Artificial Intelligence in Computer Security: Detection, Temporary Repair and Defense

by

Solofoarisina Arisoa Randrianasolo, M.S.

A Dissertation
In
Computer Science

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

DOCTOR OF PHILOSOPHY

Approved

Dr. Larry D. Pyeatt
Chairperson of the Committee

Dr. Susan Mengel
Dr. Nelson J. Rushton
Dr. Sunho Lim

Peggy Gordon Miller
Dean of the Graduate School

May, 2012

ACKNOWLEDGEMENTS

I would like to thank my Malagasy and my American parents, the Randrianasolos and the Brooms, for supporting me spiritually, mentally and financially through school. This work would not be possible without their support.

I want to show my appreciation to Dr. Larry D. Pyeatt for providing me guidance over my research. His advice enabled me to explore new ideas in the computer science world.

My appreciation also goes to all my numerous friends in Abilene and in Lubbock, my church, and Texas Tech's Computer Science department. Thank you for always encouraging me.

TABLE OF CONTENTS

ABSTRACT

Computer security system providers are unable to provide timely security updates. Most security systems are not designed to be adaptive to the increasing number of new threats. Companies lose considerable amount of time and resources when security attacks manifest themselves. As an answer to these problems, this research is aimed at developing security systems capable of learning and updating themselves. The goal is to create security systems that will autonomously mature with exposure to threats over time. To achieve this goal, this research is exploring learning techniques from the Artificial Intelligence field. This research is proposing artificial intelligence based security systems with learning capability to perform intrusion detection, temporary repair and diagnostics, and defending a network. For network intrusion detection, this research is proposing the utilization of an Artificial Immune System based on Holland's Classifier. A Q-learning approach is proposed to provide a self learning temporary repair and diagnostic mechanism for attacked software. Finally, a General Game Player approach is used as a network defender designed to fight unknown attackers. These approaches are trained and tested with simulations employing DARPA's dataset. Despite the need for an initial training time and the massive use of memory, these approaches appear to have the ability to learn and are in close competition with the other approaches that were tested on the same dataset.

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Computers have become ubiquitous in today's life style. As the use of computers increases, the desire to secure the information stored in computers increases. The field of computer security is interested in providing integrity, availability and confidentiality of information stored in a single computer and of information stored in a system of computers. The majority of the actual computer security systems consist of rule based systems. A rule based system is a software that holds a list of rules to be applied when security violations are manifesting. The computer security systems are equipped with pre-constructed rules from either the vendors or the creators. These rules can be updated and expanded via a procedure called security update. Depending on the security systems used, the computer users or the computer network administrators may or may not have the possibility to add rules to the security systems.

During a security attack, if the security system has no rules to handle the attack, then the computer user or the network administrator has to shut the computer system(s) down and restart it(them) in a safe mode. Then, the user or the administrator has to find a security update or a security patch from either the security system's vendor or the attacked software creator to repair the problem. If an update does not exist, then the user or the administrator has to spend time and resources investigating the security attack in order to manually remove the problem. This situation slows down the productivity of the organization or the individual that uses the computer or the network of computers. In addition, the security attacks are evolving at a faster rate. One attack can be mutated in different forms and can be equipped

with a stealth mode. This situation complicates defending computer system(s) since most of the built-in rule based security systems lack adaptivity to new attacks.

Solving the above situations requires solving the following questions. Is it possible to create a security system that is capable of learning by itself to create new security rules without waiting for security updates? Is it possible to create a software that is capable of repairing or patching itself after a security attack happens? Can software and network infrastructures be equipped with the capability to learn by themselves to create defense procedures from attack experiences?

Computer security systems based on artificial intelligence(AI) methods are proposed in this dissertation to help computer system administrators spend less time repairing computer security problems. These proposed security systems will accelerate the deployment of security updates and repairs. These computer security systems are intended to provide a better level of adaptivity and availability.

1.1 What is Artificial Intelligence?

It is difficult to give one universally accepted definition of Artificial Intelligence. The following definitions, provided by various researchers, will be used to define the term Artificial Intelligence.

- Artificial Intelligence is the study of ideas that enable computers to be intelligent (Henry and Winston, 2004).
- Artificial Intelligence(A.I) is the study of how to make a computer do things at which, at the moment, people are better (Rich and Knight, 1990).
- Artificial Intelligence is a branch of computer science whose objective is to endow machines with reasoning and perceptual abilities (Webber and Nilsson, 1981).
- Artificial Intelligence is the part of computer science concerned with designing

intelligent computer systems, that is, systems that exhibits the characteristics we associate with intelligence in human behavior–understanding language, learning, reasoning, solving problems, and so on (Barr and Feigenbaum, 1981).

- The object of research in AI is to discover how to program a computer to perform the remarkable functions that make up human intelligence. This work leads not only to increasingly useful computers, but also to an enhanced understanding of human cognitive process, of what it is that we mean by "intelligence" and what the mechanisms are that are required to produce it (Shapiro, 1992).

The artificial intelligence field has a historical root with the field of philosophy and psychology. The early intentions were to mimic and understand humans' behaviors. This goal is currently extended to also include mimicking and understanding humans deep into the biological level. The artificial intelligence field is composed of and not limited to the following areas: knowledge reasoning, logic, planning, reinforcement learning, natural language processing, neural network, genetic algorithm, pattern recognition, classifiers, artificial immune systems, robotics and probabilistic reasoning (Russell et al., 1996). The next few chapters will focus on some of the areas listed previously. These chapters will discuss the utilization of some of these artificial intelligent areas in computer security.

1.2 What is Computer Security?

The following definition describes the term computer security. Computer security is, *"the protection afforded to an automated information system in order to attain the applicable objectives of preserving the integrity, availability and confidentiality of information system resources (includes hardware, software, firmware, information/data, and telecommunications)"*(Guttman and Roback, 1995). This definition is further explained with the following definitions.

- Confidentiality:

Confidentiality means preventing unauthorized parties from knowing and accessing information in the computer system (Taylor et al., 2009). The term secrecy is used interchangeably with confidentiality. Use of encryptions is one method to implement confidentiality.

- Integrity:

  Preserving integrity means only authorized parties are allowed to access information (Taylor et al., 2009). This imposes creating users identifications which will be checked at different levels before granting access authorizations. The read and write permission, regularly encountered with operating system files, are examples of access authorizations.

- Availability:

  A requirement intended to assure that systems work promptly and service is not denied to authorized users (Guttman and Roback, 1995).

The previous definition is regularly known as the CIA-Triad. It is worth mentioning that hackers and malicious individuals have established their own triad, Disclosure, Alteration, and Denial (Solomon and Chapple, 2005), which is called DAD-Triad.

- Disclosure is the malicious individuals' response to confidentiality. Disclosure happens when unauthorized individuals or agents gain access to confidential information (Solomon and Chapple, 2005).

- Alteration is the malicious individuals' response to integrity. Alteration happens when unauthorized individuals or agents are making changes to the data or to the information in the computer system.

- Denial is the malicious individuals' response to availability. Denial happens when authorized users are denied legitimate access to the computer system.

1.3 Types of Computer Security

There exist many types of Computer Security. Computer security, in general, can be expressed in three different forms: application security, host security, and network security (Oppliger, 2000). Application security consists of securing the functionalities of an application. This type of security involves checking that the application does not present vulnerabilities that can be exploited for abnormal operations. Application security is interested in providing integrity, availability and confidentiality to the software and the data in a computer system.

The host security's objective is to protect a single host from being attacked by malicious entities. This type of security is concerned with hardware and firmware protection. Host security can be expressed in the form of anti-virus in a personal computer or a server. A host security can also be referred to as a physical security (Russell and Gangemi, 1991).

Network security consists of guaranteeing that no malicious activities happen in a network of computer systems. This type of security deals with spoofing, denial of service, intrusion detection and any other network based attacks. Network security, also known as communication security, is also concerned with protecting the transmission of information (Russell and Gangemi, 1991). Tools that perform packets sniffing, like Wireshark (Wireshark), Snort (Koziol, 2003), and TippingPoint (TippingPoint), are available to provide network security. It is highly preferable for hosts participating in a network of computers to own a host security. The applications, in the hosts participating in the network, need also to be equipped with an application security.

The majority of the current security systems, for computer networks, involve a network administrator monitoring one or more of the tools listed above. The network administrator, with the assistance of other analytical tools, takes actions when a security breach is detected. Snort, TippingPoint and their variations are rule based

network security tools. They come with security rules implemented in them. The network administrator, in most of the cases, can add more rules. The tools' manufacturers can add rules to the tools via updates.

Similar situation happens with the anti-virus and anti-malwares software designed to provide host security. The security software come with predefined security rules. Depending on the software, the user may or may not be able to add rules. Updates are available from the software's manufacturers to improve the built in security rules. The security rules, in the anti-virus world, are also known as viruses definitions.

1.4 Why is AI Needed in Computer Security?

The combination of AI and Computer Security can sound odd to many researchers. This oddity can be explained by the fact that AI researchers are concerned with making computers perform tasks that are naturally performed by a human, while the computer security researchers are concerned with fixing the leak in the computer infrastructures or architectures (Landwehr, 2008). From this situation, it appears that it is unlikely that the two fields will ever meet.

The development of technologies brought artificial intelligence and computer security close to each other. Computer systems are not ceasing to gain intelligent capabilities. The advance in self configuration and in automation in today's computer systems triggered the development of computer systems that are soon expected to be able to defend and secure themselves with less human intervention.

Maintaining the security of computer systems is a complex and vast task. In the network case, for example, the complexity is caused by the development of network infrastructures that started from two computers communicating to each other to a Local Area Network which now develops to a Cloud network. This increases the difficulty involved with administering a computer network. Having an artificially

intelligent network infrastructure that can learn to detect, report and repair network security problems is an advantage to network administrators.

Adaptability is another key reason that brought AI and computer security close to each other. The malicious entities that generate computer security attacks have gained intelligence. The type of attacks evolved from a simple password guessing to a staged and distributed network attack that can be equipped with a stealth mode and attack mutation. The computer security field has to adapt to the rapid change and the evolution of security attacks. AI is well situated to answer this situation. AI is concerned with creating systems that evolve and react depending on the changes in the surrounding environment. AI has a sub-area like reinforcement learning that can equip a computer system with the ability to learn by itself how to adapt to new security threats.

In summary, the rapid development of computer systems and the increase of sophistication in computer security attacks make AI a valuable field in assuring computer security for the future. Whether AI will be used as an assistant or be used as the full time security administrator, it is highly probable that AI will be utilized in computer security since the computer systems' world is turning to more adaptive and automated systems.

1.5 The Goal of this Research

This research is aimed to illustrate how artificial intelligence techniques can be used in computer security. The main goal is to build security systems that are capable of automatically learning by themselves to detect, repair and defend computer systems against attacks. Not all the areas of artificial intelligence will be discussed in this research. As previously presented, the artificial intelligence field is vast and studying how to employ all the areas of artificial intelligence in computer security cannot fit in one dissertation. For the same reason, not all possible security attacks

will be analyzed in this dissertation.

By creating computer security systems that are able to learn how to detect, repair and defend computer infrastructures, this research hopes to provide mechanisms useful for creating an automatic and adaptive security systems. This research also hopes to provide alternative computer security systems other than the built-in rule based computer security systems. Finally, this research will contribute to reducing the human's interventions in the computer security maintenance by raising the security systems' level of cognition and automation.

1.5.1 The Content of this Research

This research will use some of the artificial intelligence areas such as genetic algorithm, classifier system, reinforcement learning, Q-learning, general game playing and artificial immune system. The next chapters will provide further details about each one of these areas. Each one of the AI areas listed previously will become the main strategy utilized to create a computer security system. This research is oriented to studying how these AI areas can be used to detect intrusions and misuses. This research is also oriented to studying how these AI areas can be used to perform attack responses.

This research will use an artificial immune system framework coupled with a Holland's classifier as a network intrusion detection. This research, then, will use a Q-learning approach from the reinforcement learning method to provide software a temporary repair mechanism. Finally, it will use a general game player composed of Monte-Carlo search tree equipped with a Local Regression and a Context Opponent modeling as a network defender.

CHAPTER II

RELATED WORKS

2.1 Detection

2.1.1 The Neural Network Approach

A group of researchers from IBM proposed a generic detection of computer boot sector viruses using a single layer neural network (Tesauro et al., 1996). This group of researchers extracted features, called trigrams, out of some 512 bytes long viral boot sectors. Each trigram was 3 bytes long. Out of the extracted features, the features that appeared in non-infected boot sectors were eliminated. The list of features were further shortened by imposing a restriction that made it certain that each of the boot sector viruses in the training set had to contain at least 4 of the features in the list. This resulted in a total of 50 features that were used to form the inputs for a neural network.

The presence(1) or the absence(0) of each one of the 50 features in a boot sector was recorded in a vector. The vector, consisting of 50 elements, became an input to a neural network which assigned a weight to each element of the vector independently. The neural network outputted a positive value for an infected boot sector and a negative value for a non-infected boot sector. This neural network was trained on 100 viral boot sectors and 50 non-viral boot sectors. It then was tested on the same amount of data and achieved around 80 to 85% accuracy on viral boot sectors and 100% accuracy on non viral boot sectors.

This neural network approach was employed with IBM anti-virus and appeared to have detected 75% of the new boot sector viruses since its deployment. Due to

the limitation on the number of training data, this approach suffered numerous false positives and false negatives. A false positive was the case where a legitimate non-viral boot sector was classified as a viral. A false negative was the other way around, meaning a viral was classified as a non-viral or was simply not detected. To improve the false positives and the false negatives ratio, an activation threshold of 0.7 was used in the neural network.

2.1.2 Probabilistic Approach

The probabilistic approach to computer security consisted of assigning a threat level probability to the activities that happened in a network or in a computer. A network intrusion could, for example, be considered similar to a two classes classification problem. A probability approach could be used to assign a probability to each incoming packet or activity. The probability value determined the class to which the activity belonged to. Some researchers introduced a probabilistic approach using Hidden Markov Model for host based protections. Du et al. (2004) applied Hidden Markov Model on the system calls associated to network connections. Wang et al. (2004) performed a similar approach by recording some of the series of system calls performed and by using them as a sequence of observations for a Hidden Markov Model.

