Individual Project
Final Report

ID Number: B724890

Programme: Electronic and Computer Systems Engineering

Module Code: 22WSC325

Project Title: Reinforcement Learning for Modelling A Tabletop Cybersecurity Game

Abstract:

This Project is focusing on using Reinforcement learning to model the actions taken in a cybersecurity board game from a red agent perspective. The board game in question is the game Decisions & Disruptions. A simple Monte Carlo algorithm was used for the reinforcement learning. The board game was implemented in python and run successfully. The algorithm managing to update the agent's perceived rewards to find out which attacks were better than others due to causing more damage or being countered by less defences.

Contents

1 Introduction

In this day and age cybersecurity is very important as the amount of cyber-attacks that occur is growing steadily every year [1] With this increase it is important for people to be prepared to have to deal with cybersecurity situations.

The board game Decisions & Disruptions is one way that has been created to test if people are prepared for this increase in cyber-attacks. However, a problem with Decisions & Disruptions is that all the attacks are predetermined in the game, so players can easily learn how to perfectly stop the attacker and the attacker can never react to decisions made by the defenders.

This Project is attempting to make a modified version of Decisions & Disruptions where the attacks are controlled by a reinforcement learning algorithm. With the attacks being chosen this way, the attacker should be able to work out what attacks are the most useful in bypassing the defences chosen by someone.

This report first goes into an explanation of reinforcement learning along with some discussion of several projects using machine learning in cyber security situations. It then goes onto a more detailed description of the Decisions & Disruptions board game along with some description of parts of the process taken when working on adapting it into code. The methodology section goes into more of the details of the implementation of the board game and the reinforcement learning algorithm.

2 Reinforcement Learning

Reinforcement learning is a form of machine learning where the agent being trained is not given the rules of the situation it and instead trains itself based on the results of its own actions. "Beyond the agent and environment, an RL system features four main elements: policy, reward function, value function, and optionally, a model of the environment"[2]. The policy is some form of mapping the current state of the system to the action to be taken by the agent, the reward function either maps each state or combination of a state and action to a number and the value function defines how useful a state will be in the long term.

Some terms used in reinforcement learning that will be used in this report are:

- Red Agent, the reinforcement learning algorithm responsible for hostile actions.
- Blue Agent, the reinforcement learning algorithm responsible for defensive actions.
- Episode, a full run through of the scenario the reinforcement learning agent is being tested in. In the case of this project this is a full game of Decisions & Disruptions.

2.1 Analysis of previous reinforcement learning papers

There have been many previous projects aiming using applications of machine learning to represent cyber security as it can be very complex to work with and how we need to approach it can be constantly changing. This makes reinforcement learning a very useful technique as it doesn't need to know the full details of the system it is interacting with.

The following paragraphs show 3 of these projects.

One of these projects is known as the Intelligent Monitoring and Managing of UNexpected Events (IMMUNE) project [3]. Multiple different reinforcement algorithms were used to represent both the attackers and defenders in a simplified software defined network. Each turn the attacking agent can use a Common Vulnerabilities and Exposure (CVE)[4] to gain more access to the network. One issue with this approach is that CVE's do not account for breaches that make use of social engineering or denial of service (DOS) attacks as they do not make use of errors in the network itself. [3]

The IMMUNE project was broken down into 3 different sections, the IMMUNE model, the IMMUNE risk assessment, and the attack graph. The IMMUNE model is the simplified network that consisting of 5 hosts; the attacker, the webserver, the Secure Shell (SSH) server, the (File Transfer Protocol) FTP server and the database. The IMMUNE risk assessment is the model that calculates what the defender can see based on the actions taken by the attacker and the reliability of the sensors, and finally the attack graph is a graph showing all the actions the attacking agent can make through all the possible states the network can be in. In the IMMUNE model the defender is able to modify the structure of the network to remove one of the hosts from the network.

Another project focusing on using machine learning to simulate cyber security situations is YAWNING TITAN[5] which was created by Defence Science and Technology Laboratory (Dstl). Instead of Reinforcement learning this project chose to use causal inference. Causal inference is another form of machine learning where the algorithm attempts to work out what effect individual decisions have on the whole system.

In YAWNING TITAN, Similarly, to the IMMUNE project, the defence agent is able to isolate nodes of the network. However, the defence agent is able to remove the red agent from an already infected node. One large difference between IMMUNE and YAWNING TITAN is that not all of the red agent's attacks are guaranteed to be successful even if the blue agent has not defended the node. This is due to each node having a corresponding vulnerability score that presents how hard it is to hack. [5]

The full list of actions the blue agent can take are to [[5], p.4]:

- "reduce the vulnerability of nodes",
- "scan the network for red intrusions",
- "remove Red Agent from a node",
- "reset a node back to its initial state",
- "deploy deceptive nodes",
- "isolate a node"
- "reconnect a previously isolated node".

Every time step both agents are allowed to make a single action, however if the red agent wins the scenario from the use of their attack action, the blue agent is prevented from using their defence action in that time step [[5], p.4]. It is possible to limit what actions the agents have access to during the running of the system by editing a configuration file. [5]

In total the nodes of the network are composed of 4 different attributes, the aforementioned vulnerability score, isolation status, true compromised status and blue has seen compromised status. As previously mentioned, the vulnerability score represents how hard the node is to hack. The functions of the other attributes are to show whether the node is currently isolated from the rest of the network, whether the node has been compromised by the red agent, and whether the blue agent has discovered that a node has been compromised.[5]

A much larger scale project that is using deep reinforcement learning instead of regular reinforcement learning is the CyGIL project [6]. CyGIL was a project to create "an experimental testbed of an emulated RL training environment for network cyber operations."[6] Unlike the previously described projects which simply emulated networks, CyGIL is using the CALDERA framework which can also run on a real network instead. The attacking agent's current observation of the network is stored in a 2-dimensional array. The rows of the array represent the individual hosts on the network and the columns consist of information the attacker has gathered on the host and how much access the attacker has to the host.

