteria for action a. We list in Table 1 the reward parameters of the actions related to the WS. Additional actions with similar rewards are however defined for the other components of the system. These parameters are used for the experiments described in Section 3.

Actions operate on the state variables and allow the system to transition among states. In order to implement the MDP's transition function P, every action includes a post-condition in the form of a probability distribution over the next possible states. For instance, the post-condition of both the actions `patchVsftpd` and `updateVsftpd` is:

Action a	$T(a)$	$C(a)$	$Conf(a)$	$I(a)$	$A(a)$
ScaleUP WS	10	10	0	0	0
ScaleDown WS	10	-10	0	0	0
Startup WS	60	0	0	0	0
Shutdown WS	60	0	0	0	1
Cut Cord WS	1	0	0	0	1
`fsck` WS	1800	0	0	0	0
Start `httpd`	5	0	0	0	0
Stop `httpd`	5	0	0	0	1
Update `httpd`	1800	0	0	0	0
Patch `httpd`	10	1	0	0	1
Start `vsftpd`	5	0	0	0	0
Stop `vsftpd`	5	0	0	0	1
Update `vsftpd`	1800	0	0	0	0
Patch `vsftpd`	10	1	0	0	1

TABLE 1: Web Server Actions

$$\begin{cases} Prob = 0.9 \rightarrow & \texttt{isVsftpdNewVersionAvailable=false,} \\ & \texttt{isVsftpdVulnerable=false} \\ Prob = 0.1 \rightarrow & \emptyset \end{cases}$$

In other words, both of them have the same effect on the system, but with different rewards.

Furthermore, every action is also characterized by a pre-condition, which identifies the states in which it can be executed. For instance, the pre-condition for `startHttpd` is: `isWebServerOn` $\wedge \neg$`isHttpdStarted`.

The subset T of the secure states is: $T = \{s \in S | secure(s)\}$, where $secure(s)$ is defined as: `isWebServerOn` $\wedge$ $\neg$`isWebServerUnderAttack` $\wedge$ $\neg$`isFSCorrupted` $\wedge$ $CPULoad < 70$ $\wedge$ `isVsftpdStarted` $\wedge$ $\neg$`isVsftpdVulnerable` $\wedge$ $\neg$`isVsftpdUnderAttack`.

3 Experimental Results

The objective of this work is to show that reinforcement learning techniques, such as Q-Learning, can be used to dynamically adapt the defense policies to the non-stationary behavior of the target system. To this end, we designed three experimental scenarios, where a given system changes its behavior due to either (i) the availability of an additional action; (ii) the change in the reward parameters of the existing actions; and (iii) the removal of a previously available action. In all the experiments, an agent starts learning the system behavior without any prior knowledge. After 200, 000 learning episodes, an environment change occurs. A learning episode is the sequence of actions that take the system from the current state into a state belonging to the secure region.

The metric that we use to evaluate the effectiveness of the proposed approach is the cumulative reward achieved by the learning agent, that is, the sum of the rewards that it obtains by moving from a given initial state to a final state. We show how the cumulative reward varies according to the number of learning episodes and the learning coefficient. We compare the obtained reward to the average cumulative reward μ that would have been achieved with complete knowledge of the system. For the latter, we also compute the standard deviation σ, and we plot the average optimal cumulative reward range as $[\mu-\sigma, \mu+\sigma]$. The agent uses an $\epsilon - greedy$ policy, with $\epsilon = 0.1$, meaning that 90% of times the agent will exploit the action with the maximum q-value, while 10% of times a sub-optimal action will be chosen.

We used the same initial state for all the experiments; due to space limitations, we only report the attributes related to the WS and to the `vsftpd` service: `[isWebServerOn=true; isWebServerUnderAttack=true; isFSCorrupted=false; CPULoad=400; Instances=1; isVsftpdStarted=true; isVsftpdVulnerable=true; isVsftpdNewVersionAvailable=true; isVsftpdUnderAttack=false]`. In other words, the system is currently being attacked by a Denial of Service (`CPULoad=400` means that the current workload could be sustained with 4 servers with CPU load at 100% each, while in the current state there is only 1 active server instance). Although `vsftpd` is not under attack, a vulnerability has been found (`isVsftpdVulnerable=true`) and a new software version is already available (`isVsftpdNewVersionAvailable=true`).

The weights for $\hat{R}$ have been set as follows: $w_T = 0.33, w_C = 0.33, w_{Conf} = 0, w_I = 0, w_A = 0.34$.

The simulator has been realized using the Java BURLAP library [1], which implements many well-known algorithms for planning and learning with MDPs. We executed the experiments on a single node of the Shadow supercomputer at Mississippi State University, equipped with 20 2.8Ghz cores and 512GB of RAM.