The approaches proposed by Wang et al. (2004) and Du et al. (2004) started by building a normal set, λ, by using the Hidden Markov Model on training data. The number of the states for the model was problem specific. Once the training was terminated, a sequence of system calls was used as an observation. The trained Hidden Markov Model produced a probability that measured the degree to which the observation was in the normal set, λ. If the probability was below a defined threshold, the sequence of system calls was classified as an intrusion. The authors concluded that the number of states in the Hidden Markov Models was critical and the accuracy of the detection greatly depended on it. Further studies needed also to be carried out

to study the trade off between the level of computation and the accuracy of the detection. Wang et al. (2004) achieved a maximum of 28% abnormal activity detection on the *sendmail* system call data from the University of New Mexico.

2.1.3 The Genetic Algorithm Approach

Li (2004) proposed the employment of a genetic algorithm to generate rules that would be used to detect network intrusions. The proposed approach consisted of coding the source IP address, destination address, source port number, destination port number, duration, state, protocol, number of bytes sent by the originator and the number of bytes sent by the responder in a 57 genes chromosome. The chromosomes were first randomly generated according to the network's specification. Real network data captured by *Snort* (Koziol, 2003) or *tcpdump* were used to train the chromosomes.

The training data were hand classified by a network expert so that intrusions and non-intrusions data could be separated. The goal of the genetic algorithm was to produce rules, chromosomes, that would match intrusion attacks. The chromosomes were given fitness values that described their accuracy on matching an intrusion data. The following described the steps utilized to assign a fitness to each chromosome. First, the outcome value was calculated,

$$outcome = \sum_{i=1}^{57} Matched * Weight_i. \tag{2.1}$$

$Matched$ would have the value of 1 if a match occurred and 0 if a match did not occur. The weights were decreasing in the following order: destination IP, source IP, destination port number, duration, bytes sent by the originator, bytes sent by the responders, state, protocol and the source port number. This proposed approach introduced also two other variables that were used to calculate the fitness. The first variable, named suspicious_level, was derived from historical data and indicated

the extent to which two network connections were considered a match. The second variable, called ranking, indicated how difficult it was to detect the intrusion in the historical data. The fitness calculation proceeded as the following,

$$\Delta = |outcome - suspicious\_level| \tag{2.2}$$

$$penalty = \frac{\Delta * ranking}{100} \tag{2.3}$$

$$fitness = 1 - penalty \tag{2.4}$$

The author left it open to any interested researchers to decide how selection, crossover, mutation, and crowding could be performed to render the approach more efficient.

The approach proposed in Lu and Traore (2004) was similar to the above description. This approach expressed each real network security rule with a tree graph. The tree graph was abbreviated using the alphabets to create chromosomes. A regular genetic programming was used to produce new rules with the mutation probability set to 0.01 and the crossover probability set to 0.6. The authors suggested that more studies needed to be performed to discover the right mutation and crossover probabilities as well as the population size. Lu and Traore (2004) trained their genetic algorithm approach with 10,000 network connection records provided by DARPA. The average false negative rate for each rule was 5.04% and the average false positive rate was 5.23%. The average rate of detecting unknown attacks for each rule was 57.14%.

2.1.4 Artificial Immune System Approach

The LYSIS network security, in Hofmeyr and Forrest (2000), was among the first researches that used artificial immune system. LYSIS created a set of fixed length binary strings, called detectors, that underwent a negative selection, a tolerization and a human co-stimulation. The idea was to imitate a biological immune system. The detectors were trained to distinguish between the network's normal ac-

tivities, called self, and the abnormal activities, called non-self. During the negative selection and the tolerization period, any detectors that matched a normal behavior, self, of the network were eliminated and replaced by new detectors. The matching was done using the r-contiguous bits, meaning two strings matched each other if they had r contiguous bits in common. To lower the false positives' rate, the detectors that survived the tolerization period alerted a human operator when they matched an intrusion. The human operator either validated or discarded the intrusion alert. If an intrusion was validated, then the detector that triggered the alert received a costimulation signal and stayed active in the system. In the other case, the detector died and was replaced by a new one.

Approaches, similar to the one in Dal et al. (2008), used other detectors, called memory detectors, to performed the co-stimulation introduced previously. The approach in Dal et al. (2008) also utilized a genetic algorithm to multiply and to propagate highly fit detectors. The fitness of each detector was calculated during the negative selection process. Dasgupta and Gonzalez (2002) proposed a similar approach to Dal et al. (2008) with a modification that detectors were presented in real values instead of binary. Dasgupta and Gonzalez (2002) employed an euclidean distance instead of r-contiguous bits. The authors implemented two approaches: negative and positive characterizations. The negative characterization(NC) approach used negative selection. The positive characterization utilized a sample of self and eliminated non-self detectors. The authors concluded that the positive characterization was more accurate, however it required considerable training time and space. The negative characterization approach proposed by Dasgupta and Gonzalez (2002) used a dataset from DARPA and had a detection rate of 87.5% and a maximum false alarms rate of 1%.

2.1.5 The Classifier Approach

From an abstract perspective, intrusion detection systems were performing classification. Their goal was to classify an abnormal packet as a malicious packet and a normal packet as a non-malicious packet. The data to be classified could be of various forms. The following researches illustrated how a classification mechanism could be used in network intrusions detection.

Lee and Stolfo (1999) performed a data mining approach to classify malicious and non-malicious network packets. This approach used tcpdump as a training data. The source and the destination addresses, the ports numbers, the services, the message sizes, and the connection flags were extracted to create the features for the classification. Despite the requirement for a considerable amount of training data, the authors concluded that data mining classification could improve the performance of rule based intrusion detection systems. This data mining approach achieved a detection rate of 95% and a false positive rate of 6% when it was tested on the DARPA intrusion dataset. They claimed also that this approach could be easily added to any of the current exiting intrusion detection systems. Using a similar approach, other researchers, identical to Kim and Park (2003), utilized a support vector machine to perform the classification.

The classification process required that the training data's classes were known ahead of time. This situation created many overheads in using classifiers as an intrusions detection mechanism. To overcome this problem, a clustering of unlabelled data was used to perform intrusions detection (Portnoy et al., 2001). Similar to the previous approaches, this approach extracted the source address, the destination address and the ports numbers from TCP connections. The extracted values were transformed into numerical values and were normalized. The normalized values were then clustered without access to any class labels. Portnoy et al. (2001) assumed that the cluster that contained more than 98% of the training data was considered a normal

cluster and the remaining clusters were labelled as abnormal clusters. The detection was carried out by measuring the incoming packets' euclidean distances from the centroids of the clusters. If a distance was shorter than a defined threshold, then the packet belonged to the cluster from which the distance was taken from. The authors concluded that the accuracy and the rate of false positives for this approach depended considerably on the training data used. This approach was tested on the KDD CUP 99 and achieved 40% to 55% detection and produced 1.3% to 2.3% false positive alerts.

2.1.6 Self Organizing Map

Self Organizing Map(SOM) also known as Kohonen Map (Pachghare et al., 2009) is a single feed forward neural network commonly used in data compression and pattern recognition. Self Organizing Map is involved with transforming data from high dimensional space into a regular one or two dimensional array of neurons (Pachghare et al., 2009). The input vector for a SOM always has a higher dimension than the output vector. SOM works similar to a clustering. It uses training data to adjust the weight vectors of the neurons in the network. If a neuron has a weight vector that resembles the input vector, then the input vector becomes the new weight of the neuron. This neuron is called the wining neuron (Pachghare et al., 2009). The weight of other neurons in the network are adjusted based on their distance from the wining neurons.

Pachghare et al. (2009) used SOM as an approach to perform intrusion detection. Their approach consisted of collecting packets from a real network as training data. The collected packets were labelled according to the network event that corresponded to each packet. In addition, Pachghare et al. (2009) also used some packets that were already labelled from the DARPA web site as training data. Features were extracted from the packets to train the SOM. When the training was completed, real network packets were used to test the SOM classifiers.

Pachghare et al. (2009) indicated that the approach was successful on detecting intrusions. However, little was mentioned about how the features used in the experiments were extracted. The authors concluded that using SOM in intrusion detection would potentially provide higher quality intrusion detectors. This approach, as the authors also mentioned, was time consuming since training time had to be allocated to the SOM and the training data had to be labelled.

2.1.7 Plan Recognition

Plan recognition is an artificial intelligence area involved with predicting an agent's intention (Geib and Goldman, 2001). The actions performed by the agent are required to be observable in order to use plan prediction. In the case where the actions are not observable, the changes or the modifications produced by the actions need to be observable. Geib and Goldman (2001) proposed a theoretical approach of using plan recognition in intrusion detection. The approach originated from observing that intrusion detection systems were designed to report intrusions after they happened and not before they happened. By using a plan recognition, intrusion detection systems would be able to report intrusions before they occurred.

This theoretical approach required the use of an attack library. The sequence of events or actions associated with each attack in the library had to be available for the prediction to be accurate. This intrusion detection system used the intruders' partial sequence of actions to predict their intentions. The partial sequence of actions could infer multiple goals. To predict which one of the goals was likely to occur, probability values were assigned to each possible goal. The probability values were calculated by taking the distance between the partial sequence of actions and the possible goals' complete sequence of actions. The authors concluded that this approach could be beneficial for the network administrator since it narrowed the candidate intrusion attack responses to be employed. They mentioned, however, that this approach was not designed to handle a misleading attacker.

2.2 Automatic Software Repair

2.2.1 Repair Using System's Model

Once an intrusion was detected, the faulty software or the faulty application that induced the detectors to trigger the alarm had to be repaired or patched. Researchers had developed some intelligent patching systems that were capable of automatically patching a broken application. Classified to be among these patch software was the ClearView (Perkins et al., 2009) patch program. ClearView adopted a strategy that was in proximity to an artificial immune system to develop an automatic patching software. ClearView started by observing the normal behavior of the system. It used the register values and the memory contents to build a model of the system. It then created patches out of the model. When a failure or an abnormal activity was detected, ClearView generated a set of candidate patches. The candidate patches were evaluated to filter out the most effective ones. The patches, when executed, changed some of the registers and memory values. ClearView successfully generated 8 patches out of the 10 needed to block 10 vulnerabilities that are common with the Firefox web browser. Other researchers, similar to Kumar and Venkataram (1997), trained a neural network to perform anomaly detection and utilized case based reasoning to automatically deploy patches. These software were stand alone patching software. They were not embedded in the software that they maintained. Their mode of operation was similar to a network monitor.

2.2.2 Patch Using Source Code

Lin et al. (2007) analyzed the frequency of software patches from Microsoft. They noticed that software patches, from Microsoft, were, on average, generated 75 days after the security breaches were detected. Lin et al. (2007) used this situation to illustrate the need for automatic patch software. They introduced an automatic

patch software, named AutoPag, designed to patch damages caused by buffer overflow attacks. The AutoPag software consisted of 3 components: an out-of-bound detector, a root cause locator and a source patch generator. The out-bound detector monitored the use of the computer system's buffer. This detector was tasked with providing information to the root cause locator when a buffer overflow occurred. The root cause locator acted upon the buffer overflow information by analyzing the source code associated with the overflow. After analyzing the source code multiple times, the root cause locator produced a set containing the source code statements from which the overflow originated. The patch generator employed this set to instrument the code. The statements that triggered the overflow were changed automatically without human interventions.

The AutoPag software was designed specifically to handle the C programming language. It was not invented to patch software written in language other than C. It was only able to deliver automatic patch to buffer overflow attack. AutoPag required access to the source code of each software that it monitored. As the authors mentioned, an access to a source code could not always be possible since many software were only using their binary versions. AutoPag was tested with 18 overflow attacks from a Benchmark-Attack-Suite. It detected all of the 18 attacks and was able to localize the set of source code statements that expressed the vulnerability in each of the detected attack.

2.2.3 Patch Using Probing

Cui et al. (2007) introduced a new approach to automatically generate patches for firewalls and anti-viruses. This proposed approach, named ShieldGen, worked as follows. Incoming data, network or regular file data, were analyzed by an attack detector. The attack detector checked whether or not the data contained an arbitrary execution control, an arbitrary code execution or an arbitrary functions argument. If the incoming data contained any of the 3 exploits, listed previously, then the data

was passed to a data analyzer. The data analyzer inspected the data format to detect any formatting violations. When a format violation was detected, the data analyzer self generated a duplicate data, a probe, that amplified the violation detected. This probe was sent back to the attack detector. If the attack detector triggered an alarm while inspecting the probe, then predicate rules related to the violation in the probe were created and added to the firewall and the anti-virus system's rules.

The testing performed by the authors indicated that ShieldGen successfully generated patches for SQL vulnerabilities and Windows Remote Procedure Call vulnerabilities, but had problem generating patches for Windows Metafile vulnerabilities. This approach relied on the accuracy of the detector and the data analyzer. The detector needed to be equipped, ahead of time, with the information about the various forms of exploits. The normal format of the data to be analyzed had to be provided, in advance, to the data analyzer as well.

2.2.4 Binary Patch

Rather than automatically generating patches from source codes, Chen et al. (2010) introduced an approach, called PatchGen, that generated patches from a binary program and a sample input. This approach used Attribute-based Taint Analysis Module to find the relationship between the data and the variables in the binary program. This approach was mainly concerned about generating binary patch for buffer overflows. When the overflow was detected this approach inserted a modification to the binary program. The binary program would then jump or branch to the automatically generated guard code produced by PatchGen.

This approach was created for computers using Microsoft Windows operating systems. The approach was able to patch all the buffer overflows triggered by 6 randomly chosen attacks from the experiments. However, PatchGen had a control flow problem. The generated patches were only guaranteed to work if the binary code did not jump back to the part of the program that triggered overflows later during the

execution.

2.2.5 Auto Repair Using GA

Weimer et al. (2010) proposed an automatic program repair using a genetic algorithm. This approach appeared to be able to repair software during testing phases. The automatic repair worked as follows. The program to be tested was represented in a abstract syntax tree. Each node of the tree was either an executable statement or a control flow statement. Negative test cases and positive test cases were also needed with the abstract syntax tree. The negative test cases were the test data upon which the program failed to execute. Positive test cases were the test data upon which the program succeeded.

A set of program variants were created using a fault localization tool. A program variant consisted of an abstract syntax tree. Each statement in the abstract syntax tree was assigned a weight. The weight reflected the likelihood of the involvement of the statement with a negative test case. If a statement was visited during the execution of a negative test case but not visited during the execution of all positive test cases, then its weight was set to 1. If it was visited during the execution of a positive and a negative test case, then its weight was set to 0.1. In any other cases, the weight was set to 0.