2 different deep reinforcement learning methods are used for the attacker inside the project. One of these algorithms (Deep Q-Network) is trained on every step of the process while the other (cross-entropy) is trained after every episode is finished.

In the test they describe there are 4 hosts and 2 switches set up in the network, with one of the switches connected to the internet. The attacker is also connected to the internet and is able to interact with the network by going through it. The attacking agent is set up to only be allowed to make 100 actions in an episode before it ends. The agent's target is a single file on one of the hosts in the network and retrieving this file gives the agent a reward of 99, while any other action it takes has a negative reward associated with it.

3 Decisions & Disruptions

Decisions & Disruptions (D&D) is a cyber security board game which consists of a group of players choosing what defences should be installed to prevent cyber-attacks[7]. The game goes through 4 turns and in each turn the players are given an extra 100k currency to spend on the defences. This board game was created to test how different people would act and react in dealing with cyber security situations. They found that there were 4 different "decision-making processes; procedure, experience, scenario and intuition-driven."[7] There were 3 main groups of people tested: security experts, computer scientists and managers. The board game was designed to be created out of lego pieces so that it is very easy to visualise the situation. In the game all the attacks are set out to happen at certain times.

The different defences that can be bought in the game are CCTV, Firewalls, PC upgrades, a server upgrade, an upgrade for the SCADA controller, an antivirus, security training for the staff, network monitoring, encryption for the pcs, encryption for the database, an asset audit, and a threat assessment [7]. One important detail about these defences is that the players are not informed about some of them until the asset audit upgrade has been purchased. Additionally, all that happens when the threat assessment is purchased is that the person running the game will inform the players about what groups are attempting to hack them and their preferred methods. A reinforcement learning algorithm would not be able to use this as it would not be able to understand the information.

The board itself is made up of an office network that has some PCs, a database, a webserver, and a router connecting the network to the internet. There is also a plant network that has some PCs, a database, a SCADA controller that is controlling some turbines and a router connecting the network to the internet as seen in Figure 1.

As the reinforcement learning agent needs a reward based on how much of the networks have been compromised, an attack graph would need to be created taking into account what systems were affected by the attacks used in the game. From analysing the attacks, the only components that were directly accessed/compromised by the attacks was the webserver, general pcs, the office database, the plant database, and the SCADA controller.

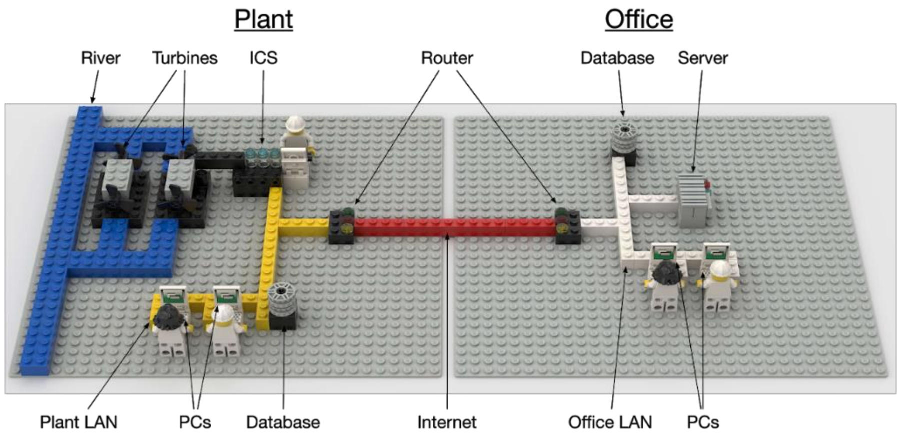


Figure 1 Decisions & Disruptions board [8]

To begin with several attack graphs were created as seen in Figure 2. These were based on the different attack chains in the game with the states of each component being represented by the numbers 0-2 which have the following meanings:

0, the attacking agent has no access to the component.

1, the attacking agent is able to see the component in the network from its current position.

2, the attacking agent has control over the component.

This method to use numbers to level the current level of access was inspired by the attack graph used in the IMMUNE project. Due to the Attacks in D&D being more simplistic than in the IMMUNE project, the numbers of levels of access were lowered as the agent always has full control over the system it has infected or not at all.

Each of the lines between the states are labelled with the action the attacker makes at that point in the attack chain. The attacks made in the game belong to 3 different groups, script Kiddies, organised crime, and nation states, leading to the chains being labelled as Kiddie, Mafia or Nation State in these graphs.

