Solving Cyber Alert Allocation Markov Games with Deep Reinforcement Learning

Noah Dunstatter(✉), Alireza Tahsini, Mina Guirguis, and Jelena Tešić

Department of Computer Science, Texas State University,
601 University Dr, San Marcos, TX 78666, USA
{nfd8,tahsini,msg,jtesic}@txstate.edu

Abstract. Companies and organizations typically employ different forms of intrusion detection (and prevention) systems on their computer and network resources (e.g., servers, routers) that monitor and flag suspicious and/or abnormal activities. When a possible malicious activity is detected, one or more cyber-alerts are generated with varying levels of significance (e.g., high, medium, or low). Some subset of these alerts may then be assigned to cyber-security analysts on staff for further investigation. Due to the wide range of potential attacks and the high degrees of attack sophistication, identifying what constitutes a *true* attack is a challenging problem. In this paper, we present a framework that allows us to derive game-theoretic strategies for assigning security alerts to security analysts. Our approach considers a series of sub-games between the attacker and defender with a state maintained between sub-games. Due to the large sizes of the action and state spaces, we present a technique that uses deep neural networks in conjunction with Q-learning to derive near-optimal Nash strategies for both attacker and defender. We assess the effectiveness of these policies by comparing them to optimal policies obtained from brute force value iteration methods, as well as other sensible heuristics (e.g., random and myopic). Our results show that we consistently obtain policies whose utility is comparable to that of the optimal solution, while drastically reducing the run times needed to achieve such policies.

Keywords: Game theory · Markov game · Network security · Machine learning · Deep reinforcement learning

1 Introduction

Motivation and Scope: The rise of Advanced Persistent Threats (APT) against private and public organizations has put a significant strain on the resources held by these organization and specially on their cyber-defense personnel (i.e., security analysts) that need to investigate security issues that arise. When an attack is launched against an organization, network and system

© Springer Nature Switzerland AG 2019
T. Alpcan et al. (Eds.): GameSec 2019, LNCS 11836, pp. 164–183, 2019.
https://doi.org/10.1007/978-3-030-32430-8_11

resources (e.g., Intrusion Detection Systems, Anti-malware tools, etc.) typically generate cyber-alerts with varying levels of severity. These alerts need to be investigated by analysts to thwart ongoing attacks. Legitimate network activity, however, can also trigger the generation of some of these alerts, making it extremely challenging for analysts to discern a true attack from legitimate activity. A worst-case scenario happens when precious time is wasted investigating false alerts while true alerts from ongoing attacks are ignored. This in fact occurred during the Target attack wherein malware alerts were repeatedly generated but not addressed by analysts [21].

To highlight the severity of this issue within the network security domain, in [16] it was reported that an average of 17,000 alerts are generated by intrusion detection software every week at the surveyed organizations. Of these alerts, about 19% (3,218) are estimated to be legitimate, with only 4% (705) eventually being investigated by security analysts. This relatively low proportion of assignment makes the defender's allocation strategy (i.e., assignment of alerts to analysts) much more critical when it comes to minimizing risk. This high volume of alerts coupled with a typically small numbers of security analysts motivated us to investigate what a game-theoretic policy looks like in such a domain.

The use of game-theory to study allocation of cyber alerts to analysts has been investigated in previous works [4,18,19]. The authors in [18,19] introduce a Stackelberg game-theoretic model to determine the best allocation of incoming cyber alerts to analysts. The model assumes a one shot-game in which both the alert resolution time and the arriving alert distribution are deterministically known. In [4] the authors develop a game-theoretic model that considers a series of games between the attacker and the defender with a state maintained between such sub-games that captures the availability of analysts as well as an attack budget metric. Using dynamic programming coupled with Q-maximin value iteration based algorithms they were able to obtain optimal policies for both players. However, as the state and action spaces of the games become larger, it becomes computationally prohibitive to use such methods to obtain optimal policies. Instead it is common to adopt methods capable of approximations of the optimal policy. Methods such as deep reinforcement learning have show great promise in the single agent domain wherein networks were shown to learn empirically successful policies from the raw frames of various video games [10,15].

As opposed to previous works, in this paper we adopt a deep reinforcement learning approach that we modify to handle the game-theoretic nature of uncooperative extensive form 2-player games.

Contributions: In this paper, we make the following contributions:

1. Develop a Deep Nash Q-Network framework that captures the game-theoretic behaviors of the attacker and defender through replacing the traditional greedy max operator with minimax one.
2. Tame the curse of dimensionality caused by the prohibitively large state and action spaces by (1) relying on deep reinforcement learning to approximate the quality of state-actions pairs, (2) performing a loss-less compression of

player action spaces without sacrificing the representation of states and/or actions, and (3) use iterative fictitious play to approximate the solutions to sub-games.

3. Assess the performance of our proposed approximate game-theoretic solution method and its derived policies against the known optimal value-iteration based solution (where the state space allows) as well as other heuristics that are typically employed.

Paper Organization: In Sect. 2 we put this work in context with related work. In Sect. 3 we present our stochastic game formulation for the cyber-alert assignment problem and in Sect. 4 we present our solution framework focusing on the methods developed to tackle the large state/action space and to solve the sub-games efficiently. In Sect. 5 we present our performance evaluation and conclude the paper in Sect. 6 with a summary.

2 Related Work

Improving the scheduling and efficiency of cyber-security analysts is a highly studied area of research [1,5,6,26]. The authors in [1] model the problem as a two-stage stochastic shift scheduling problem in which the first stage allocates cyber-security analysts and in the second stage additional analysts are allocated. The problem is discretized and solved using a column generation based heuristic. The authors in [5] study optimal alert allocation strategies with a static workforce size and a fixed alert generation mechanism. In [6] the authors develop a reinforcement learning-based dynamic programming model to schedule cyber-security analyst shifts with the model based on a Markov Decision Process framework with stochastic load demands. In [26], the author describes different strategies for managing security incidents in a cyber-security operation center. The authors in [20] propose a queuing model to determine the readiness of a Cyber-Security Operations Center (CSOC). This paper departs from this previous research by explicitly considering the presence of a strategic and well informed adversary within a temporally stateful environment.

The use of game theory has been instrumental in advancing the state-of-the-art in security games and their wide range of applications [3,7,18,19,22,25]. As mentioned in Sect. 1, the work in [4,18,19] adopted a Stackelberg game-theoretic approach in allocating alerts to analysts. Unlike the one-shot game model in [18,19], we consider a series of games with stochastic alert arrival process. Unlike the work in [4] which uses dynamic programing and value iteration, we adopt deep reinforcement learning techniques that enables us to tackle problems with large state and action spaces.