The genetic algorithm maintained a population of program variants. The fitness of each variant reflected the number of negative and positive test cases that the variant passed. The variants were allowed to crossover and to mutate. The genetic algorithm replaced 50% of the population at each iteration. The genetic algorithm stopped when it was able to generate a program variant that passed all the negative and the positive test cases. This program variant was considered to be the repair solution. The statements in the abstract syntax tree of the solution were used to modify the statements in the original program.

Despite the fact that this approach only worked with the C programming lan-

guage, the authors expressed that they were able to repair defects associated with infinite loop, segmentation fault, heap buffer overrun, non-overflow denial of service, integer overflow, invalid exception, incorrect output and format string vulnerability. This approach assumed that the paths taken by the negative test cases were different from the paths taken by the positive test cases. It also assumed that the defects were repeatable and detectable. The approach greatly relied on the accuracy of the fault localization tool used. The authors concluded that only 60% of the repair trials produced a correct repair solution.

2.3 Autonomous Defender Approach

2.3.1 Risk Metric

In 2004, a company, called Symbiot Inc (Nathan and Hurley, 2004), proposed an artificial intelligence based network security defender. This proposed defender was expected to behave similar to a biological system. The system was also expected to be *autopoiesis*, meaning autonomous. It was expected to be capable of utilizing humans as observers from which the network defender can take lessons (Nathan and Hurley, 2004). The resultant defender was named iSIMS. This defender started by modeling the network environment. Then, it created a risk metric based on the sequence of events that happened in the network. iSIMS took actions based on the value of the risk metric. One of iSIMS's actions involved attacking the network attacker (Nathan and Erwin, 2004).

Little is known or published about the artificial intelligence methods and processes behind iSIMS. This situation is not surprising for commercial products. Symbiot Inc probably wants to keep the product's detailed descriptions confidential to protect itself from concurrences. Hopes are that in the future the researchers will know more about the designs and the processes behind iSIMS.

2.3.2 First Shooter Game

Table 2.1: Mapping visual metaphor to network metrics from Harrop and Armitage (2006)

Visual Metaphor	Network
Location	IP address, port number
Shape	Representation of object type(subnet, host or connection)
Size	Time aggregate of unique connection
Colour/Texture	Content type
Rotational	Velocity Throughput Oscillation about User defined alert a fixed point
Shoot (with gun)	IP address, port number
Heal (with syringe)	Undo (remove changes)
Fine tune (with pliers)	Bandwidth rate-shape

Harrop and Armitage (2006) introduced the theoretical transformation of a first shooter game to assist network administrators. Not all computer users can be a network administrator. The network administrator job requires training and advanced skills. To involve regular computer users with the network management tasks, Harrop and Armitage (2006) transformed a first shooter game into a visualization tool to maintain network security.

The approach proposed by Harrop and Armitage (2006) abstracted the network environment into a 3-dimensional game. The objects in the 3-dimensional visualization represented an abstraction of real network objects. The network administrator logged in as a player. The actions taken by the player, network administrator, were translated into real network commands. The correspondence between the 3-dimensional game and the network was similar to the information expressed in Table A.1

This approach was implemented using "Cube 1" game engine, a router and a FreeBSD host. The authors did not provide more information about the accuracy and the correctness of the visualization. The network operations that could be performed with this approach were limited to the number of translations possible between the game environment and the real network. More actions needed to be added to the game to improve the usability of this approach in administering a network of computers.

2.3.3 Autonomous Security Infrastructure

Corsava and Getov (2003) proposed the idea of making computer security infrastructures intelligent. This proposed approach consisted of one single software that was capable of detecting intrusions, detecting failures, repairing security breaches and responding to attacks. This security software was intended to be employed in network infrastructures based on Unix and Linux operation systems.

The proposed security software was named intelliagent. Its mode of operation could be described as follows. A script was installed in the protected host. The script was tasked with awakening intelliagent every x minutes, $x = 5$ was an example. When awakened, intelliagent would check for intrusions and anomalies. If a repair was needed, this approach restored the host by using the most recent backup data. If an attack happened, then intelliagent scanned its attack library to localize the appropriate action to be executed. This approach was also equipped with the ability to inform a system administrator whenever an attack happened or if it was unsure about the event that happened in the host. Before going back to sleep, intelliagent gathered information about the processes and the system's state to create the most recent backup data.

This approach was employed with a mobile phone company that owned about 650 servers. Prior to its deployment, the authors of this approach had to analyze the security attacks encountered by the company. The authors mentioned that it took a year before they were able to configure intelliagent to be able to respond to the attacks found in the analysis. The authors explained that the mobile cell phone company's security breaches was reduced from more than 1000 to 0 after the company employed intelliagent for one year. This approach was a "do it all" software that required pre-built in detection and attack response rules. Intelliagent could be distributed to any node that required protection. It appeared to be customizable. Intelliagent's performance greatly depended on the pre-built in rules that it had.

2.3.4 Self-Reproducing Machine Learning

Huang and Chen (2007) expressed the possibility of using self-reproducing machine learning in internet and computer security. Self-reproducing machine learning was defined as follows. Let A be a factory system and E be a source system. Let P be the system's characteristic. (A,E) was said to self-reproduce (B,F) if B had the characteristic P and F was a subsystem of E (Huang and Chen, 2007). The following was an example of this case. A was a learning system and B was the internet environment.

Huang and Chen (2007) explained that this idea can be used in internet and computer security. They proposed an approach similar to the following. Given a learning system A and a database of viruses E, new variants of E could be self-reproduced to improve the database of viruses. They also expressed the possibility of using this self-reproducing machine learning in firewall systems. Given a learning system and a set of rules, a new set of rules could also be self-generated to improve a firewall's rules.

Huang and Chen (2007) were proposing a purely theoretical approach. Their paper about this self-reproducing machine learning did not contain any implementations. Readers were left to wonder about the learning system that could possibly be used to perform the self-reproduction. The authors mentioned that game theory and machine learning methods could possibly be used in the self-reproduction process.

2.3.5 Machine Learning in Reactive Security System

Rubinstein (2010), in his dissertation, proposed the approach of using machine learning to create a reactive security system. This proposed approach emphasized fighting unknown attackers in term of costs rather than in term of actions. Rubinstein (2010) abstracted the network or the computer system environment as a game in which an attacker and a defender were allowed to take turns to execute actions.

The executed actions moved the environment from one node of the graph representing the game into a new node. The attacker was given a reward when it was able to compromise a host in the network. However, the attacker had to pay the cost associated with compromising the host.

The defender's goal was to make any attackers trying to compromise any host in the network pay the maximum amount of cost while keeping their rewards at a minimum amount. The defender's actions, therefore, consisted of choosing the security mechanism or the security software to be employed at each host in the network. Each security mechanism had a cost related to it. The defender was given a limited budget and had to plan according to the cost of each security mechanism.

Rubinstein (2010) allowed the attacker to have knowledge about the defender's defense strategy. The defender had no prior information about the attacker's intention. The defender, however, was allowed to observe the sequence of actions taken by the attacker. Rubinstein (2010) proved theoretically that this reactive approach had an advantage over a proactive approach. The proactive approach was a defense strategy that had fixed and unchangeable security mechanisms. The proactive defense strategy was hard to design due to the fact that the attackers' intentions were not known ahead of time. In the proactive strategy, if the designed defense mechanism failed to consider a certain number of attacks, then designing a new proactive defense mechanism could cost more than using a reactive defense mechanism.

The reactive defense mechanism proposed by Rubinstein (2010) worked well under the condition that the system was able to survive numerous attacks without suffering catastrophic losses. It also appeared to excel if the defender was not penalized when it was moving assets.

CHAPTER III

DETECTION: ARTIFICIAL IMMUNE SYSTEM BASED ON HOLLAND'S CLASSIFIER

3.1 Introduction

The goal of this chapter is to create a self learning intrusion detection system (IDS). This new IDS is based on the combination of two AI approaches, Artificial Immune System and Holland's classifiers. The IDS, proposed in this chapter, is also designed to learn to update its own intrusion detection rules. Researchers started employing AI to detect computer system intrusion in 1990 (Hofmeyr and Forrest, 2000). The goal was to develop a system that independently would learn to detect intrusion in a computer system. An intrusion is defined as, "any use, or attempted use, of a system that exceeds authentication limits" (Solomon and Chapple, 2005). An intruder can be external or internal. There exists two basic types of intrusion: misuse intrusion and anomaly intrusion.

A misuse intrusion is defined as an attack to known system vulnerability. This type of intrusion is easy to detect due to the fact that the actions required to exploit a system's vulnerabilities are collectively known under the term attack signatures (Solomon and Chapple, 2005). The anomaly intrusion is harder to detect. The actions involved with this type of intrusion are not collectively known. This intrusion induces the system to perform abnormal activities.

Most of the artificial intelligence researches developed, since the 90s, to handle both types of intrusion attempted to imitate a biological immune system mechanism. A biological immune system uses *lymphocyte* cells that can detect, attack and block

foreign cells, *pathogens*, that intend to cause harm.

With a close similarity, researchers created detectors similar to lymphocytes. The detectors are deployed in the computer system's nodes to intercept and to report any malicious activities. The field that studies the process of imitating a biological immune system to create intrusion detection systems is referred to as Artificial Immune System(AIS)(Harmer et al., 2002).

In AIS, the detectors go through the same process as the lymphocytes. They start by learning to distinguish between the cells that are internal, *self*, judged not harmful to the system and the cells that are external, *non-self*, which are harmful to the system. Similar as in a real biological immune system, information about previous intrusions are recorded by the AIS to render the system robust to similar intrusions.

3.2 Why Mimicking the Immune System?

The immune system presents numerous qualities that are desirable in providing computer security in a network (Hofmeyr and Forrest, 2000). The immune system can parallelize operations. It is distributed, adaptive, dynamic and tolerant. By imitating the immune system, researchers hoped to capture the qualities previously listed.

In AIS, the detectors are distributed to each node in the network or in the computer system. Each node has its independent detectors and is able to perform independent detection, therefore allowing parallel detection. The frequently used detectors are saved in memory and are propagated among the nodes. The least used detectors are replaced by new detectors. These processes are needed to guarantee adaptability and dynamism. AIS is tolerant due to the fact that a misclassification from one node can be corrected in another node. In addition, AIS does not present a single point of failure as most of the currently used intrusion detection systems do.

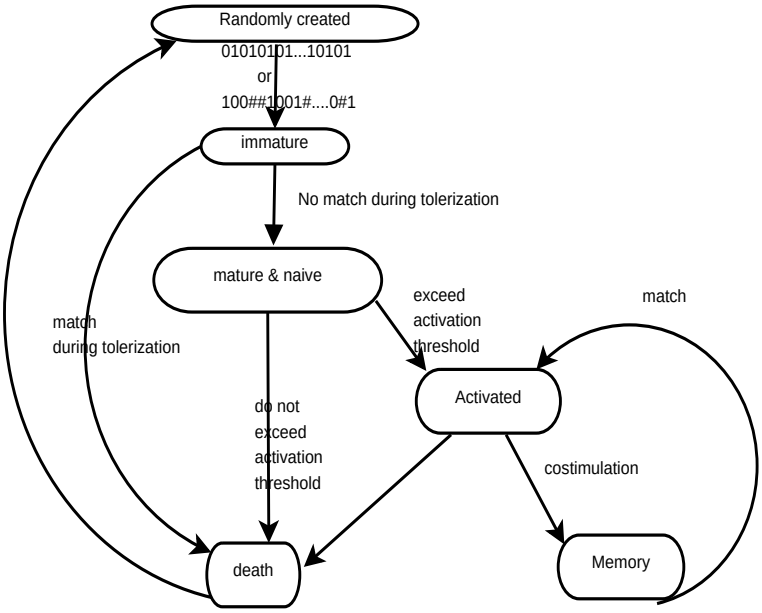


Figure 3.1: Detector life cycle from Hofmeyr and Forrest (2000)

3.3 Artificial Immune System

Since 1990, there were numerous variations of Artificial Immune Systems built. All of these AIS share the same architecture that will be explained in this section.

3.3.1 Detectors

The AIS detectors, S, are the equivalent of the lymphocytes in a biological immune system. A detector is encoded as a string, s, of length l. In the computer network situation, s can possibly consist of a binary string that contains the IP addresses of the origin and the destination host, and the port numbers where the communication is happening (Hofmeyr and Forrest, 1999). In an application security, s can reflect the identification of a pair of software components and their respective actions.

The length, l, of each detector depends on the problem to be solved. Detectors are generated randomly. The detectors are subjected to a negative selection and a tolerization which will be explained below.

3.3.2 Negative Selection

The negative selection process consists of eliminating the detectors that reflect the system's normal mode of operation (Kim et al., 2007). The negative selection process relies on assuming that fewer detectors are needed to represent intrusions. The negative selection can be achieved by two options. The first option consists of deploying the detectors to the system's nodes and allowing them to observe the normal behavior of the system. A detector that matches a normal activity is deleted and can possibly be replaced by a new one. In this first option, there is a risk of real intrusion happening and legitimate detectors can possibly be deleted. The second option consists of simulating the system's normal behavior and eliminating the detectors that match the system's normal activities. Depending on the size of the computer system, simulation may not always be possible.

Clarification needs to be added about the criteria used to decide a match between a detector and a system activity. The most common criteria is the *r-contiguous bits*. Similar to a detector, a system activity is represented by a string of length, l. A detector, d, and a system activity, e, match each other if they have r contiguous bits in common (Hofmeyr and Forrest, 2000), i.e. $match(d, e) \geq r$. Other matching criteria, such as a hamming distance, exists. More information about matching criteria is available in Harmer et al. (2002).

3.3.3 Tolerization

The period during which the detectors undergo negative selection is called tolerization. During the tolerization period, detectors are trained to bind and to detect irregular system activities also known as non-self activities (Hofmeyr and Forrest, 1999). The length of the tolerization period depends on the specifications of the domain and the system. For simplicity, let t describe the length of the tolerization period.

The tolerization and negative selection period provide the advantage of training detectors without having knowledges of the possible intrusion attacks. This fact gives AIS the possibility to be general and adaptive. Depending on the system's size and the value of t, there are possibilities that not all of the system's normal activities will be met during the tolerizarion period. This results in detectors that are capable of triggering false positives. In addition, the tolerization period does not guarantee a hole free system. Holes are irregular system activities that are not detected by the current set of detectors (Hofmeyr and Forrest, 2000).