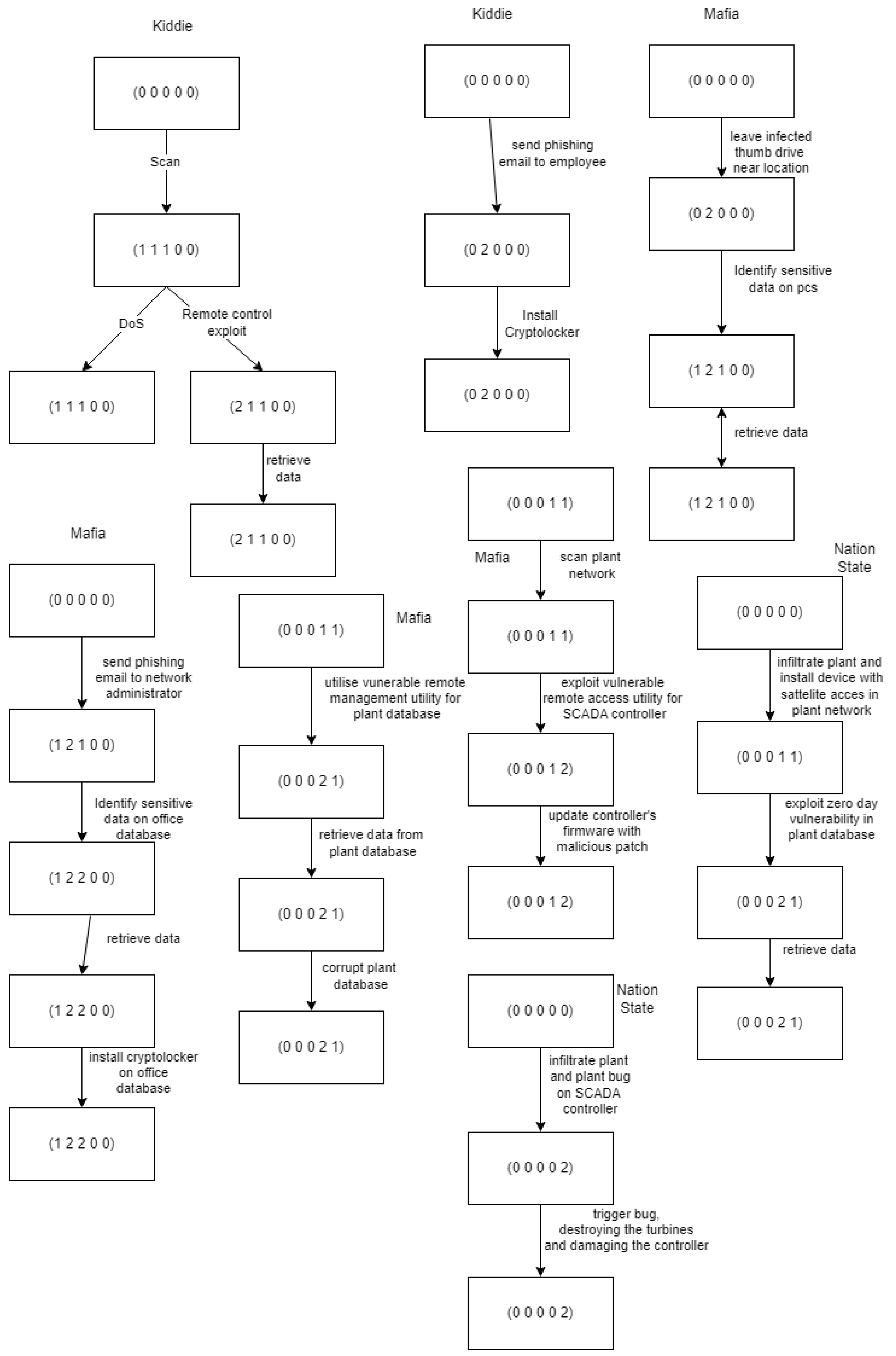

Kiddie
(0 0 0 0 0)
Scan
(1 1 1 0 0)
DoS
Remote control exploit
(1 1 1 0 0)
(2 1 1 0 0)
retrieve data
(2 1 1 0 0)
Kiddie
(0 0 0 0 0)
send phishing email to employee
(0 2 0 0 0)
Install Cryptolocker
(0 2 0 0 0)
Mafia
(0 0 0 0 0)
leave infected thumb drive near location
(0 2 0 0 0)
Identify sensitive data on pcs
(1 2 1 0 0)
retrieve data
(1 2 1 0 0)
Mafia
(0 0 0 0 0)
send phishing email to network administrator
(1 2 1 0 0)
Identify sensitive data on office database
(1 2 2 0 0)
retrieve data
(1 2 2 0 0)
install cryptolocker on office database
(1 2 2 0 0)
Mafia
(0 0 0 1 1)
utilise vunerable remote management utility for plant database
(0 0 0 2 1)
retrieve data from plant database
(0 0 0 2 1)
corrupt plant database
(0 0 0 2 1)
Mafia
(0 0 0 1 1)
scan plant network
(0 0 0 1 1)
exploit vulnerable remote access utility for SCADA controller
(0 0 0 1 2)
update controller's firmware with malicious patch
(0 0 0 1 2)
Nation State
(0 0 0 0 0)
infiltrate plant and plant bug on SCADA controller
(0 0 0 0 2)
trigger bug, destroying the turbines and damaging the controller
(0 0 0 0 2)
Nation State
(0 0 0 0 0)
infiltrate plant and install device with sattelite acces in plant network
(0 0 0 1 1)
exploit zero day vulnerability in plant database
(0 0 0 2 1)
retrieve data
(0 0 0 2 1)

Figure 2 Attack Graph created for Decisions & Disruptions

Later in development this was simplified for the actual program so that each stage has a single reward associated with it. These rewards were still based on the same structure but with the DOS and hacking attacks that come off from the scan attack separated so the agent can easily choose between them. This representation in the code can be seen in Figure 4.