In contrast to the previously mentioned Stackelberg games, the authors in [9,12,13] explore solution methods to stateful Markov games. Convergence properties and Q-minimax value iteration are studied in [12,13] providing encouraging guarantees for optimality and convergence of value functions. This process is then approximated by the authors in [9], wherein they use a least squares policy iteration approach to train a linear function approximator to predict Q-values.

The use of stateful Markov game models to study security games with real-world applications are limited. The authors in [14] used dynamic programming and value iteration to investigate attacks on power grids. Using a similar method, the authors in [4] investigated the use of full state space value iteration in the alert assignment domain. The authors in [24] used Markov games to model the level of worst-case threat faced by an institution given parameters surrounding its network infrastructure. Despite guarantees of optimality and convergence, the use of full state space value iteration in many previous works' solution methods scale poorly to large real-world sized models – prompting the need for approximate solution methods in the domain.

3 Cyber-Alert Assignment Markov Game

We consider a two-player zero-sum Markov game in which the Defender ($\mathcal{D}$) and the Attacker ($\mathcal{A}$) play a series of sub-games over an infinite time horizon [4]. At each time step t, a new batch of alerts $\omega \in \Omega$ arrives in which $\mathcal{A}$ chooses some alert level(s) to attack in and $\mathcal{D}$ attempts to detect and thwart the incoming attack(s) by assigning available analysts to the incoming alerts. We let $s \in \mathcal{S}$ denote the current state of the player resources (e.g., availability of analysts as well as the budget available to the attacker). We also let $\mathcal{D}_a$ and $\mathcal{A}_a$ denote the set of actions available to $\mathcal{D}$ and $\mathcal{A}$, respectively. We define a transition function $T : \mathcal{S} \times \mathcal{D}_a \times \mathcal{A}_a \to \Pi(\mathcal{S})$ which maps each state and player action pair to a probability distribution over possible next states. We let $T(s, a, d, s')$ denote the probability that, after taking actions $a \in \mathcal{A}_a$ and $d \in \mathcal{D}_a$ in state s, the system will make a transition into s'. In general, the system can be described as follows:

- **Alerts:** Our transition function T is manifested in the uncertainty of which i.i.d. batch of alerts will arrive in each state. At every time step t, some batch of alerts $\omega \in \Omega$ arrives according to a pre-defined probability distribution $\Pi(\Omega)$, where $\Omega = \{\omega_1, \omega_2, \ldots, \omega_{|\Omega|}\}$. Each alert $\sigma \in \omega$ belongs to one of three categories: High (h), Medium (m), or Low (l). Resolving an alert requires a certain number of time steps based on its category. The set holding each category's work-time (i.e., the number of time steps needed by an analyst to investigate and resolve an alert) is defined as $\mathcal{W} = \{w_h,\ w_m,\ w_l\}$ where $w_h > w_m > w_l$ and $w_h, w_m, w_l \in \mathbb{N}$. A similar reward structure $\mathcal{U} = \{u_h,\ u_m,\ u_l\}$, where $u_h > u_m > u_l$ and $u_h, u_m, u_l \in \mathbb{N}$, is defined for each category as well. If the alert σ^h is legitimate and is assigned to an analyst, $\mathcal{D}$ will receive a positive utility u_h. Whereas not assigning the legitimate alert results in a negative utility $-u_h$ for $\mathcal{D}$. If an alert is illegitimate (i.e., a false positive) then it awards no utility to either the attacker or defender, regardless of whether or not it is assigned. Since our model is zero-sum, the corresponding utilities for $\mathcal{A}$ are simply the additive inverse of those awarded to $\mathcal{D}$. For example, a possible arrival set may be as follows: $\Omega = \{\omega_1,\ \omega_2\}$ where $\Pr(\omega_1) = 0.4$ with $\omega_1 = \{\sigma_1^h,\ \sigma_2^m,\ \sigma_3^m\}$, and $\Pr(\omega_2) = 0.6$ with $\omega_2 = \{\sigma_1^m,\ \sigma_2^l\}$. At the beginning of a particular sub-game, both players are aware

of the exact alert batch that has arrived (with future alert arrivals remaining probabilistic, thereby impacting the current resource allocations made by the players). It is important to note that not every alert may be assigned to an analyst, and not every alert may represent a legitimate attack (i.e., alerts can be false positives).

- **The Defender:** $\mathcal{D}$ has n homogeneous cyber-security analysts available to handle incoming alerts. We define R_s as a vector of length n that describes the load of each analyst in state s. For example, $R_s = [0\ 2\ 1]$ means that $\mathcal{D}$ has 3 analysts on their team in which analyst 1 is available for assignment (has load 0), analyst 2 will be available after two time steps, and analyst 3 will be available after one time step. We also define the function $\mathcal{F}(R_s)$ as the number of analysts currently available for allocation in state s. In every time step t, $\mathcal{D}$ receives a batch of alerts ω and determines their allocation strategy based upon the current availability of analysts and the varying severity and volume of alerts in ω. Once the set of possible alert batches Ω and each alert category's respective work-times $\mathcal{W}$ are known, we can construct the set of all possible analyst states. In general, $|R|$ is bounded above by the following:

$$|R| \leq (w_h + 1)^n \tag{1}$$

- **The Attacker**: $\mathcal{A}$ has an attack budget $B \in \mathbb{N}$ and decides when to attack and in what category. We assume $\mathcal{A}$ knows the alert level that would be generated due to their attack. The set $\mathcal{C} = \{c_h,\ c_m,\ c_l\}$ defines the respective cost to the attacker given the alert level their attack generates, where $c_h > c_m > c_l$ and $c_h, c_m, c_l \in \mathbb{N}$. $\mathcal{A}$ can attack with as many alerts as they wish as long as the sum of their costs is affordable given the current budget. The attacker's budget enables us to model the amount of risk they are willing to undertake. Attacking more frequently with attacks that generate high level alerts would likely expose $\mathcal{A}$. To capture this behavior, if $\mathcal{A}$ chooses to abstain from attacking in a state s with budget B_s they will be credited with 1 unit of budget in the subsequent state. However, their budget is capped to some value B_{max} representing the maximum amount of risk they are willing to undertake in any one state.
- **State Representation:** At the beginning of a time step, we assume that the system state is known to both players. A state is thus defined as follows:

$$s = [R_s | B_s] \tag{2}$$

Once we know the current state s and current alert arrival batch ω, we can formulate the action spaces of both players. The size of the defender's action space $\mathcal{D}_a$ grows combinatorially and is defined in Eq. 3. The size of the attacker's action space $\mathcal{A}_a$ is defined in Eq. 4, where the indicator function $\mathbb{1}_{\{\cdot\}}$ enumerates all the ways the attacker could attack using the alerts in ω (while also allowing them to abstain from attacking altogether).

$$|\mathcal{D}_a(s, \omega)| = \sum_{i=0}^{\mathcal{F}(R_s)} \binom{|\omega|}{i} \tag{3}$$

$$|\mathcal{A}_a(s, \omega)| = \sum_{i=1}^{2^{|\omega|}} \mathbb{1}_{\{B_s \geq Bin(i-1) \cdot \mathcal{C}_\omega\}} \tag{4}$$

$Bin(i)$ is a function that maps the integer i to its binary representation and $\mathcal{C}_\omega$ is the cost vector for the alerts in ω according to $\mathcal{C}$. For example, an alert arrival $\omega = \{\sigma_1^m,\ \sigma_2^l,\ \sigma_3^l\}$ yields a cost vector $\mathcal{C}_\omega = [3\ 1\ 1]$. Thus, given an attacker budget $B_s = 2$ for this state and arrival $\mathcal{A}_a = \{[0\ 0\ 0],\ [0\ 1\ 0],\ [0\ 0\ 1],\ [0\ 1\ 1]\}$.