3.3.4 Memorization

The detectors that survived the tolerization period are called *mature detectors* (Dal et al., 2008). Different values of contiguous bits, r, can possibly be assigned to the mature detectors. Each mature detector owns a detection counter that counts how many times the particular detector detected intrusions. A mature detector that owns a detection counter greater than a threshold, activation threshold, is called activated. A mature detector that fails to reach the activation threshold within a defined time period dies and is replaced by a new and immature detector (Harmer et al., 2002). The activated detectors can be subjected to a lower activation threshold. This guarantees that they have a longer life span. An activated detector can become a memory detector via a co-stimulation process. The co-stimulation process will be explained below. A memory detector has an indefinite life span and can have an activation threshold of 1 (Hofmeyr and Forrest, 2000).

3.3.5 Co-Stimulation

The co-stimulation process is also known as *Second Responses*. The main idea is to assure that the intrusions detected by the activated detectors are not false positives. In this case, once an active detector notifies an intrusion a second agent needs to confirm or deny the intrusion. The second agent can be a sister detector from

the same node or a sister detector from a different node. Dal et al. (2008) use the memory detectors to confirm the intrusions detected by an active detector. Hofmeyr and Forrest (2000) and Harmer et al. (2002) use a human operator to validate the intrusion.

In the case where an active detector reports a false positive, no co-stimulation message is given to the detector. Such detector dies after waiting for a co-stimulation message for a defined time period. An active detector that received a co-stimulation message survives and has the opportunity to become a memory detector. The number of memory detectors can be limited for each node in the system. The least used memory detector is the most probable candidate to be replaced when the number of memory detectors reaches the specified limit. The detector's life cycle is summarized by Figure 3.1.

3.3.6 Additional Learning Phase

An additional learning phase can be added before the detectors become mature. Dal et al. (2008), for example, added a phase where the detectors that survived negative selection were again trained with non-self, irregular activities, before becoming mature detectors.

Harmer et al. (2002) and Hofmeyr and Forrest (2000) proposed the idea of using a genetic algorithm structure to assign fitnesses to the detectors that successfully matched an intrusion. Detectors with high fitnesses were cloned, then promoted as memory detectors and finally propagated in the network. Dal et al. (2008) started from this idea and allowed the highly fit detectors to be cloned and mutated. The resultant detectors, after cloning and mutation, were allowed to compete to become memory detectors. Other approaches, like the one in (Li, 2004), used a full genetic algorithm approach to bring detectors to maturity.

3.3.7 Proximity to Holland's Classifier System

"A classifier system is a machine learning systems that learns syntactically simple string rules (called classifiers)" (Goldberg, 1989). The classifier system was first described by Holland and Reitman (1977) . A classifier or a rule is composed of two components: a condition and an action. If the condition is satisfied, then the action is carried out. A classifier system consists, first, of generating rules that are supposed to represent the behavior of a system or an environment. Then, the rules are tested on either real or simulated data to exclude rules that do not reflect the system's behavior. During this refinement period, also known as apportionment of credit system Holland (1985), each rule is given a strength or a fitness value. Finally, by using a full genetic algorithm process, the most fit rules are allowed to duplicate and multiply.

The rule encoding in Holland's classifier has a small difference from the AIS's detector encoding. Each rule is encoded using three characters, 0, 1, and #, rather than being encoded by only using 0 and 1. The '#' character is called the "don't care" symbol. To check whether a rule should be activated or not, the message from the environment or the system is matched to the condition part of the rule. If there is a complete or a partial match, then the rule competes for an activation via a bidding process. A complete match occurs if at every position a 0 in the message matches 0 or # in the condition of the rule, a 1 matches 1 or # in the condition of the rule (Goldberg, 1989).

The bidding system works as follows. Each rule that has its condition part matched by the system's message makes a bid. The bid is a numerical value that is a portion of the strength of the rule. Each rule is given an initial strength. The rule offering the highest bid is activated and is allowed to execute its action. Any bids made by a rule are subtracted from its strength. The rule that won the previous bid receives the current highest bid as an addition to its strength.

Despite pointing out some similarities, researchers, such as Hofmeyr and Forrest (2000) and Harmer et al. (2002), advocate that the field of classifier system and AIS are independent. The r-contiguous bit is close to the matching process using the "don't care" symbol. There is also a similarity on using a genetic algorithm to generate new detectors or new rules.

3.4 Using Holland's Classifier System in AIS

Many researchers, such as Sun et al. (2010) and Yuan et al. (2009), have abandoned the traditional AIS and moved to a new field called *"Danger Theory(DT)."* Danger Theory believes that the biological immune system is not concerned with distinguishing self from non-self. The biological immune system, according to DT, is concerned with distinguishing between the pathogens or the bacteria that cause danger and the ones that do not.

While the danger theory approach is more general and close to the real biological immune system, this dissertation believes that AIS remains adequate for intrusions and misuses detections. The reasons that pushed researchers to have more interest in DT are related to the fact that AIS has a tendency to trigger numerous false positives alerts. Also, AIS is also perceived to have a slow adaptivity.

To overcome these two problems, this dissertation is implementing an AIS based on Holland's classifier system. Classifier systems can detect intrusions up to 95% (Lee and Stolfo, 1999). The best AIS system can reach 87.5% of accuracy in detecting intrusions (Dasgupta and Gonzalez, 2002). The combination of these two approaches is expected to bring an accurate and adaptive IDS system that is capable of continuously learning by itself to develop detection rules. Since there exist numerous similarities between Holland's classifiers and AIS, this dissertation has chosen to exploit these similarities to create a new intrusion detection system.

For simplicity, the new intrusion detection system, proposed in this disserta-

tion, will be called, AIS-CS. The CS part stands for classifier system. The Holland's classifier system encoding is expected to add more generality to AIS. The classifier system processes are also expected to bring more adaptivity to AIS. The modifications that will be operated on the traditional AIS are listed below.

- Detector: The detectors are generated randomly with the three characters 0, 1, and # in the initial stage. The detectors are given initial strengths. The detectors are given a status: immature, mature, activated and memory. They are also time stamped. The time stamp is used to control the replacement or dying policy. When the AIS-CS system has passed the initial phase, generating a new detector is done by using a genetic algorithm.

- Negative Selection: Similar to the traditional AIS, the negative selection consists of eliminating detectors that describe the system's normal behaviors.

- Memorization: Mature detectors that reach the activation threshold become active detectors. The frequently used mature detectors are broadcasted to the nodes in the system. The frequently employed active detector becomes a memory detector automatically.

- Co-stimulation: An active detector that did not trigger a false positive becomes a memory detector. A human operator or a rule based system needs to validate the alarm sent by the active detectors in the initial phase. When more memory detectors are available, they are used to validate the active detectors' alerts.

- Additional Learning phase: The mature detectors are in competition with each other by using a genetic algorithm. Mature detectors that match an incoming message bid for an activation. The bidding system controls the fitness, strength, of each mature detector. Non-bidding detectors are charged a life tax to guarantee that their strengths decrease with time. Let $F_d(t)$ be the strength of a

detector, d, at time t. Let C_{tax} be a life tax and C_{bid} be a bid tax. Let $R_d(t)$ represents the reward receive by d at time t. The fitness of each detector can be calculated as follow,

$$F_d(t+1) = F_d(t) - C_{tax}F_d(t) - C_{bid}F_d(t) + R_d(t). \tag{3.1}$$

Note that $R(t) = 0$ if $\arg\max_i C_{bid}F_i(t) \neq d$. If $\arg\max_i C_{bid}F_i(t) = d$, then $R_d(t)$ is equal to the highest bid at time t. A new detector is only generated when one of the current detectors dies. The highly fit, mature detector, is expected to have a high probability to multiply. The less fit, mature detector, dies. Also, the mature detector that does not reach the activation threshold within a defined time period becomes a probable candidate to be replaced during the genetic algorithm process. When a new immature detector is to be generated, the mature detector that is far away from reaching the activation threshold is replaced.

3.5 Experiment

This section is designed to express the influences of the parameters involved with AIS-CS. In this section, graphs that express the performances of AIS-CS will be seen as the parameters such as activation threshold, match points, number of nodes, tolerization period, crossover probability and mutation probability change. An extended version of the 1999 DARPA intrusion detection dataset is used in the experiment.

3.5.1 Setting

A network of computer systems was simulated with CSIM (Schwetman, 2009). The network consisted of some regular nodes, computers that belong to the network, and some irregular nodes. The regular nodes were allowed to communicate to each

other by sending TCP/IP packets. Each regular node managed its own AIS-CS system.

The communication between any two nodes in the network is represented by a 30 bits binary string. The first 14 bits indicated the compressed version of the source's IP address and the port number used. The second 14 bits indicated the destination's IP address and the port number. The last two bits indicated the type of packet sent, which can be a SYN or an ACK or a DATA packet. The extension to the DARPA data set consisted of multiplying the number of attacks and the number of hosts in the network.

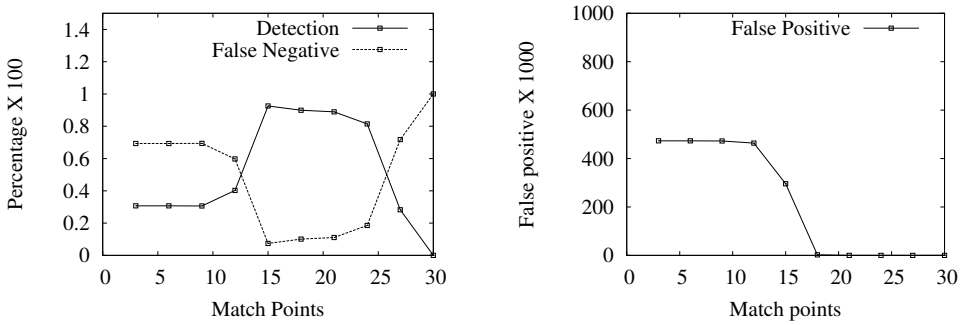
3.5.2 Measurement Criteria

In each experiment, AIS-CS was judged on three criteria. The first criteria is the percentage of intrusions that it was able to detect. For each experiment, the number of attacks was fixed and a percentage was calculated to see how many of the attacks were detected. The second criteria is the percentage of false negatives. False negatives describe the case where some of the attacks are not detected. The last criteria is the number of false positives. False positives reflect the case where some of the legitimate connections are classified as an attack.

3.5.3 Result

Match Points

This experiment studied the performance of AIS-CS under different match points values. Match points is the parameter that controls how many bits a communication representation and a detector need to have in common before an intrusion alarm is triggered. For this experiment, there were 50 regular nodes and 50 irregular nodes or attackers. There was an imposed tolerization period of 15 seconds and a simulation period of 15 seconds. The activation threshold was set to 4. Recall that



(a) Detection and False negative (b) False positive

Figure 3.2: Performance under different match points values. The experiment was performed with 50 regular and 50 irregular nodes, 15 seconds of simulation time, an activation value of 4 and 15 seconds of tolerization time. Probability of mutation is 0.001 and crossover probability is 0.6.

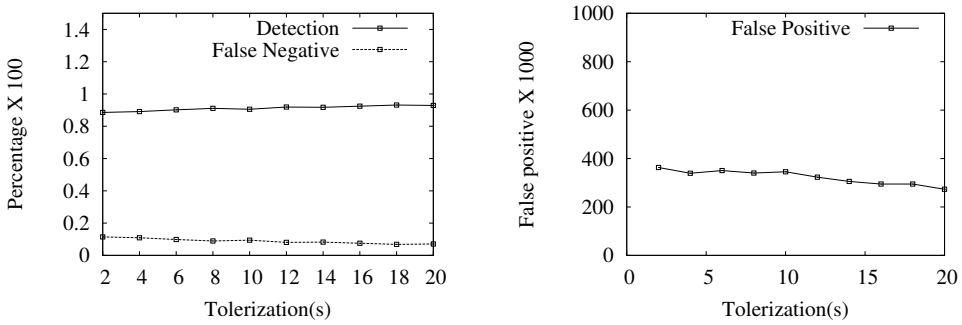
during the tolerization period, only the regular nodes are communicating. During the simulation period, all the nodes are allowed to communicate. There was a total of 24847 attacks which were launched in two waves. The data represented in Figure 3.2 are the average of a 1000 runs.

Tolerization

Tolerization is the period within which negative selections are performed. During this period, it is assumed that only regular nodes are communicating. This experiment studied the effect of the length of the tolerization period on the performance of the AIS-CS. For each tolerization period involved in this experiment, 15 seconds of simulation period was performed. For this experiment, the match point was set to 15 and the activation threshold was set to 4. There were 50 regular and 50 irregular nodes. There was a total of 24847 attacks which were launched in two separate groups. The data represented in Figure 3.3 are the average of a 1000 runs.

Number of Nodes

This experiment studied the effect of the number of regular and irregular nodes in the network. In total, there were 100 nodes. Some of the nodes were set to be



(a) Detection and False negative (b) False positive

Figure 3.3: Performance under different tolerization periods. The experiment was performed with 50 regular and 50 irregular nodes, 15 match points and an activation value of 4. Probability of mutation is 0.001 and crossover probability is 0.6.

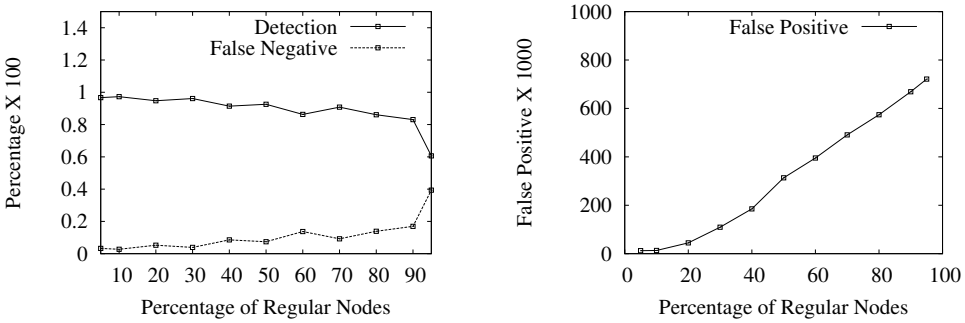
regulars and the remaining were considered attackers. In this experiment, a match point of 15, a simulation period of 15 seconds, an activation parameter of 4 and a tolerization period of 15 seconds were used. There was a total of 24847 attacks which were launched in two groups. The data represented in Figure 3.4 are the average of a 1000 runs.