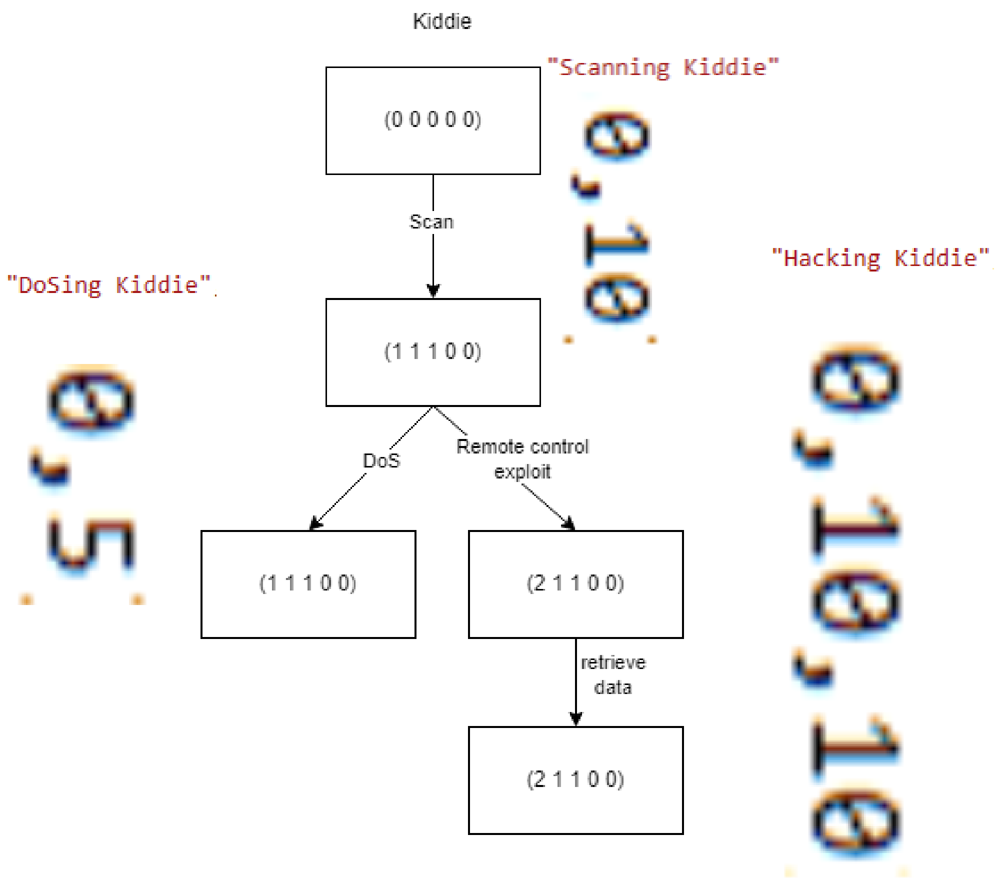


Figure 3 Example of new rewards from diagram

4 Methodology

One big difference between the original game and how it was implemented in this project is that, in the original the attacks were already set but here the game has been implemented so that the attacks can be chosen in different orders. Initially a simulation of the board game itself was created in python. The first thing created was a class that holds the data of the different types of attacks. A class function was created inside the attacks class to check if the defence needed to counter the attack at the current point in the attack chain has been installed. If the defence has been installed the attack gets disabled by this function.

A class for the defences was created as well, which contains the name of the defence, a variable to say if the defence is currently active, the price of the defence, and a variable to say whether asset audit needs to be purchased before that defence is purchased.

At this early stage the functionality of the implementation was tested by having a user input what attacks to make and what defences to purchase. The next class created was a class to keep track of the current state of the game which was named gamestate. The gamestate class would eventually expand to hold; the current budget, how much money was added into the budget each turn, the current turn of the game, an array of the currently active attacks, an array of the currently active defences, a 2D array that contains the real rewards associated with each of the stages of the attack chains.

A major change made from the board game was that the attacks were done in stages. It is possible for the agent to choose multiple stages of a single attack chain during one turn unlike in the board game. Each attack stage has a reward associated with it. The agent begins with having equal perceived rewards for each stage of each attack and will choose to either pick the attack that's next stage gives the highest reward or a random attack out of the remaining attacks. After each episode, the chance for the agent to experiment with its choice is lowered by lowering a variable named certainty. A randomly generated number is tested against the variable certainty to select whether the attack giving the highest reward, or one of the remaining attacks, will be chosen in the function named calculateattack.

This Reinforcement learning algorithm is essentially a monte carlo algorithm with reducing epsilon. This was used as it a very simple algorithm to implement quickly and could be improved to use a more powerful algorithm in the future.

One issue with this method is with the function being used to find the attack stage with the current highest reward. The function being used is argmax which give the position of the largest variable in the list. The issue with this is if multiple variables all share the same largest value it will always pick the first one in the list, which can lead to bias as it would treat the first option in the array as the largest and avoid it when it chooses to experiment. However as soon as it makes an attack that is not countered by a defence, the largest reward would change, and the first option will no longer be biased for no reason.

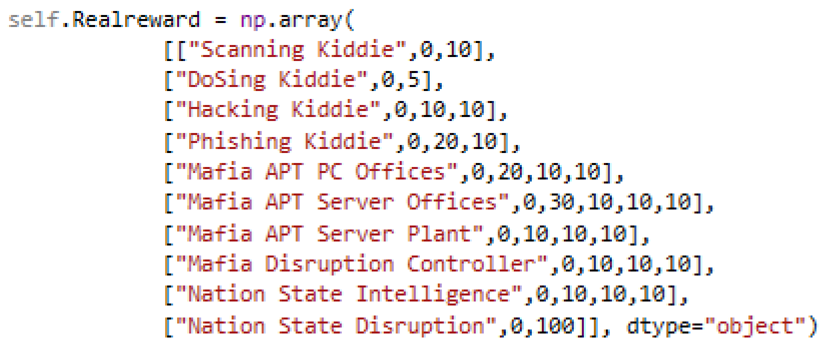
The perceived rewards are updated at the end of each turn when the program checks which attacks made have been disabled by the installed defences. Attacks that were countered have their rewards decreased and attacks that were successful have their rewards increased as an average up to the real reward.

Initially defences had to be chosen manually. To help speed up the process of running the reinforcement algorithm an option was created that would change the flow of the program. This change made it so a piece of code that randomly selected a number of the currently available defences was run. Both the amount of attacks and amount of defences installed can be changed by altering the amount of times their for loops are looped through.

One issue encountered after the code was changed to allow it to run the game multiple times was that the arrays used to reset the state of the game were getting wiped to not contain anything even though they were never accessed. It turned out that with the type of array being used unless you specifically copy the array using its own .copy function, it will actually just be another view of the original array. This caused the original arrays to have their contents removed whenever they were removed from the versions of the arrays being used by the game.