3.1 Availability of an Additional Action

This set of experiments shows that the proposed technique is able to quickly incorporate new actions in the defense policies, if they exhibit a better reward. We started this experiment by letting an agent learn the behavior of a system that did not have the `patchVsftpd` action. As a consequence, only `updateVsftpd` was present to fix the vulnerability. After $200,000$ learning episodes, we introduced the `patchVsftpd` action, and the agent was able to quickly take advantage of the new action in the average in the next 10 training steps from its introduction, reflecting the $\epsilon = 0.1$ parameter of the $\epsilon - greedy$ policy.

Figure 2 shows the cumulative reward of the learning agent according to the number of played episodes. As expected, the agent with the lowest learning coefficient is also the slowest to approximate the optimal q-value function, as it converges to Q^* after nearly $200,000$ episodes. With $\alpha \geq 0.5$, instead, Q^* is approximated in only about $60,000$ learning episodes. It is worth noting that the reward gap between the learning agents and the planning with complete knowledge is proportional to ϵ. Thus, it can be tuned to trade convergence speed with accuracy and vice versa.

3.2 Change of the Reward Parameters

This set of experiments shows that the proposed technique is able to quickly react to changes in the reward parameters of the actions. We started this experiment by letting an agent learn the behavior of the system described in Section 2.1

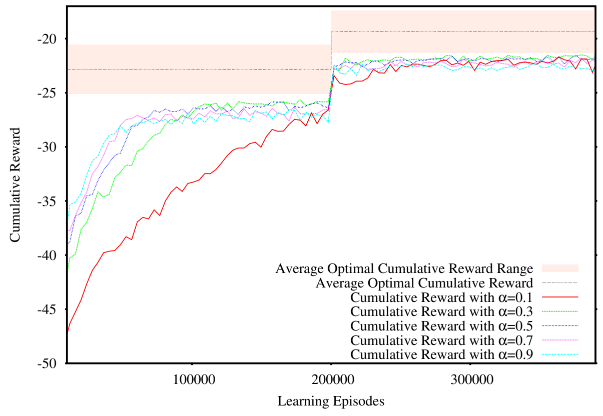

Fig. 2: Availability of an Additional Action

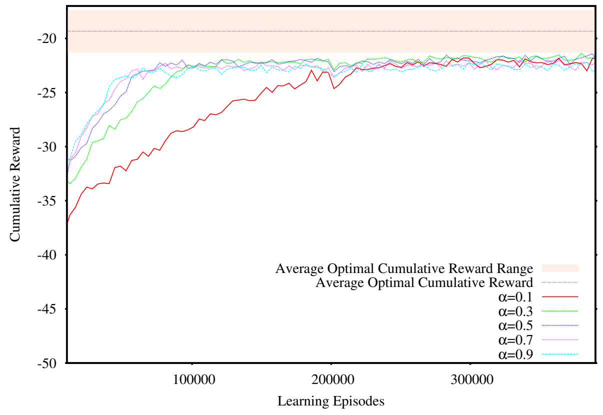

Fig. 3: Change of the Reward Parameters

and, after $200,000$ learning episodes, we swapped the reward parameters of `patchVsftpd` and `updateVsftpd`. In the first part of the experiment, the former was always chosen because it offered a better reward, but in the second part, its reward was sub-optimal. It is possible to see, indeed, that the cumulative reward in Figure 3 has a negative peak at $200,000$ learning episodes, which is due to the abrupt, and unexpected, change of behavior of the system. However, the agent was able to quickly adapt to the new environment, as all the curves with $\alpha \geq 0.3$ converged again in approximately 100 learning episodes. Of course, the higher the learning coefficient, the faster is the adaptation to the new environment.

3.3 Removal of an Action

In this set of experiments we show how the agent reacts to the removal of a defense action. We always start by training the agent with the system described in Section 2.1 and after $200,000$ learning episodes, we removed the `patchVsftpd` action, which was preferred over `updateVsftpd` for its better reward. Figure 4 shows the cumulative reward obtained by the agent, which has a negative peak at $200,000$ learning episodes due to the abrupt removal of the action. The time needed to find a new good approximation of Q^*, in terms of learning episodes, is in this case higher than that of the other experiments. This is due to the fact that Q-Learning updates

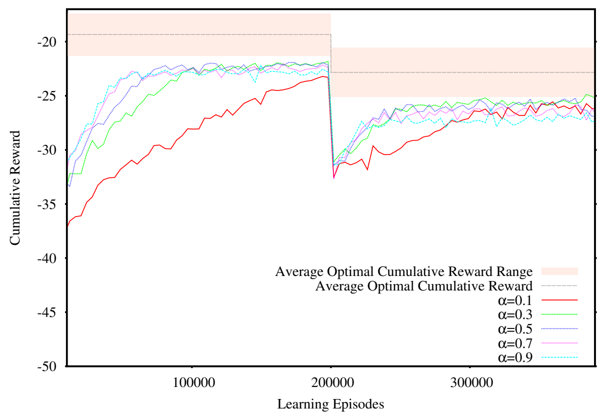

Fig. 4: Removal of an Action

the q-value for a given state-action pair $\langle s, a \rangle$ only upon the selection of action a in state s. Since `updateVsftp` has a low reward, it also has a low probability of being chosen.