Activation Threshold

The activation threshold parameter controls the number of detections that a mature detector needs to perform to become an active detector or a memory detector. Mature detectors that did not meet the activation threshold within 100 ms were replaced by immature detectors. The effects of the values of the activation parameter to the performance of AIS-CS are recorded in Figure 3.5. The match point parameter was set to 15 in this experiment. There were 50 regular and 50 irregular nodes. A simulation and a tolerization period of 15 seconds each were used. There was a total of 24847 attacks which were launched in two waves.

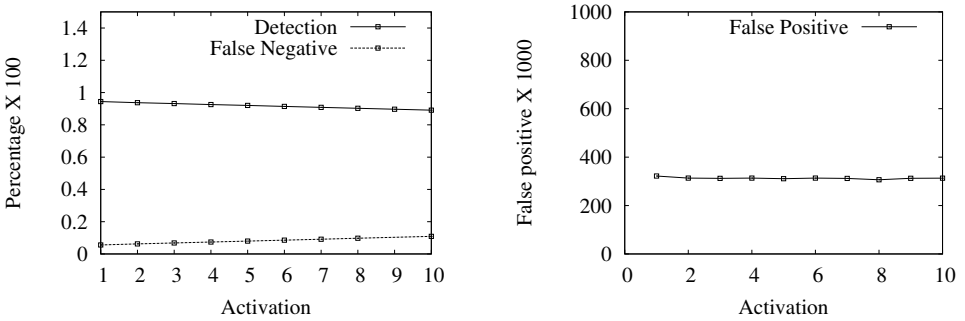
Crossover

The crossover probability controls how likely two detectors chosen by a roulette wheel selection can exchange bits to form a new detector. The new detector produced



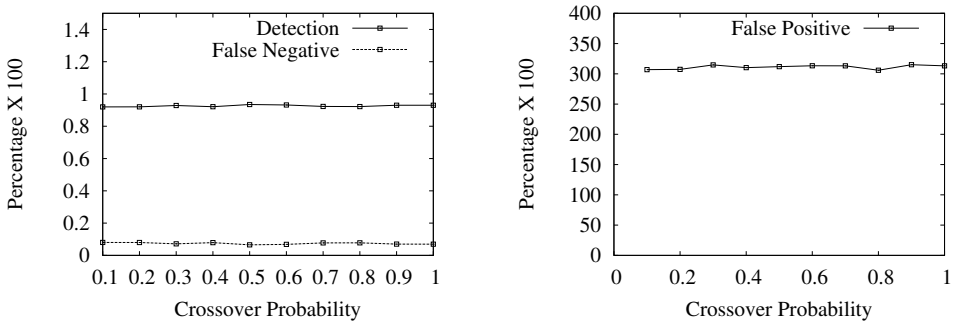
(a) Detection and False negative (b) False positive

Figure 3.4: Performance under different number of regular and irregular nodes. The experiment was performed with 15 match points, an activation value of 4 and a tolerization and a simulation of 15 seconds each. Probability of mutation is 0.001 and crossover probability is 0.6.



(a) Detection and False negative (b) False positive

Figure 3.5: Performance under different activation values. The experiment was performed with 15 match points and a tolerization and a simulation of 15 seconds each. 50 regular and 50 irregular nodes were used. Probability of mutation is 0.001 and crossover probability is 0.6.



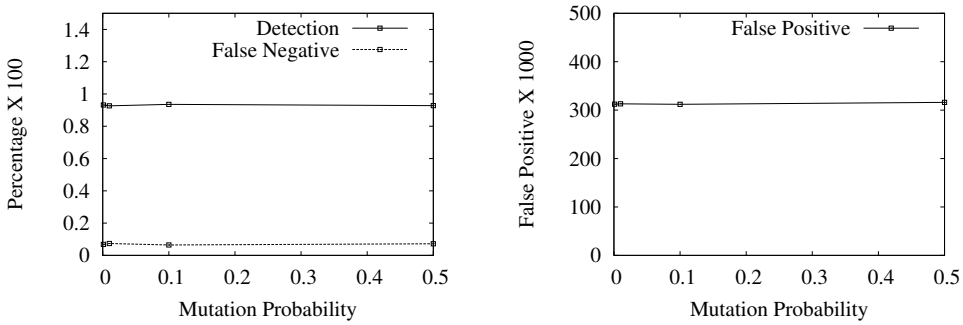
(a) Detection and False negative (b) False positive

Figure 3.6: Performance under crossover probability. The experiment was performed with 15 match points and a tolerization, a simulation of 15 seconds each and activation of 3. 50 regular and 50 irregular nodes were used. Probability of mutation is 0.001.

is tagged as an immature detector and is expected to undergo negative selection. The experiment in this section was conducted to study the effect of the crossover probability on the performance of AIS-CS. 15 match points were used. There were 50 regular and 50 irregular nodes. The tolerization and the simulation time were 15 seconds each. The activation parameter was set to 3. There was a total of 24847 attacks which were launched in two groups.

Mutation

Mutation probability controls how likely a bit in a a newly formed detector can change. The mutation rule is defined as follows. A '#' can become a '0' or a '1', a '0' can become a '1' or a '#' and a '1' can become a '0' or a '#'. This experiment describes how mutation can affect the performance of AIS-CS. 15 match points were used. There were 50 regular and 50 irregular nodes. The tolerization and the simulation time were 15 seconds each. The activation parameters was set to 3. There was a total of 24847 attacks which were launched in two separate sets.



(a) Detection and False negative (b) False positive

Figure 3.7: Performance under mutation probability. The experiment was performed with 15 match points and a tolerization, a simulation of 15 seconds each and activation of 3. 50 regular and 50 irregular nodes were used. Probability of crossover is 0.6.

3.6 Key Findings

Having a match points at 15 appears to provide a better detection and a better false negative percentage. This value, however, is not providing the lowest false positives alert. This is confirmed by Figure 3.2. Having match points between the interval $[16, 22]$ appears to be more appropriate, with a possibility of a small amount of compromise in the detection and the false negatives percentage, for the network in the experiment. A match points of 15 is about half the length of the detectors and is equivalent to the source's address and port. This situation favors the detection of packets originating from unknown or not previously seen hosts. However, it is possible that not every packets from an unknown or not previously seen hosts are intrusion packets. This last situation is handled by increasing the number of match points to lower the false positive alters.

Providing more tolerization period appears to increase the detection percentage and lower the false negatives percentage. As it can be seen from Figure 3.3, a greater amount of tolerization period lowers the number of false positives as well. The match point parameter plays an important role in this process. It can be seen from Figure 3.2 that a tolerization of 15 seconds with 20 match points triggered no

false positives. In Figure 3.3, however, the same experiment with match points of 15 provided numerous false positives.

As the number of nodes to be protected increases, the number of false positives and the percentage of false negatives increase. The detection percentage is decreasing as the number of regular nodes increases. Protecting a network that is composed of numerous nodes is not an easy task. The result in Figure 3.4 clearly expresses this fact. This situation, however, can possibly be leveraged by lengthening the tolerization period.

A low activation threshold provides a high detection percentage, a low false negatives percentage and a high false positive rate. A high activation threshold provides a slightly lower detection percentage, a slightly higher false negatives percentage and a slightly lower false positives rate. The mutation and crossover probability appear to have a very small effect on the false positives rate. This effect is related to the fact that these two probabilities are directly involved with the generation of new immature detectors. Most of the new immature detectors are expected to trigger false positive alerts. The ideal setting for these two parameters is to set the mutation probability very low, close to 0, and to set the crossover probability slightly above 0.5.

Depending on the computer system being used, the network administrator has to find the set of parameters that are acceptable to the desired detection expected. The acceptable parameters may impose some trade off between the three measurements criteria presented here. In our attempt to find the suitable parameters for the network used in the experiments, we have discovered that using different parameters values for the tolerization and the simulation appears to improve the performance of AIS-CS. Under a setting with an activation value of 4, a mutation probability of 0.001 and a crossover probability of 0.6, we set the tolerization's match points at 20 and the simulation's match points at 15. The result from the average of 1000 runs on a non-

altered and simulated 1999 DARPA dataset (2 weeks of initial tolerization, 3 weeks of testing with 244 attacks on 7 hosts) shows a detection percentage of $92.5750\% \pm 1\%$ and an average false positives number of 17.21 ± 10 per week.

Table 3.1: Table Comparing AIS-CS, AIS and Classifier System

	AIS-CS	Classifier Systems	AIS
Detection	92.5750%	95.3%	87.5%
two-tailed p-value		<0.0001	<0.0001

CHAPTER IV

TEMPORARY REPAIR: THE Q-LEARNING APPROACH

4.1 Introduction

The previous chapter has illustrated how artificial intelligent can be used in an intrusion detection system. The focus was on detection and little was said about the responses or the actions that need to be taken to answer the detected intrusion. In this chapter, an artificial intelligence approach is proposed to temporary repair misuses and anomalies intrusions that can happen in a software. Possible extension of this approach to a network system will also be discussed.

After an intrusion or a misuse or an anomaly has been detected, the application or the software or the operating system related to the detected anomaly has to be repaired by a process known as patching. The patching process is aimed to prevent the detected misuse from happening once more in the future. In most of the cases, either software vendors or creators provide the patches needed via software security updates. It can take days before software vendors can provide adequate patches to a detected anomaly. A software that can fully or temporarily patch itself is a big advantage for companies or system users that require a high level of system availability and usability.

Installing security features in software is a requirement for many companies. Software development has evolved from a centralized production into a distributed production. It is common in today's software development to see the different parts of a software being developed by different companies in different geographical location. This situation raises a concern in security since it is not always certain that

the different parts will always be secure as they are expected to be. To limit the number of security updates and patches to be performed, security can be embedded in the software itself, starting in the software design. To enforce security, in software development, researchers have developed security features in software architecture language such as ADL, XADL and many more. The goal is to force the development teams to include verifiable security features in the software's code when the real implementation is performed.

This chapter gives an alternative approach into software security and security repair by using an Artificial Intelligent technique called Q-learning. The Q-learning approach does not impose restriction on the language to be used. The goal of this approach is to equip a software or an operating system with the ability to learn by itself to develop a temporary repair mechanism for its own protection. This repair mechanism has to automatically evolve over time and be adaptive to new security attacks. The repair mechanism proposed in this approach is not a stand alone software that is only focused on finding and providing patches. This proposed security repair mechanism is embedded as a part of the software itself. Note also that this proposed repair mechanism neither modifies the software's source codes nor changes the binary or executable files, that is the reason why it is called temporary repair.

4.2 Q-learning

In Q-learning, the learning or the environment is represented by a tree-like graph. The nodes of the tree represent the states in the environment. Each branch from each node of the tree describes an action that can be taken from the state represented by the node. The tree has starting states and final or goal states. There is a reward, r, associated with taking an action, a, from a state, s. The total discounted reward, $r(t)$, received by the learner starting at time t is:

$$r(t) = r(t) + \gamma r(t+1) + \gamma^2 r(t+2) + \ldots + \gamma^n r(t+n) + \ldots \tag{4.1}$$

γ is a discount factor, $\gamma\epsilon[0,1]$. Q-values are created to describe the quality of taking an action, a, while in a state, s. This value is represented by $Q(s,a)$. If the environment is in a state, s, and the learner took an action, a, then the environment transitions into a new state, s', where the learner can take a new action, a'.

The goal of the learning is to visit all the possible pairs of state-action, in the tree, multiple times in order to build a policy. A policy, π, is a mapping from states to actions that will maximize or minimize the long term total discounted reward of the learner. The Q-values received by following a policy, π, is denoted by $Q^{\pi}(s,a)$,

$$Q^{\pi}(s,a) = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+1} | s_t = s, a_t = a\} \tag{4.2}$$

$$\pi(s) = \arg\max_{a} Q^{\pi}(s,a). \tag{4.3}$$

In Q-learning, the learner has to finish an episode, a learning instance going from a starting state into a final state, before the Q-values of the states encountered in the episode can be updated in accordance to the total reward received. This process is referred to as backup technique.

The learning phase requires the learner to visit all possible state-action pairs multiple times before an optimal policy can be produced. More explanations about this approach can be found in Sutton and Barto (1998). The theoretical convergence proof Q-learning is available in Watkins and Dayan (1992). Algorithm 1 expresses the tabular version of Q-learning.

4.3 Software Composition

A software consists of a configuration of software components and software connectors. The components and the connectors interact with each other while performing the overall specific task to be performed by the software. The following definitions of software component and connectors were taken directly from (Taylor

Algorithm 1 Tabular Q-Learning

```
for each episode do
    s is set to the current state
    while s is not terminal do
        Take action a, observe r, s'
        Choose a' from s' using policy derived from Q
        Q(s,a) = (1 − α)Q(s,a) + α(r + γQ(s',a'))
        s ← s'; a ← a'
    end while
end for
```

et al., 2009).

- A software component is an architectural entity that encapsulates a subset of the system's functionality and/or data, restricts access to that subset via an explicitly defined interface, and has explicitly defined dependencies on its required execution context.

- A software connector is an architectural element tasked with effecting and regulating interactions among components.

- An architectural configuration is a set of specific associations between the components and connectors of a software system's architecture.