5 Results

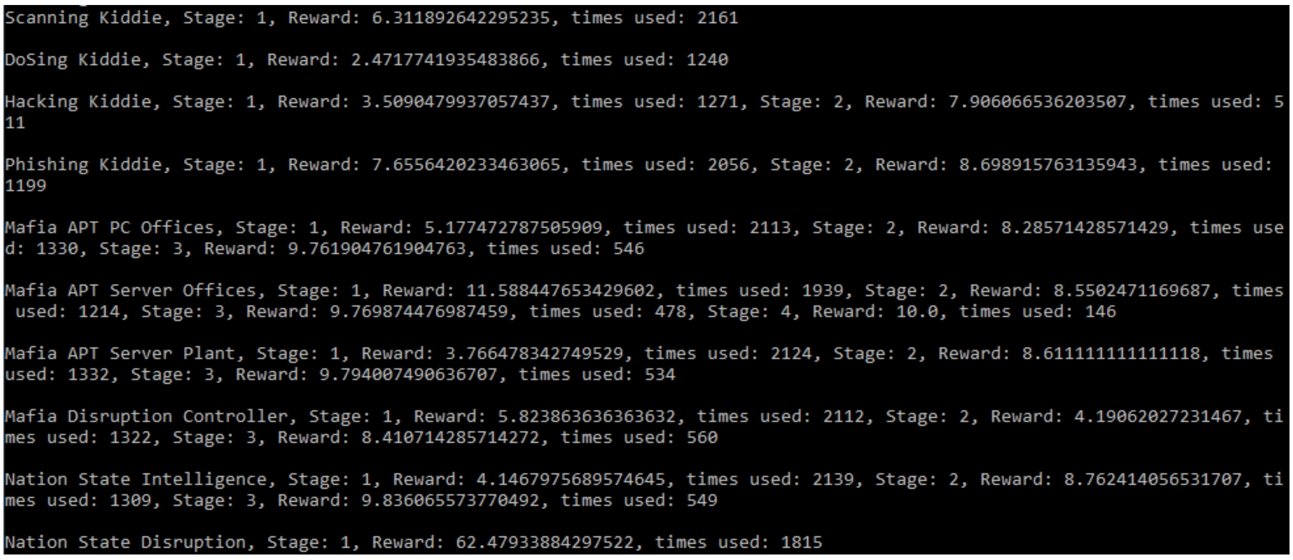
The program has been set up so that, after running through all episodes, it will print what the agent currently perceives the rewards for each of the stages of the attacks. In the tests run the program was set so it would randomly choose 2 defences each turn, and the red Agent would choose 4 attacks each turn. The likelihood to experiment started at 0.9 and decrease by a factor of 0.001 each episode until it reached a chance of 0.1. The program was run for a total of 2500 episodes.



```
self.Realreward = np.array(
            [["Scanning Kiddie",0,10],
            ["DoSing Kiddie",0,5],
            ["Hacking Kiddie",0,10,10],
            ["Phishing Kiddie",0,20,10],
            ["Mafia APT PC Offices",0,20,10,10],
            ["Mafia APT Server Offices",0,30,10,10,10],
            ["Mafia APT Server Plant",0,10,10,10],
            ["Mafia Disruption Controller",0,10,10,10],
            ["Nation State Intelligence",0,10,10,10],
            ["Nation State Disruption",0,100]], dtype="object")
```

Figure 4 Real Rewards array stored in program

The above values are the real rewards given to each attack stage in the game.



```
Scanning Kiddie, Stage: 1, Reward: 6.311892642295235, times used: 2161

DoSing Kiddie, Stage: 1, Reward: 2.4717741935483866, times used: 1240

Hacking Kiddie, Stage: 1, Reward: 3.5090479937057437, times used: 1271, Stage: 2, Reward: 7.906066536203507, times used: 5
11

Phishing Kiddie, Stage: 1, Reward: 7.6556420233463065, times used: 2056, Stage: 2, Reward: 8.698915763135943, times used:
1199

Mafia APT PC Offices, Stage: 1, Reward: 5.177472787505909, times used: 2113, Stage: 2, Reward: 8.28571428571429, times use
d: 1330, Stage: 3, Reward: 9.761904761904763, times used: 546

Mafia APT Server Offices, Stage: 1, Reward: 11.588447653429602, times used: 1939, Stage: 2, Reward: 8.5502471169687, times
 used: 1214, Stage: 3, Reward: 9.769874476987459, times used: 478, Stage: 4, Reward: 10.0, times used: 146

Mafia APT Server Plant, Stage: 1, Reward: 3.766478342749529, times used: 2124, Stage: 2, Reward: 8.611111111111118, times
used: 1332, Stage: 3, Reward: 9.794007490636707, times used: 534

Mafia Disruption Controller, Stage: 1, Reward: 5.823863636363632, times used: 2112, Stage: 2, Reward: 4.19062027231467, ti
mes used: 1322, Stage: 3, Reward: 8.410714285714272, times used: 560

Nation State Intelligence, Stage: 1, Reward: 4.1467975689574645, times used: 2139, Stage: 2, Reward: 8.762414056531707, ti
mes used: 1309, Stage: 3, Reward: 9.836065573770492, times used: 549

Nation State Disruption, Stage: 1, Reward: 62.47933884297522, times used: 1815
```

Figure 5 Print out of perceived rewards after finishing all episodes

Figure 5 shows the values that the agent has calculated as the rewards of each attack stage, the printout also shows how many times the agent has used each attack stage. We can see that the later into an attack chain the less attacks have been used to being stopped at earlier points. can see that the Dos and hacking attacks have been used significantly less than other attacks. This will be caused by the fact that prior to them being chosen the scan attack also has to be chosen.

One seeming oddity is that the nation state disruption attack has been used less than other attacks even though its reward is much higher. This could easily be caused by the explore/exploit function as that removes the attack with the highest reward to allow for other attacks to be chosen. To test this hypothesis, we would need to make a graph of when these decisions were made.

It's important to note that the real rewards are the rewards that would only be received if an attack was not stopped. Consequentially the perceived reward will always be lower or equal the real reward and attacks that have more counters over more stages will have lower rewards.