Using the players' action spaces we can now formulate a zero-sum game represented by a payoff matrix $\mathcal{R}_s$ of instantaneous rewards where $\mathcal{R}_s(a, d)$ represents the reward received when players commit to actions a and d. Each state-arrival pair in our environment represents a potential sub-game where the defender attempts to detect incoming attacks through some of assignment of alerts to analysts and the attacker attempts to evade detection when launching their attacks. Since the game is zero-sum, for the remainder of the paper we will not distinguish between defender and attacker rewards and will only discuss rewards from the defender's perspective (i.e., the defender seeks to maximize rewards and the attacker seeks to minimize rewards).

Players follow policies $\pi_{\mathcal{D}}$ and $\pi_{\mathcal{A}}$ (e.g., $\pi_{\mathcal{D}}(s, d)$ is the probability $\mathcal{D}$ takes action $d \in \mathcal{D}_a(s, \omega)$). These policies are obtained by solving for the Nash equilibrium of the payoff matrix $\mathcal{R}$. While many algorithms exist to solve for such Nash equilibrium (e.g., linear programming) we chose to use an iterative fictitious play algorithm as it was much faster than the other methods we explored and still yielded accurate results. Once derived, both players will sample from their Nash equilibrium mixed strategies and commit to their respective actions. They are then awarded their respective utility and the state evolves from s to s' according to the transition function T.

4 Methods

In this section we will first describe our method for compressing the combinatorial action spaces of our players, followed by a description of the iterative fictitious play algorithm used to quickly solve sub-games. Lastly we will define the DNQN methodology used to obtain approximate Nash policies in large Markov games.

4.1 Action Space Compression

We can represent our agents' actions as a binary vector where a 1 represents that a given alert is either assigned to an analyst for the defender, or attacked in by the attacker. For example, if we are in a state $s = \big[R = [0, 0, 0] \mid B = 10\big]$ and a batch of alerts $\omega = \{\sigma_1^h, \sigma_2^h, \sigma_3^m\}$ arrives, where $c_h = 5$ and $c_m = 3$, our defender and attacker action spaces would be formulated as follows:

$$\mathcal{D}_a(s, \omega) = \{[0, 0, 0],\ [0, 0, 1],\ [0, 1, 0],\ [0, 1, 1],\ [1, 0, 0],\ [1, 0, 1],\ [1, 1, 0],\ [1, 1, 1]\}$$
$$\mathcal{A}_a(s, \omega) = \{[0, 0, 0],\ [0, 0, 1],\ [0, 1, 0],\ [0, 1, 1],\ [1, 0, 0],\ [1, 0, 1]\}$$

Where $d_6 = [1, 0, 1]$ means that the defender assigns alerts σ_1^h and σ_3^m and ignores alert σ_2^h. Similarly $a_4 = [0, 1, 1]$ means that the attacker attacks in alerts σ_2^h and σ_3^m and ignores alert σ_1^h.

However, consider the fact that alerts of the same severity level are homogeneous (i.e., their utility, work-time, and cost are equal). This means that our agents need not worry about which specific alerts are being assigned/attacked, only the number of alerts from each severity level being assigned/attacked. Thus we can represent actions as a 3-tuple $\langle h, m, l\rangle$ representing how many alerts from each severity level are being assigned/attacked:

$$\hat{\mathcal{D}}_a(s,\omega) = \{\langle 0,0,0\rangle,\ \langle 0,1,0\rangle,\ \langle 1,0,0\rangle,\ \langle 1,1,0\rangle,\ \langle 2,0,0\rangle,\ \langle 2,1,0\rangle\}$$
$$\hat{\mathcal{A}}_a(s,\omega) = \{\langle 0,0,0\rangle,\ \langle 0,1,0\rangle,\ \langle 1,0,0\rangle,\ \langle 1,1,0\rangle\}$$

where $d_3 = \langle 1, 1, 0\rangle$ means that the defender assigns one of the two high alerts and the one medium alert. Similarly $a_3 = \langle 1, 0, 0\rangle$ means that the attacker attacks in one of the high alerts and ignores the other high and medium alert.

To illustrate just how advantageous this compression is, consider an arrival $\omega_s = \{\sigma_1^h, \sigma_2^h, \sigma_3^m, \sigma_4^m, \sigma_5^l, \sigma_6^l\}$ in a state where the number of available analysts $\mathcal{F}(R_s) = 5$. Under the combinatorial action space $|\mathcal{D}_a(s,\omega)| = 63$ whereas the compressed action space $|\hat{\mathcal{D}}_a(s,\omega)| = 26$ (a 58% reduction). The compression is even more substantial when arrival batches posses more redundancy. For instance, if $\omega_s = \{\sigma_1^h, \sigma_2^h, \sigma_3^h, \sigma_4^h, \sigma_5^h, \sigma_6^h\}$ then $|\hat{\mathcal{D}}_a(s,\omega)| = 6$ while our combinatorial action space remains unchanged at $|\mathcal{D}_a(s,\omega)| = 63$ (while this small example is given for illustration purposes, in our larger models the compression can routinely get as high as 96%).

Under the compressed action space formulation joint action rewards are calculated in expectation since we no longer specify exactly which alerts are being assigned/attacked. The simplified equation for deriving the compressed payoff matrix rewards $\hat{\mathcal{R}}(\cdot)$ is presented below:

$$z_{\max} = \min(a_k, \sigma_k)$$

$$z_{\min} = \begin{cases} 0, & \text{if } d_k - (\sigma_k - a_k) < 0; \\ d_k - (\sigma_k - a_k), & \text{otherwise.} \end{cases}$$

$$\hat{\mathcal{R}}(a,d) = \sum_{k\in\{h,m,l\}} \sum_{z=z_{\min}}^{z_{\max}} \frac{\binom{a_k}{z}\binom{\sigma_k - a_k}{d_k - z}}{\binom{\sigma_k}{d_k}} u_k(2z - a_k)$$

where σ_k is the number of alerts in severity level k, d_k and a_k are the number of alerts assigned/attacked from severity level k, and z represents the number of alerts caught by the defender in the current severity level.