4.4 Related Works on Software Security

Taylor et al. (2009) list few languages such as Secure XADL and XACML that are capable of imposing security concern in software architectures. These languages' objectives are to impose the way connectors and components interact with each other. These languages provide a way where the software architect can specify the permission, privilege and the role for each component and connector in the architecture. Some tools and verification strategies exist to check whether the implemented software match the software architecture. These tools can help to guarantee that the security restrictions are not violated. Researches similar to the one in Ren and Taylor (2005) built upon the previous idea and implemented access control list

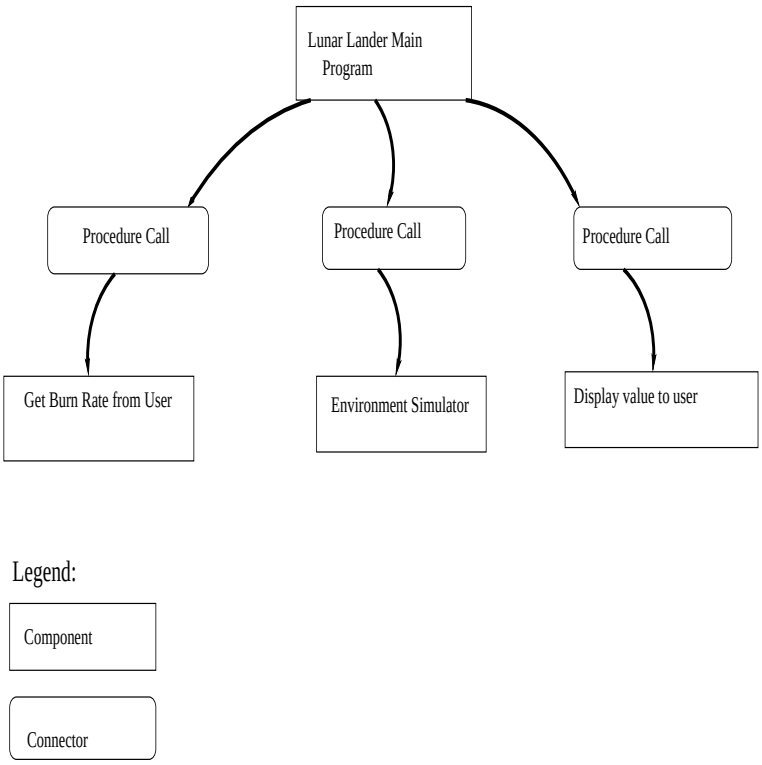


Figure 4.1: Example of a software architecture from Taylor et al. (2009)

inside the software connector to guarantee security. Other approaches such as the one in Magee and Kramer (1996) and Deng et al. (2003) are focused on implementing an access control list within the components. The access control list specifies which components or connectors are allowed to interact with the list holder. The access control lists are, in most of the cases, local to each connector and component rather than being a global one.

In terms of using AI in software security, the field of Artificial Immune System(AIS) (Hofmeyr and Forrest, 2000), from the previous chapter, has the potential of being employed in the software security domain. Even if the artificial immune system is more focused on network intrusion detection, the same concept can be applied to software security. The software components and connectors can be perceived as communicating nodes in a network. With this idea, the regular artificial immune system approach is usable with possibly a slight modification in the detector encoding.

More information about AIS are available in (Hofmeyr and Forrest, 2000).

Rather than being embedded in the software itself, software patches are often represented as a stand alone software that monitors other software's activities in a system. Classified to be among these patches software is the ClearView (Perkins et al., 2009) patch program. ClearView adopts a strategy that is close to an artificial immune system to develop an automatic patch. ClearView starts by observing the normal behavior of the system and uses the register values and the memory contents to build a model of the system. It creates patches out of the model. When a failure or an abnormal activity is detected, ClearView generates a set of candidate patches. The candidate patches are evaluated to filter out the most effective ones. The patches, when executed, can change some of the registers and memory values. Other researchers, like (Kumar and Venkataram, 1997), trained a neural network to perform anomaly detection and use case based reasoning to automatically deploy patches.

4.5 Using Q-learning in Software Security

The idea of using Q-learning in the software security field comes from the observation that the software architectural configuration is similar to a tree-like structure. The components and the connectors constitute the nodes of the tree. The interactions or the links between the components and the connectors constitute the available actions. These facts make it possible to consider using Q-learning in software security.

Using Q-learning in software security comes with an assumption that the actions taken by a component or a connector to initiate the interactions with another component or another connector are distinguishable and can be given unique identifications. This case guarantees the creation of Q-values such as $Q(connector_{id}, a)$ or $Q(component_{id}, a)$. The $connector_{id}$ and the $component_{id}$ describe the states or

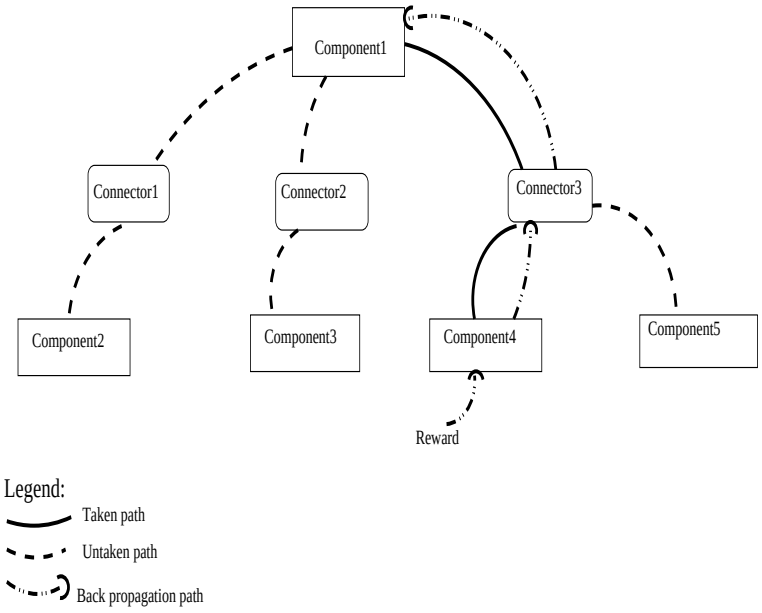


Figure 4.2: Example of a back propagation

the locations where the action, a, took place. A reward is given when the software finishes the execution of one task. The reward is back propagated to all the components and the connectors in the path taken by the completed task. If the task results in a security violation, then a negative reward is given. In the other case, a positive reward is given. By using the backup technique from Algorithm 1,

$$Q(connector_{id}, a) = (1 - \alpha)Q(connector_{id}, a) + \alpha(Reward + \gamma Q(component_{id}, a')) \tag{4.4}$$

$$Q(component_{id}, a) = (1 - \alpha)Q(component_{id}, a) + \alpha(Reward + \gamma Q(connector_{id}, a')) \tag{4.5}$$

Components and connectors are to record the Q-values assigned to them. When a task passes through a connector, for example, the connector will evaluate the action needed to execute the task. If the needed action has a Q-value below a threshold, then the connector can abstain from executing the action and reports an alert. A similar situation happens within a component. The policy, π, can be

described as follows,

$$\pi(connector_{id}, a) = \begin{cases} execute(a) & : Q(connector_{id}, a) \geq Threshold \\ notexecute(a) & : Q(connector_{id}, a) < Threshold \end{cases} \tag{4.6}$$

$$\pi(component_{id}, a) = \begin{cases} execute(a) & : Q(component_{id}, a) \geq Threshold \\ notexecute(a) & : Q(component_{id}, a) < Threshold \end{cases} \tag{4.7}$$

A human agent or a rule based agent or an artificially intelligent detector can be used to reward the Q-learning system in the beginning of the learning phase. When the Q-values and the Q-learning performance stabilize, then the human or the rule based agent can be decommissioned. This idea imposes a security learning phase during the software testing.

There are possibilities that not all of the valid actions in the software will be met during the security learning phase. Back propagation, therefore, needs to continue working as the software is deployed. Keeping back propagation allows the Q-learning security system to adapt to new attacks.

4.5.1 Extension to Network Security

This approach can be implemented in a computer network to guarantee the security of the computers involved in the network as well. In this case, the components and the connectors are replace by the computer hosts or the network nodes that interact with each other. By using a back propagation system and by rewarding the path taken by the communication's packets, each node will be able to store Q-values associated with their internal action. The goal is to make the nodes learn to only use the high rewarding internal actions. This Q-learning approach can be added as a modification to the TCP and the UDP library.

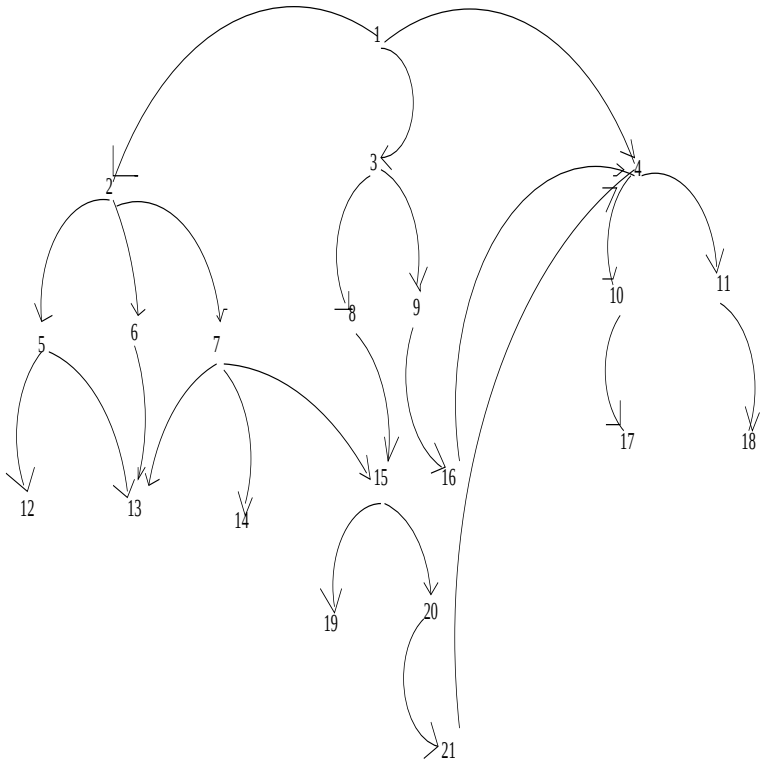


Figure 4.3: Experiment's components configuration. The components are represented by their identification number.

4.6 Experiment

4.6.1 Setting

In the experiments, a software prototype composed of 21 components and 75 connectors was created as a test bed. Each component had 5 actions that interacted with 5 different connectors. The actions are used to route the execution path of the software prototype. Each connector only interacted with one component. 6 of the components were execution path terminals. They were not connected to any connectors. The components configuration without the connectors is represented in Figure 4.3. An artificial immune system based detector that was trained to perfection and a rule based detector were used together to perform anomalies and intrusions detections. Each component and each connector dynamically built their own Q-

values. When an action was met for the first time, it was given a Q-value of 0. For each component, an action was uniquely identified by the name of the connector called and the value of the parameter passed to the connector. For each connector, an action was uniquely identified by the name of the component called and the value of the parameter passed to the component. The following back propagation equation was used,

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(Reward + \gamma Q(s',a')) \tag{4.8}$$

s described the component or connector where the action, a, was taken. s' described the next component or the next connector, down in the execution path, where the action, a', was taken. α was a learning factor and γ was a discount factor. α controlled the amount of the reward that can affect the current Q-value. γ controlled how much of the Q-value of the next component or connector in the path could affect the current Q-value.

4.6.2 Measurement Criteria

The test bed software was exposed to 275 anomalies or misuses in each completion of 100 execution paths. 120 of the anomalies were unique and the remaining were redundant copies of the unique 120. The Q-learning system was measured on the percentage of anomalies blocked and on the percentage of legal actions blocked. These percentages are displayed in figures. The performance of this Q-learning approach under the training time, learning factor and the threshold parameters were recorded. The data points represented in each figure are the average of 1000 data points collected during the experiments.

4.6.3 Result

All data points represented in the figures in this section are the average of a 1000 data points collected during the experiments.

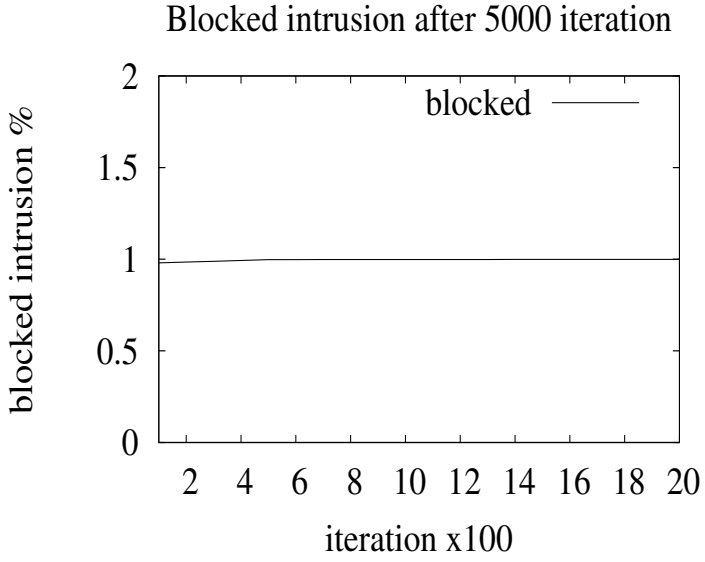


Figure 4.4: The Q-learning system trained on 5000 complete execution paths before being tested. α is 0.2 and γ is 0.9. The rewards are +20 and -20. The threshold is -10.

Training time

In this experiment, the effect of the training time was studied. The test bed software was allowed to execute some random execution paths and reward itself for some period of time before being tested on the 275 attacks. The parameters settings were as follows, $\alpha = 0.2$, $\gamma = 0.9$, reward is +20 if the detector did not trigger an alarm and -20 if the detector triggered an alarm. The threshold is set to -10.

Learning factor

The influence of the learning factor α was studied in this experiment. The parameters settings were as follows, $\gamma = 0.9$, reward is +20 if the detector did not trigger an alarm and -20 if the detector triggered an alarm. The threshold is set to -10. The test bed software was given a 5000 complete execution paths as a training time.

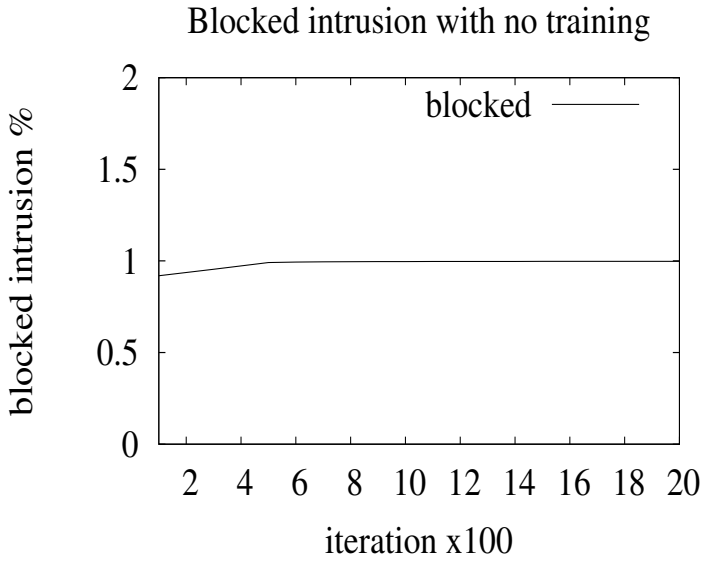


Figure 4.5: The Q-learning with no prior training before being tested. α is 0.2 and γ is 0.9. The rewards are +20 and -20. The threshold is -10.