As we can see by comparing Figure 4 and Figure 5, most of the agents perceived rewards for the first stage of an attack chain are around 50% of the real reward. Later stages are closer to their real rewards which is potentially due to the fact that a lot of those later stages require counters locked behind the asset audit defence. The chance of these defences being chosen would be much lower as the defence would need to pick asset audit and then another defence while being completely random.

6 Conclusion

This project has successfully implemented an adaptation of the Decisions & Disruptions board game. This adaptation of Decisions & Disruptions has been modified to allow the attacks to be chosen by a reinforcement learning algorithm instead of them being used in a set order. Using the code implemented, the red agent was able to find the best attacks based on which attacks cause the most damage and which attacks have been stopped by the defence previously.

There are several improvements that could be made to this project in the future. For one a defence agent could also be created so the defences can react to the attacks. The current simple Monte Carlo algorithm used for the defence agent could be replaced with a more sophisticated one i.e. Q-learning. The program could also be split into multiple files with the different classes being referenced between them.

7 References

[1] R. Gupta, Dr S.P. Agarwal, “A Comparative Study Of Cyber Threats In Emerging Economies” Globus An International Journal of Management & IT, vol. 8, no.2, pp. 24-28, Jan 2018

[2] A. Ridley, “Machine learning for autonomous cyber defense” The Next Wave, vol. 22, no. 1, pp. 7–14, Jan. 2018.

[3] V. Varris, "Using adversarial reinforcement learning to evaluate the immune risk assessment," France: Sophia Antipolis: Eurecom, 2020 (2.1)

[4] CVE Numbering Authority (CNA), CVE Numbering Authority (CNA) Rules, MITRE Corporation, 2020. [Online] Available from: https://www.cve.org/ResourcesSupport/AllResources/CNARules [Accessed 29/04/2023].

[5] A. Andrew S. Spillard J. Collyer 1 N. Dhir, "Developing Optimal Causal Cyber-Defence Agents via Cyber Security Simulation" in International Conference on Machine Learning, (39th ICML), July 2022, pp. 1-10

[6] L. Li1 , R. Fayad , A. Taylor , "CyGIL: A Cyber Gym for Training Autonomous Agents over Emulated Network Systems" in International Workshop on Adaptive Cyber Defense, August 2021, pp. 1-8

[7] S. Frey , A. Rashid, P. Anthonysamy, M. Pinto-Albuquerque, and S. A. Naqvi, "The Good, the Bad and the Ugly: A Study of Security Decisions in a Cyber-Physical Systems Game", IEEE Transactions on Software Engineering, VOL. 45, NO. 5, pp. 521-536, May 2019

[8] B. Shreeve, A. Rashid, Decisions & Disruptions, Lancaster University, 2018. [Online] Available from: https://www.decisions-disruptions.org/ [Accessed 1/10/2022]