4 Related Works

Researchers have used reinforcement learning to address system security in various ways [13] [5]. Dejmal et al., applied reinforcement learning to denial of service attacks in a Peer-to-Peer network [3]. Reinforcement learning has also been used for security risk assessment to dynamically assess risks and select defense mechanisms [6]. While these approaches show that reinforcement learning techniques are appropriate when addressing security concerns, we have not found any prior work that addresses the dynamic nature of the computing environment and the time required to reach a near optimal solution using a model-free approach to intrusion response. In our previous work, we used the Markov decision process (MDP) framework to model a system controlled by an intrusion response system [9]. In this paper, we present a novel, model-free approach to autonomic security management that applies a hybrid technique using simulation to converge on a near optimal configuration thus reducing the time to independent system execution. Our results show that we can capture the dynamics of the system and respond to actions by dynamically changing the system configuration.

5 Conclusions and Future Works

Automating the defense of computer systems is critical to reduce the chances of successful malicious attacks. Most of the research in this area over the past decade has been done on intrusion detection, whilst research on intrusion response is still at its early stages. The works proposed so far try to model either the behavior of the attacker, or the behavior of the defender. In some cases, a game theoretical formulation is provided to combine the behaviors of the attacker and the defender. However, we observe that computer systems may exhibit non stationary behavior due to configuration changes, availability of new applications, and change in user behavior. It is important therefore not to base the entire defense life-cycle only on a static model of the system, because it might not be representative of its actual behavior.

We proposed and evaluated an automatic intrusion response based on Q-Learning, which is able to automatically adapt to changes in the environment. We showed that a learning agent can quickly react to the addition of a new defense action and to the change of the reward parameters of existing actions. The removal of actions requires instead more time to adapt to, due to how Q-Learning is designed. As a consequence, we can say that although it is certainly infeasible to train an agent with zero knowledge on a real system, due to the large number of needed learning episodes, it is certainly feasible to train it with a model-based simulation of the target system. Such an agent can then be detached from the simulator and attached to the real system where it can exploit its ability to adapt to environmental changes and to learn the differences between the system model and the real system.

In the future, we will compare different learning techniques, among which, Q-Learning, SARSA and ANN. Furthermore, we will extend scope to include attacker behavior.

References

[1] Brown-umbc reinforcement learning and planning (burlap). http://burlap.cs.brown.edu/.

[2] R. Bellman. Dynamic programming. princeton, nj: Princeton universitypress. *BellmanDynamic Programming1957*, 1957.

[3] S. Dejmal, A. Fern, and T. P. Nguyen. Reinforcement learning for vulnerability assessment in peer-to-peer networks. In *AAAI*, pages 1655–1662, 2008.

[4] M. J. Druzdzel and R. R. Flynn. Encyclopedia of library and information science, chapter decision support systems, 2003.

[5] B. A. Fessi, S. Benabdallah, N. Boudriga, and M. Hamdi. A multi-attribute decision model for intrusion response system. *Information Sciences*, 270:237–254, 2014.

[6] J. C. Georgia. Next generation intrusion detection: Autonomous reinforcement learning of network attacks. In *In Proceedings of the 23rd National Information Systems Secuity Conference*, pages 1–12, 2000.

[7] J. S. Hunter. The exponentially weighted moving average. *Journal of quality technology*, 18(4):203–210, 1986.

[8] S. Iannucci and S. Abdelwahed. A probabilistic approach to autonomic security management. In *Proceedings of the 13th IEEE International Conference on Autonomic Computing (ICAC)*, 2016.

[9] S. Iannucci and S. Abdelwahed. Model-based response planning strategies for autonomic intrusion protection. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 13(1):4, 2018.

[10] S. Iannucci, Q. Chen, and S. Abdelwahed. High-performance intrusion response planning on many-core architectures. In *Workshop on Network Security Analytics and Automation (NSAA)*, 2016.

[11] M. Kearns, Y. Mansour, and A. Y. Ng. A sparse sampling algorithm for near-optimal planning in large markov decision processes. *Machine Learning*, 49(2-3):193–208, 2002.

[12] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[13] E. Miehling, M. Rasouli, and D. Teneketzis. Optimal defense policies for partially observable spreading processes on bayesian attack graphs. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, pages 67–76. ACM, 2015.

[14] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[15] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 2018.

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

[17] E. Yuan, N. Esfahani, and S. Malek. A systematic survey of self-protecting software systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 8(4):17, 2014.

[18] S. A. Zonouz, H. Khurana, W. H. Sanders, and T. M. Yardley. Rre: A game-theoretic intrusion response and recovery engine. *Parallel and Distributed Systems, IEEE Transactions on*, 25(2):395–406, 2014.