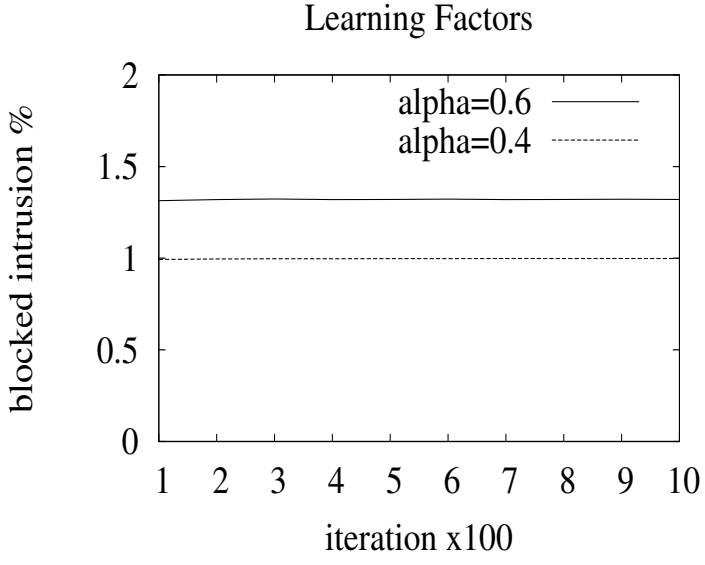


Figure 4.6: Performance of the Q-learning system under different values for α. γ is 0.9. The rewards are +20 and -20. The threshold is -10.

Threshold

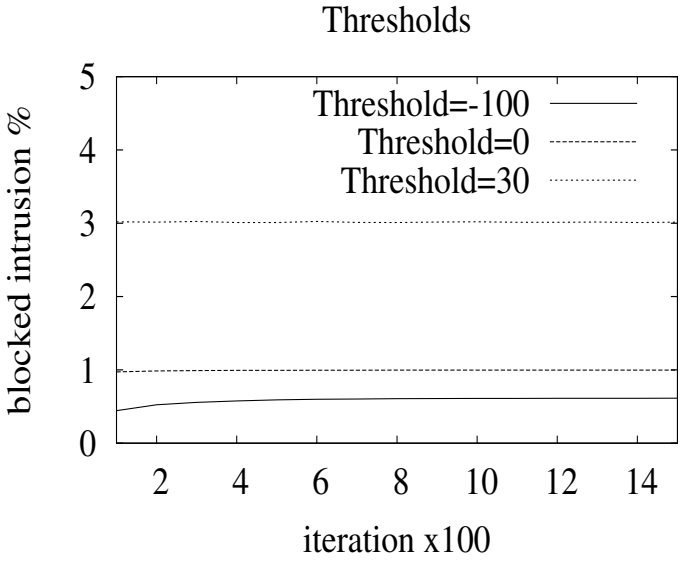


Figure 4.7: Performance of the Q-learning system under different threshold values. α = 0.2 and γ is 0.9. The rewards are +20 and -20.

In this section, the effect of the threshold values was studied. The threshold is the parameter that controls if an action is to be executed or not. The parameters settings were as follows, $\alpha = 0.2$ and $\gamma = 0.9$. The reward is +20 if the detector did not trigger an alarm and -20 if the detector triggered an alarm. The test bed software was given 5000 execution paths as a training time.

4.7 Key Findings

The training time experiment showed that prior training time can help the Q-learning system to be ready at the software's day zero. Most of the temporary repair can be preformed during the training time making most of the real attacks unsuccessful during the deployment time. For the software used in this experiment, the security temporary repairs were fully completed after approximately 500 complete execution paths, see Figure 4.5.

The value of α controls the learning factor. If α is relatively low, then only the action that triggered the anomaly is punished and prohibited from being executed. If α is high, then some of the actions that preceded the abnormal action can also be punished and banned from being executed. This is the reason why the percentage of blocked intrusion is higher than 100% in Figure 4.6 for $\alpha = 0.6$. This fact gives the system administrator a control over the repair policies to be employed. Note, however, that a high α can possibly prohibit a legitimate action from being executed.

The threshold value is required to be within the rewards interval. With a threshold that falls below the lowest reward, the system may not be able to block any attacks or it may take long a time before the system can be equipped with all of the necessary temporary repairs. With a threshold higher than the highest reward, all executions are blocked rendering the system unable to execute any action. Figure 4.7 summarizes these cases.

CHAPTER V

DEFENSE: GENERAL GAME PLAYER AS NETWORK DEFENDER

5.1 Introduction

Chapter III provided a way to use artificial intelligence to detect network intrusions. Chapter IV explained how Q-learning could be used to temporary repair an intrusion that could induce abnormal activities in a software. This chapter will focus on handling and responding to an attack once it was detected.

Dropping a suspicious packet, ignoring an action request, and shutting down infected infrastructures can all be considered as valid responses to a computer security attack. The appropriate response that needs to be carried out depends on the owner of the system's objective. Some companies prefer to shut everything down when a security attack is detected. Others prefer to make parts of the computer system available so that important tasks can still be accomplished.

The risks involved with the attack and the desired answers need to be analyzed to guarantee that the organization and the computer system being attacked can maintain the basic functionalities required. Most of the time, the signatures or the behaviors of the attackers are not known in advance. This makes it difficult to create a library system that will have all the possible counter-measures to any possible attacks.

The general game playing field appears to be able to help with this situation. The intruder is an opponent that the system has no knowledge about. The system cannot always know ahead of time the intent and the goal of the intruder. The system however knows what assets it has and which assets are valuable and which are not.

It also knows about the actions that can be used to manipulate, move and protect the assets. The intruder and the system can be viewed as two players playing a game where each player tries to hold and control the assets involved in the computer system.

In this chapter, a computer system attack defender based on a general game player is proposed. The goal is to create an attack responder that will learn by itself to defend a computer system from any unknown intruders without waiting for the intruders' attack signatures update.

5.2 General Game Playing

Let us start by explaining what a general game player is. General game playing is a competition, initiated by Stanford University and sponsored by the International Joint Conferences on Artificial Intelligence(IJCAI). In this competition, participants are required to create an artificially intelligent game player. The AI game player needs to have the ability to play various games without being told what kind of game it is playing. The main idea is to make the participants create one general algorithm that can be used in any game situations. Any AI players that compete in this competition need to be equipped with knowledge representation, reasoning, learning and rational decision making (Genesereth and Love, 2005).

The majority of the AI players that participate in this competition are either a simulated or a non-simulated player. Kuhlmann and Stone (2007) introduced a non-simulated player that detects the similarity between an unknown game with a known one so that state-value functions can be transferred to the unknown game to better play the game and to predict the opponent. In the simulated approaches, Finnsson and Bjornsson (2008) use a simulation directed by a variant of the Monte-Carlo method called *Upper Confidence-bounds Applied to Trees* to calculate value functions related to the unknown game. Finnsson and Bjornsson (2008) created an AI player,

called CADIAPLAYER, that builds a game-tree that represents the possible rewards that can be received in each state encountered during the simulation. It builds a separate tree for the opponents in order to predict and exploit them.

5.3 Monte-Carlo, Local Regression and Context Based General Game player as Network Defender

Creating a real time general game player that is capable of playing various games without any prior training is a difficult and an interesting task. Reinforcement learning appears to be the best candidate method that can be used to create such game player. The general game player, however, not only inherits the curse of dimension but faces the problem of time constraint as well. The games played by the general game player can be of various complexity and can go from a simple tic-tac-toe to a Go game. The thinking or simulation time allowed for each game to be played is very short, a few seconds in general. The previous facts render it impossible to use straight forward reinforcement learning methods or tree structures to create a general game player. This situation, however, did not stop many researchers from creating algorithms that are capable of dealing with the curses previously listed.

Monte-Carlo Tree Search(MCTS) is one of the algorithms commonly used to create a general game player (Finnsson and Bjornsson, 2008) (Mhat and Cazenave, 2011). Each MCTS-based general game player imposes its own modification to the original and basic MCTS. In the same way, this chapter provides explanations about how a local regression and a context-based opponent modeling can be used to improve the simulation part of the original MCTS. The modified MCTS is, then, used as a computer network defender against unknown attackers. The following subsection will explain more about Monte-Carlo Tree Search , local regression and context-based opponent modeling.

5.3.1 Monte-Carlo Tree Search : MCTS

Monte-Carlo Tree Search is a search method closely related to a minimax and an alpha-beta search. The learning or the environment is represented by a tree-like graph. The nodes of the tree represent the states in the environment. Each branch from each node of the tree describes an action that can be taken from the state represented by the node. The tree has starting states and final or goal states. There is a reward, r, associated with taking an action, a, from a state, s. State-action values are created to describe the quality of taking an action, a, while in a state, s. This value is represented by $Q(s, a)$. If the environment is in a state, s, and the learner took an action, a, then the environment transitions into a new state, s', where the learner can take a new action, a'.

Monte-Carlo Tree Search builds a game tree, as previously explained, in memory. MCTS is composed of four steps (Winands et al., 2008). The first step is called *selection*. In this step, the game tree is traversed and an action is selected to move the environment to a new state or a new node. The second step is called *simulation*. In this step, MCTS performs playouts that go from a starting state or node to an end node. The third step, *back propagation*, consists of forwarding back to each node, involved in a playout, the reward received at the end of a playout. The last step, called *tree expansion*, consists of deciding which node(s), encountered in the simulation, is(are) to be added to the search tree.

MCTS requires a considerable amount of simulation time to be allocated to promise that the search converges in to an optimal policy. A policy is a mapping from states to actions that will maximize or minimize the long term reward of the learner. The goal of the simulation is to visit all the possible pairs of state-action, in the tree, multiple times (Sutton and Barto, 1998). An important amount of simulation time is not always possible for real time general game players. This triggers the many attempts by different researchers to modify some or most of the MCTS

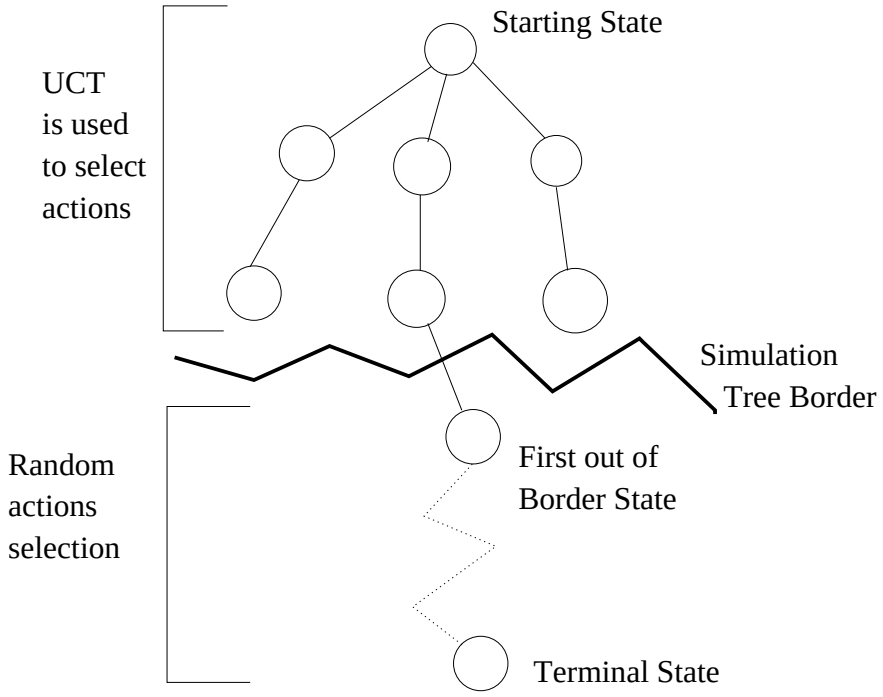


Figure 5.1: MCTS with an UCT selection

steps. Hammersley and Handscomb (1964) explained in their book the probabilistic and statistical theory behind the Monte-Carlo methods.

5.3.2 MCTS Steps

Selection

While traversing the game tree, greedy selection or ϵ-greedy selection can be used (Sutton and Barto, 1998). The main goal of the selection method is to find a balance between exploitation of known moves or actions and the exploration of unseen or unknown moves. The most common used strategy for action selection while traversing the tree in MCTS is called Upper Confidence bound applied to Tree(UCT)(Gelly and Silver, 2007). UCT creates new state-action using the equation,

$$Q^*_{UCT}(s,a) = Q_{UCT}(s,a) + c\sqrt{\frac{\lg n(s)}{n(s,a)}}. \tag{5.1}$$

It selects an action by following the calculation,

$$\pi_{UCT}(s) = \arg\max_{a} Q^{*}_{UCT}(s,a), \tag{5.2}$$

where $n(s,a)$ is the number of times that an action, a, has been selected from a state, s, and $n(s)$ is the total number of visits to a state, s, $n(s) = \Sigma_a n(s,a)$. In the UCT approach, if an action, a, has not been used in a state, s, then UCT defaults to selecting, in another word exploring, this action rather than using the actions that have been used.

UCT is a famous approach in attempting to solve the GO game (Enzenberger and Mueller, 2009) and becomes the method of choice in the General Game Playing researches (Genesereth and Love, 2005). Previous winners of the IJCAI General Game Playing competitions are believed to be UCT based (Finnsson and Bjornsson, 2008). More information about this method can be found in Gelly and Silver (2007). The theoretical proof showing that UCT can balance exploitation and exploration is available in Auer et al. (2002). The modified MCTS explained in this paper uses UCT as an action selection while traversing the game tree.

Simulation

The simulation consists of performing playouts. The original approach to perform a playout is to use self play. Recent MCTS-based general game players adopted the following strategy when conducting a playout. If the node or state encountered in the playout is within the game tree built in memory, then they use UCT as an action selection. In the other cases, they use a random action selection or some other knowledge based selections. This situation is illustrated by Figure 5.1. In a game or during the simulations, if the resulting state, s', from taking an action, a, is not part of the game tree, then the UCT policy, π_{UCT}, is abandoned and replaced by a random policy. This section proposes an idea to improve the action selection in

the simulation and more information will be provided by the later subsections.