8 Appendices

8.1 Code

```
import os
import numpy as np

class gamestate(object):
    def __init__(self, Agent):
        self.currentBudget = 100
        self.budgetIncrease = 100
        self.currentTurn = 1
        self.maxturns = 4
        self.Agent = Agent
        self.possible_Attacks = []
        self.available_attacks = []
        self.active_attacks = []
        self.active_defences = []
        self.available_defences = []
        self.possible_defences = []
        self.Realreward = np.array(
                    [["Scanning Kiddie",0,10],
                    ["DoSing Kiddie",0,5],
                    ["Hacking Kiddie",0,10,10],
                    ["Phishing Kiddie",0,20,10],
                    ["Mafia APT PC Offices",0,20,10,10],
                    ["Mafia APT Server Offices",0,30,10,10,10],
                    ["Mafia APT Server Plant",0,10,10,10],
                    ["Mafia Disruption Controller",0,10,10,10],
                    ["Nation State Intelligence",0,10,10,10],
                    ["Nation State Disruption",0,100]], dtype="object")
    #increment the turn
    def incrementturn(self):
        self.currentTurn += 1
        self.currentBudget += self.budgetIncrease
        totalattacks = len(self.active_attacks)
        for i in range(totalattacks):
            self.active_attacks[totalattacks-i-1].attack_check(self)
    #return the current budget
    def getbudget(self):
        return self.currentBudget
    #pay for defence
    def payment(self,price):
        self.currentBudget -= price
    #add defence to active defences
    def addDefence(self, name):
        self.active_defences.append(name)
        print(self.active_defences)
    #reset game for new episode
    def resetGame(self, currentBudget, budgetIncrease, availableAttacks,
availableDefences):
        self.currentBudget = currentBudget
        self.budgetIncrease = budgetIncrease

        self.available_attacks = availableAttacks.copy()
        self.available_defences = availableDefences.copy()
        self.active_attacks = []
        self.active_defences = []
        self.currentTurn = 1
        for i in self.possible_Attacks:
            i.stage = 0
            i.prevstage = 0



class defences(object):
    def __init__(self, name, target, price, needAssetAudit):
        self.name = name
        #what component it defends
        self.target = target
        self.active = False
        self.price = price
        self.needAssetAudit = needAssetAudit
    #activates the defense
    def defense_activate(self, Game):
        if self.needAssetAudit == True:
            for i in Game.active_defences:
                if i == "Asset Audit":
                    if self.price < Game.getbudget():
                        self.active = True

        elif self.price < Game.getbudget():
            self.active = True
        if self.active == True:
            Game.payment(self.price)
            Game.addDefence(self.name)
            Game.available_defences.remove(self)
            if self.name == "Asset Audit":
                for i in Game.possible_defences:
                    if i.needAssetAudit:
                        Game.available_defences.append(i)
        return Game



class attacks(object):
    def __init__(self, name, counters, group,maxstage):
        self.name = name
        #what defences counter the attack
        self.counters = counters
        #what component does the attack effect
        #what reward will the reinforcement learning algorithm get
        self.group = group
        self.prevstage = 0
        self.stage = 0
        self.maxstage = maxstage


            # checks if a defense that counters an attack is already installed
to see if the attack is succesful
    def attack_check(self, Game):
        passcheck = True
            #loops through defences that counter attack

        turncheck = 1


        stagestopped = -1
        #get list of names on the counters
        for i in self.counters[::2]:
            turncheck = 1
            #get the stages the defences counter them on
            for y in self.counters[turncheck]:

                #check if stage has changed this turn
                if self.prevstage != self.stage:
                    if y-1 in range(self.prevstage, self.stage):
                        #check if defence is installed
                        if i in Game.active_defences:
                            if y < stagestopped or stagestopped == -1:
                                stagestopped = y
                            passcheck = False;
                            break
                    if passcheck == False:
                            break
                else:
                    if y == self.stage:
                        if i in Game.active_defences:
                            if y < stagestopped or stagestopped == -1:
                                stagestopped = y
                            passcheck = False;
                            break
                        if passcheck == False:
                            break
                turncheck+= 2
        #deactivate attack if check has been triggered
        if passcheck == False:
            Game.active_attacks.remove(self)
            print(self.name + " disabled")
        Game.Agent.train(self.name, self.stage, stagestopped, self.prevstage)
        self.prevstage = self.stage
        return Game
    #activate attack
    def attack_activate(self, Game):
        self.stage += 1
        if self not in Game.active_attacks:
             Game.active_attacks.append(self)
        print(self.name + " chosen")
        #check if attack chain has finished
        if self.stage == self.maxstage:
            Game.available_attacks.remove(self)

            #check if scan has been completed resulting in Dos and Hacking being
added
            if self.name == "Scanning Kiddie":
                check = 0
                for i in Game.possible_Attacks:
                    if i.name == "DoSing Kiddie":
                        Game.available_attacks.append(i)
                        check += 1
                    elif i.name == "Hacking Kiddie":
                        Game.available_attacks.append(i)
                        check += 1
                    if check == 2:
                        return Game

        return Game


class Agent(object):
    def __init__(self):
        self.certainty = 0.9
        self.Perceivedreward = np.array([["Scanning Kiddie",0,10],
                    ["DoSing Kiddie",0,0],
                    ["Hacking Kiddie",0,0,0],
                    ["Phishing Kiddie",0,0,0],
                    ["Mafia APT PC Offices",0,0,0,0],
                    ["Mafia APT Server Offices",0,0,0,0,0],
                    ["Mafia APT Server Plant",0,0,0,0],
                    ["Mafia Disruption Controller",0,0,0,0],
                    ["Nation State Intelligence",0,0,0,0],
                    ["Nation State Disruption",0,10]], dtype="object")
        self.attackamount = np.array([[0],
                    [0],
                    [0,0],
                    [0,0],
                    [0,0,0,0],
                    [0,0,0,0],
                    [0,0,0],
                    [0,0,0],
                    [0,0,0],
                    [0]], dtype="object")
        self.totaluses = np.array([[0],
                    [0],
                    [0,0],
                    [0,0],
                    [0,0,0,0],
                    [0,0,0,0],
                    [0,0,0],
                    [0,0,0],
                    [0,0,0],
                    [0]], dtype="object")
    #choose what attack will be inputted based on current perceived rewards
    def calculateAttack(self, Game):
        options = []
        stages= []
        k = 0
        for i in Game.available_attacks:
            options.append(i.name)
            stages.append(i.stage)
        predictedincrease = []
        k = 0
        for i in options:
            predictedincrease.append(self.predict(i,stages[k]))
            k += 1
        #if randomly generated number is higher than current certainty return
available attack stage with current highest perceived reward
        #if not return a different randomly selected attack
        if len(options) == 1:
            return options[0]