Most advantageous of all is that this compression is completely loss-less with respect to finding the value of a Nash equilibrium in our sub-games (i.e., whether using $\mathcal{R}$ or $\hat{\mathcal{R}}$ our linear program will find the same expected value of the game).

The mixed-strategies will necessarily be different (since the action spaces are different), however the mixed strategies derived from $\hat{\mathcal{R}}$ will be more meaningful to our neural network as they contain much less redundancy.

4.2 Fictitious Play

The biggest bottleneck for both of the algorithms developed in this work is the game solving mechanism. Solving the game via a linear programming approach, while accurate, incurs a large cost in terms of run-time. In both of our solutions we are solving millions of games – often containing hundreds of potential action pairs. Thus while solving one of these games is more or less instantaneous, solving our Markov game can take many hours.

This motivated us to explore potential alternatives to using linear programming to solve our games. Ultimately we settled on the use of fictitious play, an iterative algorithm first introduced by Brown in 1951 [2]. In fictitious play each player tracks the empirical frequency of actions chosen by their opponent and best responds to this strategy. The other player, having also tracked the empirical frequency of their opponent, also plays their best response. By iterating this process many times we are able to derive close approximations of both the game's value and the Nash equilibrium policies of the agents. A proof of convergence for this iterative approach in zero-sum games is given in [17].

Algorithm 1. Iterative Fictitious Play

$\mathcal{R}$ is an $m \times n$ matrix of rewards
$rowReward$ and $rowCnt$ are m-length arrays of zeros
$colReward$ and $colCnt$ are n-length arrays of zeros
Initialize $bestResponse$ to any random row action
for $iterations$ **do**
  $colReward = colReward + \mathcal{R}[bestResponse, \cdot\,]$
  $bestResponse = \text{argmin}(colReward)$
  $colCnt = colCnt + 1$
  $rowReward = rowReward + \mathcal{R}[\,\cdot\,, bestResponse]$
  $bestResponse = \text{argmax}(rowReward)$
  $rowCnt = rowCnt + 1$
end for
$gameValue = \Big(\big(\max(rowReward) + \min(colReward)\big) \,/\, 2\Big) \,/\, iterations$
$rowMixedStrat = rowCnt \,/\, iterations$
$colMixedStrat = colCnt \,/\, iterations$=0

The algorithm we use for our fictitious play was first introduced by Williams in [23] and is described in Algorithm 1. (please note that element-wise vector operations are implied here, with scalar values being broadcast to all elements of the vector in question).

4.3 Deep Nash Q-Network

Motivated by the success in [15] we wanted to explore the possibility of applying Deep Q-Networks in the Markov game domain. After all, even complex systems like those of Atari games can be formulated as MDP's. Furthermore, the transformation from the single agent MDP to the multi-agent Markov game is straightforward enough that we hoped the approximation power achieved in [15] would carry over to the Markov game domain. Especially if we were able to train this network in a manner similar to the provably optimal value iteration approach presented in [4].

This move from single agent to multi-agent would naturally necessitate some changes to the Q-learning and network loss equations in [15]. Namely, while both algorithms employ a Q-function to estimate future rewards attainable from an action pair, their Q-target equation need only apply a greedy max over the next action for a single player, whereas our game theoretic approach must choose actions for two players in a minimax fashion. Given the convergence proofs for value iteration in Markov games provided in [12] and the empirical success of its use in our domain in [4], we can be confident that the substitution of a game theoretic maximin in place of the greedy max operator will provide a stable and meaningful reward signal. The equations used to train the network are as follows:

$$y_i = \mathbb{E}_{s \sim \mathcal{E}}\left[\mathcal{R}_s(a, d) + \gamma \max_{d' \in \hat{\mathcal{D}}_a} \min_{a' \in \hat{\mathcal{A}}_a} Q(s', a', d'; \hat{\theta}) | s, a, d\right] \tag{5}$$

$$L_i(\theta_i) = \mathbb{E}_{s,a,d \sim \rho(\cdot)}\left[\left(y_i - Q(s, a, d; \theta_i)\right)^2\right] \tag{6}$$

Equation 5 is the target we are moving our approximator towards, with $\mathcal{E}$ being the environment we sample states from. Equation 6 is the loss function, with $\rho(s, a, d)$ representing the behavior distribution (in our case, the current maximin ϵ-greedy policies of our agents) that defines a probability distribution over states and actions. Differentiating Eq. 6 with respect to the weights gives us the following,

$$\begin{aligned} \nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,d \sim \rho(\cdot); s' \sim \mathcal{E}} \Big[& \nabla_{\theta_i} Q(s, a, d; \theta_i) \cdot \\ & \Big(\mathcal{R}_s(a, d) + \gamma \max_{d' \in \hat{\mathcal{D}}_a} \min_{a' \in \hat{\mathcal{A}}_a} Q(s', a', d'; \hat{\theta}) - Q(s, a, d; \theta_i)\Big)\Big] \end{aligned} \tag{7}$$

To avoid computing the full expectations of Eq. 7 we can use single samples of actions from the behavior distribution ρ and transitions from our environment $\mathcal{E}$ while using an optimization function (e.g., stochastic gradient descent) to minimize the loss.

The authors in [15] use two sets of networks while training – a target network $\hat{\theta}$ and a training network θ. Their main purpose in using a target network was to hold the data the network is approximating towards fixed, empirically alleviating convergence issues. While this is also true in our case, we extended the

Algorithm 2. Deep Nash Q-Network with Experience Replay

Initialize $i = 0$
Initialize learning network with random weights θ_i
Initialize target network weights $\hat{\theta} = \theta_i$
Initialize τ to desired update cycle
Initialize replay memory $\mathcal{Z}$ to capacity N
for episode $= 1, M$ **do**
  Initialize s_1 randomly
  for $t = 1, T$ **do**
    $\mathcal{Q}_s$ = GetQMatrix(s, θ_i)
    $\langle \pi_{\mathcal{A}},\ \pi_{\mathcal{D}} \rangle$ = GameSolver($\mathcal{Q}_s$)
    With probability ϵ select a random action a_t
      otherwise sample $a_t \sim \pi_{\mathcal{A}}$
    With probability ϵ select a random action d_t
      otherwise sample $d_t \sim \pi_{\mathcal{D}}$
    Execute actions $\langle a_t, d_t \rangle$ in $\mathcal{E}$ and observe reward r_t and next state s_{t+1}
    Store transition $(s_t, a_t, d_t, r_t, s_{t+1})$ in $\mathcal{Z}$
    Sample random mini-batch of transitions $(s_j, a_j, d_j, r_j, s_{j+1})$ from $\mathcal{Z}$
    $\mathcal{Q}_{s_{j+1}}$ = GetQMatrix(s_{j+1}, $\hat{\theta}$)
    $\langle v_{s_{j+1}} \rangle$ = GameSolver($\mathcal{Q}_{s_{j+1}}$)
    Set $y_j = r_j + \gamma v_{s_{j+1}}$
    Take gradient descent step on $\big(y_j - Q(s_j, a_j, d_j; \theta_i)\big)^2$ using Eq. 7
    Set $i = i + 1$
    if i mod $\tau = 0$, **then**
      $\hat{\theta} = \theta_i$
    end if
  end for
end for=0