        elif np.random.random() > self.certainty:
            return options[np.argmax(predictedincrease)]
        else:
            options.remove(options[np.argmax(predictedincrease)])
            if k == 2:
                return options[0]
            else:
                p = np.random.randint(k-1)
                return options[p]




            #return current perceived reward of an attack
    def predict(self, Attackname,stage):
        for i in range(len(self.Perceivedreward)):
            if Attackname == self.Perceivedreward[i][0]:
                return self.Perceivedreward[i][stage+2]



    #update perceived rewards
    def train(self, Attackname, stage, stoppedstage, prevstage):
        for i in range(len(self.Perceivedreward)):
            if Attackname == self.Perceivedreward[i][0]:
                #check if the attack has been stopped
                if stoppedstage != -1:
                    for j in range(stage):
                        #update perceived reward
                        if j>prevstage-1:

                            oldvalue = self.Perceivedreward[i][j+2] *
self.totaluses[i][j]
                            newvalue = int(Game.Realreward[i][j+2]) + oldvalue
                            #if attack was stopped lower reward
                            if j >= stoppedstage-1:
                                newvalue = 0 + oldvalue
                            self.attackamount[i][j] += 1
                            self.totaluses[i][j] += 1
                            newvalue = newvalue/self.totaluses[i][j]
                            self.Perceivedreward[i][j+2] = newvalue

                    break

                else:
                    for j in range(stage):
                        if j>prevstage-1:
                            oldvalue = self.Perceivedreward[i][j+2] *
self.totaluses[i][j]
                            newvalue = int(Game.Realreward[i][j+2]) + oldvalue
                            self.totaluses[i][j] += 1
                            self.attackamount[i][j] += 1
                            newvalue = newvalue/self.totaluses[i][j]
                            self.Perceivedreward[i][j+2] = newvalue

                    break


        return Game
    #print the current perceived rewards
    def printrewards(self):
        for i in range(len(self.Perceivedreward)):
            for j in range(len(self.Perceivedreward[i])):
                if j == 0:
                    print(str(self.Perceivedreward[i][j]), end = '')
                elif j>1:
                    print(", Stage: " + str(j - 1) + ", Reward: " +
str(self.Perceivedreward[i][j]) + ", times used: " + str(self.totaluses[i][j-
2]), end = '')
            print("")
            print("")

    def updatecertainty():
        if self.certainty > 0.1:
            self.certainty *= 0.999
```

```
if __name__ == '__main__':

   #setting up Agent and gamespace objects
    Attacker = Agent()
    Game = gamestate(Attacker)
    #seting up defence objects
    sectraining = defences("security training","everything",30,False)
    firewallPlant = defences("Firewall (Plant)","plant",30,False)
    firewallOffice = defences("Firewall (Office)","office",30,False)
    cctvPlant = defences("CCTV (Plant)","plant",50,False)
    cctvOffice = defences("CCTV (Office)","office",50,False)
    netMonPlant = defences("Network Monitoring (Plant)","plant",50,False)
    netMonOffice = defences("Network Monitoring (Office)","office",50,False)
    antiVirus = defences("Anti Virus","everything",30,False)
    assetAudit = defences("Asset Audit","everything",30,False)
    threatAssesment = defences("Threat Assesment","everything",20,False)
    servUp = defences("Server Upgrade","server",30,True)
    pcEnc = defences("PC Encryption","pcs",20,True)
    dbEnc = defences("Database Encryption","database",20,True)
    contUp = defences("Controller Upgrade","Scata controller",30,True)
    pcUp = defences("Pc Upgrade","pcs",30,True)
    #set up attack objects
    SK = attacks("Scanning Kiddie",["Firewall (Office)",[1]],0,1)

    DK = attacks("DoSing Kiddie",["Firewall (Office)",[1]],0,1)

    HK = attacks("Hacking Kiddie",["Server Upgrade",[1], "Network Monitoring
(Office)",[2],"Database Encryption",[2] ],0,2)

    PK = attacks("Phishing Kiddie",["security training",[1], "Anti
Virus",[1,2],"Pc Upgrade",[1,2]],0,2)

    MAPTPC = attacks("Mafia APT PC Offices",["security training",[1], "Anti
Virus",[1,2,3],"Network Monitoring (Office)",[2,3],"PC Encryption",[3] ],1,3)

    MAPTPSO = attacks("Mafia APT Server Offices",["security training",[1],
"Network Monitoring (Office)",[2,3,4],"PC Encryption",[3,4] ],1,4)

    MAPTPSP = attacks("Mafia APT Server Plant",["Asset Audit",[1], "Server
Upgrade",[2],"Network Monitoring (Plant)",[2,3],"Database Encryption",[3] ],1,3)

    MDC = attacks("Mafia Disruption Controller",["Firewall
(Plant)",[1,2],"Controller Upgrade",[2,3]],1,3)

    NSI = attacks("Nation State Intelligence",["CCTV (Plant)",[1],"Network
Monitoring (Plant)",[2,3]],2,3)

    NSD = attacks("Nation State Disruption",["CCTV (Plant)",[1]],2,1)
    possibleAttacks = [SK, DK, HK, PK, MAPTPC, MAPTPC, MAPTPC, MAPTPSO, MAPTPSP,
MDC, NSI, NSD]
    availableAttacks = [SK, PK, MAPTPC, MAPTPSO, MAPTPSP, MDC, NSI, NSD]
    availableDefences = [sectraining, firewallPlant, firewallOffice, cctvPlant,
cctvOffice, netMonPlant, netMonOffice, antiVirus, assetAudit, threatAssesment]
    possibleDefences = [sectraining, firewallPlant, firewallOffice, cctvPlant,
cctvOffice, netMonPlant, netMonOffice, antiVirus, assetAudit, threatAssesment,
servUp, pcEnc, dbEnc, contUp, pcUp]



    Game.possible_Attacks = possibleAttacks
    Game.possible_defences = possibleDefences
    numofepisodes = 2500
    defencechoice = 0
    print("Enter 1 for defences to be entered randomly or 2 to enter defences 
manually")
    while defencechoice != "1" and defencechoice != "2":
        defencechoice = input()
    for episode in range(numofepisodes):
        #reset gamestate for new epispode
        Game.resetGame(100, 100, availableAttacks, availableDefences)
        while Game.currentTurn-1 < Game.maxturns:
            choice = ''
            while choice != '2':
                print("current Money is " + str(Game.currentBudget))
                if defencechoice == "2":
                    print("Enter 1 to enter a defence, Enter 2 to End the turn, 
Enter 3 to print the current perceived reward, Enter 4 to manually enter an 
attack if debugging is needed")
                    choice = input()
                elif defencechoice == "1":
                    #choose randome defences
                    for i in range(2):
                        defenceindex = 
np.random.randint(len(Game.available_defences))

Game.available_defences[defenceindex].defense_activate(Game)
                    choice = "2"
                #clear console
                os.system('cls')
                print("episode number " + str(episode + 1))
                print("turn number is " +str(Game.currentTurn))
                if choice == '1':
                    for i in range(len(Game.available_defences)):
                        print ("Enter " + str(i + 1) + " for " + 
Game.available_defences[i].name)
                    i = input()
                    if i.isnumeric():
                        Game.available_defences[int(i)-1].defense_activate(Game)
                elif choice == '2':
                    for i in range(3):
                        if len(availableAttacks) != 0:
                            attack = str(Attacker.calculateAttack(Game))
                            for i in Game.possible_Attacks:
                                if attack == i.name:
                                    Game = i.attack_activate(Game)
                                    break
                    Game.incrementturn();
                elif choice == '3':
                    Game.Agent.printrewards()

                elif choice == '4':
                    attackname = input()
                    for i in Game.possible_Attacks:
                        if i.name == attackname:
                            Game = i.attack_activate(Game)
                            break



        Game.Agent.updatecertainty;
    Game.Agent.printrewards()
```