use of a target network by delaying the amount of time between its successive updates (i.e., when $\hat{\theta}$ is set to the current θ) by τ learning steps. This update cycle $\tau \gg 1$ allows the network to come closer to understanding the future rewards available from a state before we change the target data. This results in a loose approximation of the value iteration approach in [4]. For example, when learning begins $\hat{\theta}$ initially makes prediction very close to zero. So before the first update takes place our target y_i is essentially a slightly noisy representation of immediate rewards. By the end of our first update cycle θ will predict these immediate rewards very accurately and once we update $\hat{\theta}$, our network will then begin bootstrapping from the learned immediate rewards to understand future rewards. Our state-action inputs to the neural network were as follows:

- For each analyst we calculate the current percentage of their remaining wait time resulting in n features.
- The percent of currently available budget for the attacker.
- Three features specifying the number of alerts from each severity level that had arrived (min-max normalized).

– Three features for the number of alerts assigned from each severity level (min-max normalized).
– Three features for the number of alerts attacked in each severity level (min-max normalized).

Most supervised machine learning algorithms rely on learning a fixed distribution given some set of samples. The difficulty in using such methods (e.g., stochastic gradient descent) in an RL context is that the samples obtained from an on-policy RL algorithm come from the estimator itself. Every time we update our policy we change the agent's behavior and thereby the distribution of rewards we are going to see. This kind of moving target can lead to convergence issues and is addressed in part via the use of experience replay [11]. Experience replay is a method where we store a 5-tuple of the agents' experience $e_t = \langle s_t, a_t, d_t, r_t, s_{t+1} \rangle$ at each time step in some large data set $\mathcal{Z} = e_1, ..., e_N$ from which we can sample from in a way analogous to traditional supervised learning. Readers are encouraged to review [15] for a detailed explanation of experience replay's many benefits when training RL agents in a large state space.

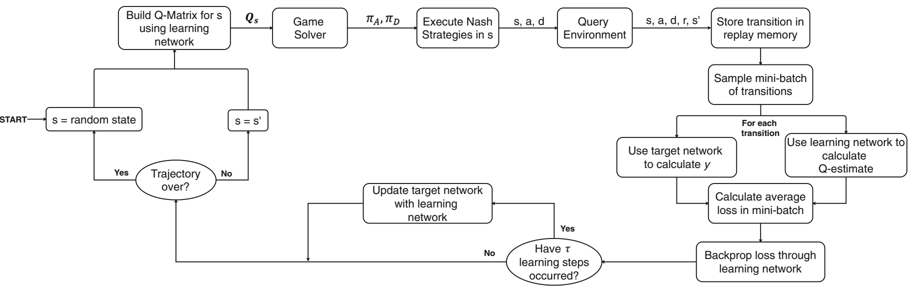

Fig. 1. Diagram of the DNQN learning process

We refer to this learning architecture (depicted in Fig. 1) as a Deep Nash Q-Network (DNQN). This network learns off-policy and is model-free. The pseudocode for training a DNQN is presented in Algorithm 2. The function GetQMatrix (s, θ) re-populates the payoff matrix $\mathcal{R}_s$ with Q-values predicted using features about state the state-action pairs in $\mathcal{R}_s$ and network weights θ. GameSolver ($\mathcal{Q}_s$) is a function that solves the matrix game $\mathcal{Q}_s$ and returns the Nash equilibrium value v_s of the game as well as the mixed strategies for both players that result in that value, $\pi_{\mathcal{A}}$ and $\pi_{\mathcal{D}}$.

Additionally, all experiments presented herein were performed using the Adam optimizer described in [8] in favor of the simple stochastic gradient descent approach. While both optimizers performed well in practice, Adam consistently lead to smoother learning curves and better performing policies.

5 Performance Evaluation

This section will present our experimental results. All results presented herein were obtained on a machine with two-dozen 2.2 GHz cores and 64 GB of RAM. Furthermore, the dynamic programming approach was run in a fully parallel manner while the DNQN approach was run serially.

5.1 Dynamic Programming Tractable Model

The first set of results we will discuss were obtained within a state space small enough to be solved in a reasonable amount of time (30 h) via the brute force value-iteration approach introduced in [4]. The parameters used when constructing the state space are presented in Table 1 and yield an environment with a total of 2,117,682 possible states our agents can inhabit.

Table 1. Parameters used when constructing the DP tractable model.

Parameter	Value
Number of experts n	5
Attack budget B	20
Utilities $\mathcal{U}$	$u_h = 100$, $u_m = 20$, $u_l = 5$
Attack cost $\mathcal{C}$	$c_h = 8$, $c_m = 4$, $c_l = 2$
Work times $\mathcal{W}$	$w_h = 6$, $w_m = 3$, $w_l = 1$
Alert batches in Ω where $\omega = \langle h, m, l \rangle$	$\omega_1 = \langle 0, 2, 2 \rangle$, $\omega_2 = \langle 1, 2, 2 \rangle$, $\omega_3 = \langle 0, 3, 3 \rangle$ $\omega_4 = \langle 1, 1, 4 \rangle$, $\omega_5 = \langle 2, 2, 3 \rangle$, $\omega_6 = \langle 3, 3, 3 \rangle$
Arrival prob. $\Pi(\Omega)$	$\omega_1 = 0.15$, $\omega_2 = 0.21$, $\omega_3 = 0.21$ $\omega_4 = 0.18$, $\omega_5 = 0.20$, $\omega_6 = 0.05$

This dynamic programming solution provides us with an example of what optimal behavior looks like in the Markov game domain, allowing us to understand how well our approximate DNQN approach compares in terms of both long-term cumulative utility and solution time.

When training our DNQN we used a fully connected $32 \times 64 \times 128 \times 128 \times 128$ architecture with ReLU activation functions and an update cycle $\tau = 1,000$. We experimented with various network sizes and update cycle lengths, finding the aforementioned values to be both the most stable and fruitful. While they did impact the overall accuracy of the model, our solution remained quite robust across all the network sizes we tried.

Figure 2 presents the mean loss of our network's predictions on each learning batch after every step taken during training. As discussed in Sect. 4.3, over the first update cycle (the first 1,000 iterations) our agent's are primarily learning immediate rewards and our network's loss drops quite rapidly. However, when the second update to the target network takes place our loss suddenly spikes by roughly 2,000%. This initial convergence and sudden spike demonstrate two things. First, our network can quickly learn the immediate rewards of a given action pair. Second, these immediate rewards are a poor reflection of long term

reward (Q-values). After the first update cycle our agents understand immediate reward very well meaning solving the Q-matrices from Algorithm 2 yield near-perfect Nash play (for each normal form game). However, once the normal form game becomes an extensive form game their strategies must re-adjust greatly as their greedy actions now bear heavy consequences.

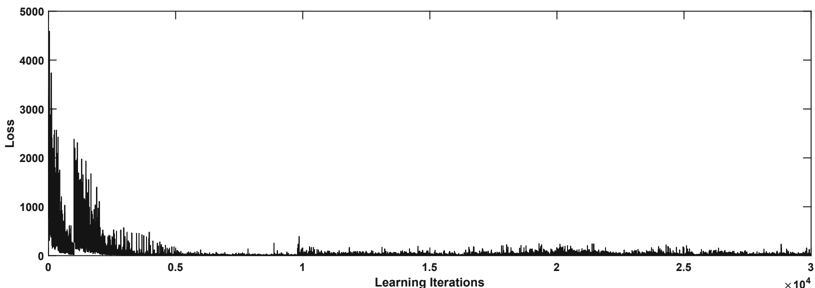

Fig. 2. Loss values obtained while training the DNQN on the DP tractable model

While the early learning exhibits very noisy loss values this seems to stabilize drastically after about 5,000 iterations. Initially this steadiness seemed to imply that our network had essentially finished learning after the first 5,000 updates. To investigate, we ran simulations against an optimal dynamic programming opponent after each update cycle and plotted the average cumulative utility in Fig. 3.

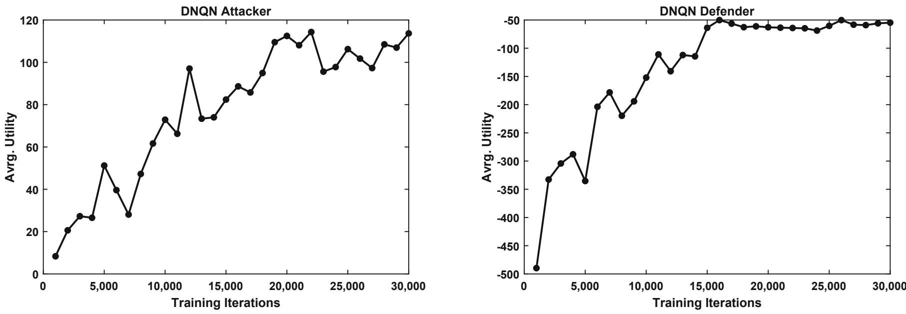

Fig. 3. DNQN agents' utility against a DP opponent at various stages of learning

The results of these simulations show that both agents continue to learn well past the point implied by the loss curve. It is interesting to note that the DNQN Defender seems to reach convergence at around the 15,000 iteration mark while the DNQN attacker continues to progress up until the end of its training. Taken together, Figs. 2 and 3 show that while later iterations' exhibit rather accurate

Q-value predictions (i.e., low loss) the strategies derived from these Q-values continue to evolve throughout the learning process.

After 30,000 training iterations we want to compare our solution methods' policies against one another to understand how well they perform with respect to cumulative utility. We also include two heuristic policies, random and myopic. A random policy for either agent simply randomizes over their action space in each state. A myopic policy plays as if the agent is in a normal form game, solving the payoff matrix of immediate rewards in each state and then sampling from the derived mixed strategy.

For each of the policy pairs we run 1,000 independent simulations with a time horizon of 100 rounds (starting from random states). The average discounted cumulative utility against the optimal dynamic programming opponents are presented in Fig. 4 for the defender and Fig. 5 for the attacker.

Both figures show that our approximate solution maintains a similar utility as its optimal counterpart.

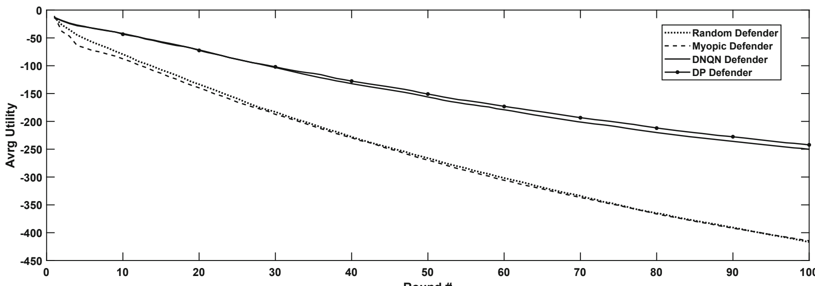

Fig. 4. Cumulative utilities obtained by all defender policies against the optimal DP attacker policy.

5.2 Dynamic Programming Intractable Model

The second set of results we will discuss were obtained within a state space too large to be solved by the dynamic programming approach. The parameters used when constructing the state space are presented in Table 2 and yield an environment with a total of 2.1 billion states. Performing the 30,000 training updates on our DNQN took little over 5.5 h in this environment. For comparison, a liberal estimate obtained by extrapolating the run-times in Sect. 5.1 would put the DP solution time at roughly 114 years.

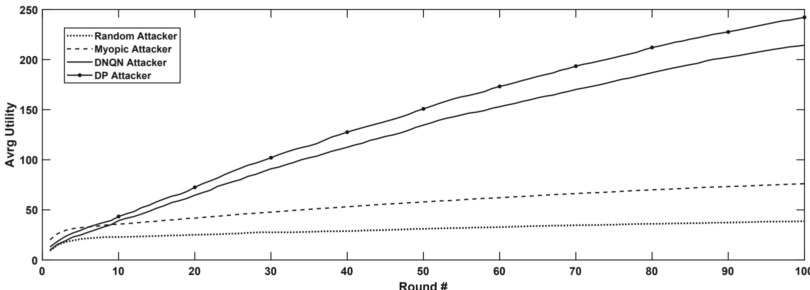

Fig. 5. Cumulative utilities obtained by all attacker policies against the optimal DP defender policy.

Table 2. Parameters used when constructing the DP intractable environment.

Parameter	Value
Number of experts n	8
Attack budget B	30
Utilities $\mathcal{U}$	$u_h = 100,\ u_m = 20,\ u_l = 5$
Attack cost $\mathcal{C}$	$c_h = 8,\ c_m = 4,\ c_l = 2$
Work times $\mathcal{W}$	$w_h = 6,\ w_m = 3,\ w_l = 1$
Alert batches in Ω where $\omega = \langle h, m, l \rangle$	$\omega_1 = \langle 0, 2, 2 \rangle$, $\omega_2 = \langle 0, 3, 3 \rangle$, $\omega_3 = \langle 1, 2, 5 \rangle$ $\omega_4 = \langle 1, 3, 6 \rangle$, $\omega_5 = \langle 1, 4, 4 \rangle$, $\omega_6 = \langle 2, 2, 2 \rangle$ $\omega_7 = \langle 2, 2, 4 \rangle$, $\omega_8 = \langle 2, 3, 4 \rangle$, $\omega_9 = \langle 2, 3, 5 \rangle$ $\omega_{10} = \langle 3, 3, 3 \rangle$, $\omega_{11} = \langle 3, 4, 5 \rangle$, $\omega_{12} = \langle 4, 5, 6 \rangle$
Arrival prob. $\Pi(\Omega)$	$\omega_1 = 0.02$, $\omega_2 = 0.03$, $\omega_3 = 0.08$ $\omega_4 = 0.08$, $\omega_5 = 0.09$, $\omega_6 = 0.10$ $\omega_7 = 0.10$, $\omega_8 = 0.10$, $\omega_9 = 0.10$ $\omega_{10} = 0.13$, $\omega_{11} = 0.12$, $\omega_{12} = 0.05$

In state spaces as large as this it can be very difficult to understand what optimal behavior looks like. In the absence of our optimal DP solution we can make no concrete guarantees as to the efficacy of our results. Despite this fact, our algorithm still maintains a converging loss curve and a superior utility when compared to our previously mentioned random and myopic policies.

Figure 6 shows the loss curve obtained while training the DNQN on the DP intractable model. Similar to Fig. 2 we can see early convergence over the first update cycle followed by a large spike in loss as future rewards begin to be considered.

After training we again want to assess the efficacy of our DNQN policy against the other policies but have lost the DP solution due to the intractability of the environment's size. We run 1,000 independent simulations with a time horizon of 100 rounds to obtain the average cumulative utility in each policy pair using just the DNQN, myopic, and random policies. These utilities are presented for the attacker and defender in Figs. 7 and 8, respectively.

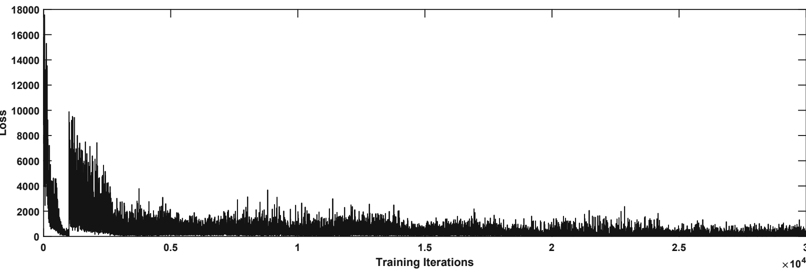

Fig. 6. Loss values obtained while training the DNQN on the DP intractable model

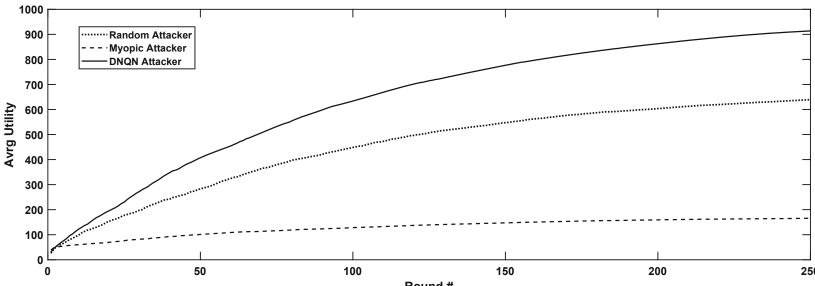

Fig. 7. Cumulative utilities obtained by all attacker policies against the approximate DNQN defender policy.

Figure 7 illustrates a harrowing fact. Due to the vast numbers of alerts, even a random policy can be effective at evading detection. While the random policy does not acquire as much utility as the DNQN attacker, it still performs quite well. The myopic attacker performs poorly in this environment because it acts greedily in each sub-game. This limits the amount of budget the attacker will build up before launching attacks and restricts the myopic attacker to using mostly low severity alerts.

In Fig. 8 the DNQN policy performs much better than myopic and random and shows the importance of employing an intelligent alert allocation strategy. This figure demonstrates the disparity between the attacker and defender's challenges within this domain. When the volume of alerts is so high even a random attacker policy can perform well as the increasing proportion of false positives provide ample camouflage for attacks with no input from the attacker. Contrast this with the defender, whose task only gets more challenging as the volume of alerts increases. This explains why we see a disparity between the relative efficacy of the DNQN policy compared to the myopic and random for the defender and attacker.

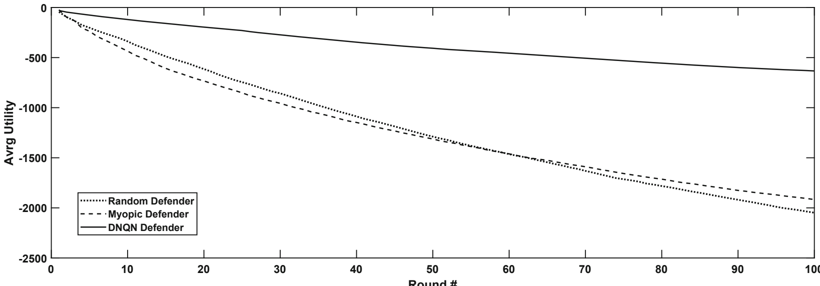

Fig. 8. Cumulative utilities obtained by all defender policies against the approximate DNQN attacker policy.

When investigating further into the DNQN defender's strategy we noticed that it would never fully allocate all of its analysts, preferring to hover around a 75% utilization. Essentially the defender was playing a game of chicken with the attacker, trying to discourage attacks by always keeping some analysts ready for assignment. While this kind of behavior is quite irrational against the random and myopic attackers, the DNQN defender plays very well against the intelligent DNQN attacker – almost quadrupling the utility of the other two policies. This represents a kind of trade-off between general coverage and acute prevention that is the core of a good defender policy.

To show how influential just a few extra analysts can be we ran simulations using the values from Table 2, varying the number of analysts on staff while keeping all other parameters fixed. The results of this experiment are presented in Fig. 9 where we can clearly see a positive correlation between the number of analysts and the defender's utility. Increasing the analyst on staff from five to 7 resulted in a 45% increase in utility.

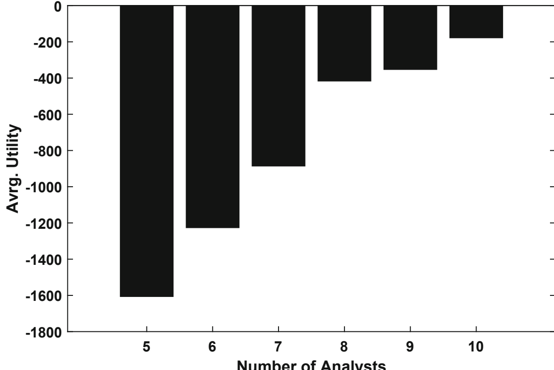

Fig. 9. Comparison of the cumulative utility obtained by a DNQN defender with varying numbers of analysts against a DNQN attacker.

As previously mentioned, a strong attacker policy is much easier to learn than a strong defender policy. The attacker only needs to stockpile their budget (a single resource) until the defender allocates a high percentage of their analysts, whereupon they flood the defender with attacks. Contrast this with the defender who must learn to balance the allocation times of their analysts (multiple resources) with the expected volume of incoming alerts. We find it quite remarkable that we are able to derive two very different policies from a single network while still maintaining a good performance for both tasks.

As for implementing this kind of policy in the real world, defenders would obviously always want to maintain 100% utilization of their analysts and keep those who were unassigned by the model on call for reassignment if the model dictates. Furthermore, the strategies derived by our model could simply be viewed as a kind of threshold for game-theoretically sound behavior that could inform an organizations as to their level of security (or lack thereof). For instance, given the known volume of attacks they face, an organization could run simulations similar to those in Fig. 9 to investigate how many analyst they may need to have on staff to reach an acceptable level of security.

6 Conclusions

In this paper we provided a Markov game framework for modeling the adversarial interaction of computer network attackers and defenders in a game-theoretic manner. By framing this interaction as a series of zero-sum games wherein a state is maintained between each sub-game we were able to simulate long term periods of play and apply reinforcement learning algorithms to derive intelligent policies.

An approximate solution method using our Deep Nash Q-network (DNQN) algorithm was presented that allowed for previous works' results to be extended to much larger state spaces where an explicit model of the environment was not known. This DNQN approach was capable of deriving intelligent policies on par with the optimal approach in a much shorter time and with less information about the environment.

These results motivate the use of DNQN-like architectures when solving very large Markov games as this approach proved to be both computationally expedient and empirically effective.

Acknowledgement. This work was supported in part by NSF grant #1814064.

References

1. Altner, D., Servi, L.: A two-stage stochastic shift scheduling model for cybersecurity workforce optimization with on call options (2016)
2. Brown, G.W.: Iterative solution of games by fictitious play. In: Koopmans, T.C. (ed.) Activity Analysis of Production and Allocation. Wiley, New York (1951)

3. Brown, M., Sinha, A., Schlenker, A., Tambe, M.: One size does not fit all: a game-theoretic approach for dynamically and effectively screening for threats. In: AAAI Conference on Artificial Intelligence (2016)
4. Dunstatter, N., Guirguis, M., Tahsini, A.: Allocating security analysts to cyber alerts using markov games. In: 2018 National Cyber Summit (NCS) (2018)
5. Ganesan, R., Jajodia, S., Cam, H.: Optimal scheduling of cybersecurity analysts for minimizing risk. ACM Trans. Intell. Syst. Technol. **8**, 52 (2015)
6. Ganesan, R., Jajodia, S., Shah, A., Cam, H.: Dynamic scheduling of cybersecurity analysts for minimizing risk using reinforcement learning. ACM Trans. Intell. Syst. Technol. **8**, 4 (2016)
7. Jain, M., Kardes, E., Kiekintveld, C., Ordónez, F., Tambe, M.: Security games with arbitrary schedules: a branch and price approach. In: Proceedings of AAAI (2010)
8. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. CoRR (2014)
9. Lagoudakis, M.G., Parr, R.: Value function approximation in zero-sum markov games. In: Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence, pp. 283–292. Morgan Kaufmann Publishers Inc., San Francisco (2002)
10. Lample, G., Chaplot, D.S.: Playing FPS games with deep reinforcement learning. CoRR abs/1609.05521 (2016)
11. Lin, L.J.: Reinforcement learning for robots using neural networks. Ph.D. thesis, Pittsburgh, PA, USA (1992)
12. Littman, M.: Value-function reinforcement learning in markov games. Princeton University Press (2000)
13. Littman, M.: Markov games as a framework for multi-agent reinforcement learning. In: Proceedings of the Eleventh International Conference on Machine Learning, pp. 157–163. Morgan Kaufmann (1994)
14. Ma, C., Yau, D., Lou, X., Rao, N.: Markov game analysis for attack-defense of power networks under possible misinformation. IEEE Trans. Power Syst. **28**, 1676–1686 (2013)
15. Mnih, V., et al.: Playing atari with deep reinforcement learning. CoRR (2013)
16. Ponemon Institute: The cost of malware containment (2015)
17. Robinson, J.: An iterative method of solving a game. Ann. Math. **54**(2), 296–301 (1951)
18. Schlenker, A., et al.: Don't bury your head in warnings: a game-theoretic approach for intelligent allocation of cyber-security alerts. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17, pp. 381–387 (2017)
19. Schlenker, A., et al.: Towards a game-theoretic framework for intelligent cyber-security alert allocation. In: Proceedings of the 3rd IJCAI Workshop on Algorithmic Game Theory, Melbourne, Australia (2017)
20. Shah, A., Ganesan, R., Jajodia, S., Cam, H.: A methodology to measure and monitor level of operational effectiveness of a CSOC. Int. J. Inf. Secur. **17**(2) (2018)
21. Shu, X., Tian, K., Ciambrone, A., Yao, D.: Breaking the target: an analysis of target data breach and lessons learned. CoRR (2017)
22. Sinha, A., Nguyen, T., Kar, D., Brown, M., Tambe, M., Jiang, A.: From physical security to cybersecurity. J. Cybersecur. **1**(1), 19–35 (2015)
23. Williams, J.D.: The Compleat Strategyst: Being a Primer on the Theory of Games of Strategy. Dover, New York (1986)

24. Xiaolin, C., Xiaobin, T., Yong, Z., Hongsheng, X.: A markov game theory-based risk assessment model for network information system. In: 2008 International Conference on Computer Science and Software Engineering, vol. 3, pp. 1057–1061, December 2008
25. Yin, Z., et al.: Trusts: scheduling randomized patrols for fare inspection in transit systems using game theory. In: Proceedings of the 24th IAAI, Palo Alto, CA (2012)
26. Zimmerman, C.: Ten strategies of a world-class cybersecurity operations center. MITRE corporate communications and public affairs (2014)