Back Propagation

Various back propagation methods can be used and there is no agreed upon method that appears to be the best. MCTS back propagation methods can be similar to those used in Q-learning, Q(λ), TD(0) and TD(λ) (Sutton and Barto, 1998). The main idea is to distribute back up in the tree the total reward received at the end of a playout. As for this paper, the back propagation used is similar to the one used in Warkins'Q(λ) (Sutton and Barto, 1998). λ is called an eligibility trace parameter which allows the learner to update the Q-value of the pair state-action in an episode at any step of the episode rather than at the end of the episode. Watkins' Q(λ) perform backup once a non-greedy move is chosen in the simulation. More explanations about this approach can be found in Sutton and Barto (1998).

Algorithm 2 is the tabular approach to Watkins' Q(λ). Q(s,a) is the Q-value as defined previously. $e(s, a)$ is the eligibility trace for the pair of state-action (s,a). The eligibility trace determines how likely the pair (s,a) can undergo learning. γ is a discount factor and α is the learning factor.

Algorithm 2 Tabular Watkins's Q(λ)

for each episode **do**
  Initialize s, a
  while s is not terminal **do**
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q
    $a^* \leftarrow \arg\max_b Q(s', b)$
    $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
    $e(s, a) \leftarrow e(s, a) + 1$
    for all s, a **do**
      $Q(s, a) \leftarrow Q(s, a) + \alpha\delta e(s, a)$
      If $a' = a^*$, then $e(s, a) \leftarrow \gamma\lambda e(s, a)$
      else $e(s, a) \leftarrow 0$
    end for
    $s \leftarrow s'; a \leftarrow a'$
  end while
end for

Tree Expansion

To use memory efficiently, only the first state or node that happens to be outside the game tree is added to the tree, this is illustrated by Figure 5.1. Any states encountered following the first state out of the game tree will be ignored once the playout reaches an end node and back propagation is performed.

5.3.3 Local Regression

Local regression is a method used to approximate the relationship between a predictor variable, x_i, and a response variable, y_i. This situation is described by a function, f, such that

$$y_i = f(x_i) + \varepsilon_i. \tag{5.3}$$

f is an unknown function and ε_i is an assumed error value.

The objective is to predict the value, $f(x_0)$, for a target variable, x_0. The local regression method uses the predictor variables within the interval

$$[x_0 - span(x_0), x_0 + span(x_0)]$$

when it is computing $f(x_0)$. This is the reason why f is considered to be a local function.

To calculate $span(x_0)$, let α be a span percentage. If $\alpha = 0.1$, then 10 predictor variables out of 100 predictor variables will be use to calculate $f(x_0)$. Let

$$\Delta_i(x_0) = |x_0 - x_i|, \tag{5.4}$$

and let $q = \alpha * N$, where N is the total number of predictor variables, then define

$$span(x_0) = \Delta_q(x_0). \tag{5.5}$$

For simplicity, assume that $f(x_i)$ is a quadratic function such that,

$$f(x_i) = \beta_0 + \beta_1(x_i - x_0) + \frac{1}{2}\beta_2(x_i - x_0)^2, \tag{5.6}$$

with $x_i \epsilon [x_0 - span(x_0); x_0 + span(x_0)]$. f can be approximated by calculating the parameter $\beta = (\beta_0, \beta_1, \beta_2)'$ as follow,

$$\beta^* = \arg\min_{\beta \epsilon R^3} \sum_{i=1}^{n} w_i(x_0)[y_i - \beta_0 + \beta_1(x_i - x_0) + \frac{1}{2}\beta_2(x_i - x_0)^2]^2 \tag{5.7}$$

$$w_i(x_0) = W\left(\frac{x_i - x_0}{span(x_0)}\right) \tag{5.8}$$

$$W(u) = \begin{cases} (1 - |u|^3)^3 & : u <= 1 \\ 0 & : u > 1 \end{cases}$$

x_i can be a single variable or a vector, similar situation can happen with x_0 and y_i. This section is a simplified introduction to local regression. Any interested readers can acquire more in depth explanation from Cleveland and Devlin (1988) and Cleveland et al. (1990).

5.3.4 Context-Based Prediction

The Context-Based prediction is a prediction technique used to encode audio and video (Marpe et al., 2003). The main idea behind this technique is to code an input symbol depending on its relationship to neighbors. The Context-Based approach can be explained by the following example. Assume the following stream of input, $xyxxyyzz$, was read by the encoder. A frequency table is created to illustrate how many times each symbol has occurred in the stream, this process creates what is called context of order-0. The frequency table is extended to also count the number of occurrences of each symbol after one another, this creates the context of order-1. The frequency can be further extended to count the occurrences of each symbol after

two successive symbols, this creates the context of order-2. The order of the context can go as far as the encoder wishes.

Given the frequency table, assume that the encoder has read xy and the symbol z appears next. The encoder has the choice of coding z in order-2 with the context xy or code z in order-1 with the context y or code it in the order-0 with the context z. The advantage of using context-based coding is that after reading some part of the input stream, the decoder uses the frequency table built by the encoder to predict which symbol will likely to appear next. More detailed information about this technique can be found in Marpe et al. (2003).

Context Update

Define a context as an ordered collection of moves performed by an opponent. Assume, for example, that at time, t, an opponent, $p1$, has performed the moves $a_1a_2 \ldots a_k$ and just made a move a_t. It can be said that in the context of $a_1a_2 \ldots a_k$ the opponent's move is a_t. This can be represented by the context function, C, as follows,

$$C_{p1}(a_1a_2 \ldots a_k, a_t) = C_{p1}(a_1a_2 \ldots a_k, a_t) + 1 \tag{5.9}$$

As the opponents are allowed to change behaviors, the move performed at each context can change over time. To reflect an opponent change of behavior, a probability table is derived from the context update function. By using the same opponent, $p1$, as above, the probability of this opponent taking the action, a_t, while the context is $a_1a_2 \ldots a_k$ is calculated in the following way,

$$Prob_{p1}(a_1a_2 \ldots a_k, a_t) = \frac{C_{p1}(a_1a_2 \ldots a_k, a_t)}{sum} \tag{5.10}$$

$$sum = \sum_{i=0}^{n} C_{p1}(a_1a_2 \ldots a_k, a_i) \tag{5.11}$$

5.3.5 Idea to Improve MCTS

Instead of performing a random action selection when faced with a state that is outside of the game tree, use a local regression to perform a guided action selection. MCTS is proven to converge into an optimal policy while it is using a random action selection during the simulation. This convergence requires that a sufficient amount of simulation time is given. This requirement, however, cannot be met in the general game playing case. An alternative to the random action selection is therefore needed.

Assume that an out of game tree state, s_i is encountered. The neighbor states, for a state s_i, can be defined as the states within the interval $I_{s_i} = [s_i - span(s_i), s_i + span(s_i)]$. The query or the target variable is a pair (s_i, a_1). Once a query is obtained, define the new neighbor states as the states in the game tree and in I_{s_i} that have a recorded value for the action a_1. Let a set S represents these new neighbor states. A local regression can then be applied to S in order to estimate the value of $f(s_i, a_1)$. In this case, the predictor variables to be used in the local regression consist of all s in S. The response variables consist of all the $Q(s, a_1)$ for all s in S.

It is possible to encounter a situation where there are no states in the game tree and in I_{s_i} that have a recorded value for the action, a_1. In this case, the local regression increases the computation to a three dimensional case. The predictor variables become all pair (s, a) for any s in I_{s_i} and in the game tree. The response variables become all the $Q(s, a)$ for all s in I_{s_i} and in the game tree. It can happen that I_{s_i} is empty. In this situation, the default random action selection from the traditional MCTS method is used. In this research, it is assumed that f is a quadratic function. By setting $(s_i, a_1) = x_0$, $x_i = (s, a_1)$ for any s in I_{s_i} and in the game tree, $span(x_0) = span(s_i)$ and $span(x_i) = span(s)$ for any s in S and in the game tree, the equations from the local regression section can be used without any modifications.

The parameter β can be calculated in the following way. For any $s_1, ..., s_n$ in S,

$$Y = \begin{bmatrix} Q(s_1, a_1) \\ . \\ . \\ . \\ Q(s_n, a_1) \end{bmatrix} \tag{5.12}$$

$$X = \begin{bmatrix} 1 & (s_1, a_1) - (s_i, a_1) & ((s_1, a_1) - (s_i, a_1))^2 \\ . & . & . \\ . & . & . \\ . & . & . \\ 1 & (s_n, a_1) - (s_i, a_1) & ((s_n, a_1) - (s_i, a_1))^2 \end{bmatrix} \tag{5.13}$$

$$W = \begin{bmatrix} w_1(s_i, a_1) & 0 & 0 & ... & 0 \\ 0 & w_2(s_i, a_1) & 0 & ... & 0 \\ . & 0 & w_3(s_i, a_1) & ... & 0 \\ . & . & . & ... & 0 \\ . & . & . & ... & 0 \\ 0 & 0 & 0 & ... & w_n(s_i, a_1) \end{bmatrix} \tag{5.14}$$

Matrix multiplication is used to perform the following calculations,

$$Z = X^T * W \tag{5.15}$$

$$Y = Z * Y \tag{5.16}$$

$$X = Z * X \tag{5.17}$$

$$\beta = X^{-1} * Y \tag{5.18}$$

The value of $Q(s_i, a_1) = \beta(1, 1)$.

The simulation can also be improved by the context-based opponent modeling. In the simulation, the probability table, derived from the context update, is used to determine each opponent's following move. Like any ordinary frequency table, the move that has the highest probability is the most likely to occur. The following equation reflects this case. Assuming that $p1$'s current context is $a_1 a_2 \ldots a_k$,

$$a_t = \arg\max_{a_i} Prob_{p1}(a_1 a_2 \ldots a_k, a_i) \tag{5.19}$$

A context that has no record of the opponents' moves will be encountered in the simulation. It is in this situation that context-based prediction differs from regular frequency tables.

An opponent's context is a direct reflection of its moves. Suppose an opponent is in a context, c where

$$c = a_1 a_2 \ldots a_k \tag{5.20}$$

with a_k being the most recent move. Assume that there is no record of the actions taken by this opponent in this context, c. To predict the next move of this opponent, the least recent move is eliminated. In this case a_1 is eliminated from c to obtain a new context,

$$c' = a_2 \ldots a_k \tag{5.21}$$

The move that is predicted to be taken by the opponent in c' can be calculated by the following formula,

$$a_t = \arg\max_{a_i} Prob_{p1}(c', a_i) \tag{5.22}$$

If no move has been recorded in c', then the least recent move, in c', is eliminated again to obtain a new context, c''. This process keeps going if no move has been recorded at each new derived context. If certain numbers of new contexts have been

derived or if the context that directly represents the last move is met and there is still no move recorded at this context, then a default action selection for the opponent is used. A default action selection for the opponent can be a random selection or some sort of ε-greedy policy. In a game of two players, for example, the opponent can be defaulted to always choose the move that will give the learner the least reward. Algorithm 3 is an example of a context-based prediction.

Algorithm 3 Example of Context Based Prediction

```
1: context c = a_1 a_2 ... a_k
2: opponent move a_t = null
3: leastrecentindex = 1
4: a_t = arg max_{a_i} Prob(c, a_i)
5: while c ≠ a_k and a_t = null do
6:     c = c − a_{leastrecentindex}
7:     leastrecentindex = leastrecentindex + 1
8:     a_t = arg max_{a_i} Prob(c, a_i)
9: end while
10: if a_t = null then
11:     a_t = randommove
12: end if
13: return a_t
```

5.3.6 Related Works

A GO game player, called MoGo (Gelly et al., 2006), was the first game player that employed UCT selection and modified MCTS. MoGo used a Go game specific move selection when the action searches happened to go out of the simulation tree. MoGo strategy consists of evaluating a move based on a 3x3 Go game sub-board. The evaluation process will then decide if the move is playable or not. The creators of MoGo claimed that this approach was superior to the pure random action selection. Since MoGo's modification is purely related to the GO game, it cannot be applied to a general game playing case.

Finnsson and Bjornsson (2008) review a few of the action selection methods used in the simulation step of an MCTS. The first action selection, called Move-Average Sampling Technique(MAST), keeps track of quality values, $Q_h(a)$, for any

actions, a, seen in the simulation. $Q_h(a)$ are calculated from the back propagation values received by each node in the playout. Instead of choosing random action during the simulation, MAST uses a Gibbs distribution to assign a selection probability, $P(a)$, to each action. The selection probability is calculated in the following way,

$$P(a) = \frac{e^{Q_h(a)/\tau}}{\sum_{b=1}^{n} e^{Q_h b/\tau}}. \tag{5.23}$$

The second simulation action selection is called Tree-Only MAST. This action selection differs from MAST by only using the back propagation values of the nodes in the tree and the first node outside the tree to calculate $Q_h(a)$. The third action selection is called Predicate-Average Sampling Technique(PAST). PAST calculates predicate-action values, $Q_p(p, a)$, instead of only action values, $Q_h(a)$. The predicate p can be any predicate associated with a state, s. PAST uses the MAST selection by substituting $Q_h(a)$ with $Q_p(p', a)$ where p' is the predicate that has the highest predicate-action value for a state s. The fourth action selection, called Features-to-Action Sampling Technique(FAST), uses feature detection and TD(λ) (Sutton and Barto, 1998) to create the action values, $Q(a)$, which is used to replace the $Q_h(a)$ in MAST.

Using the back propagation values as a way to assign values to possible actions that can be taken in a state is a common idea to improve MCTS. A method called Rapid Action Value Estimation(RAVE) (Gelly and Silver, 2007), for example, uses back propagation values to not only create a state-action value, $Q(s, a)$, but an approximate state-action value, $Q_RAVE(s, a')$.In $Q_RAVE(s, a')$, the action, a', was used in one or more nodes, under s, in a playout. $Q_RAVE(s, a')$ gives an estimate of the quality of choosing the action, a', while in a state, s. It helps MCTS to make an informed action selection during simulation instead of a random one. Other approaches, like the one in Mhat and Cazenave (2011), use the results of parallel

simulations to create a combined evaluation for each move, m. However, the authors did not specify if this overall evaluation is employed in the playouts of each simulation or a random action selection is used in the playouts.

5.3.7 Transforming Monte-Carlo, Local Regression and Context Based General Game Player into Network Defender

The first step required, in order to use general game player in computer security, is to abstract the computer network system into a gaming environment. Each host in the network can be thought of as an entity that is composed of assets. Assets can consist of ports, an operating system, TCP/IP connections and many more (Futoransky et al., 2010). To control a host, the attacker must acquire either the majority of the assets or the important assets in the host. For a simple abstraction, it can be imposed that an attacker must find an open port in a host, then get access to the operating system and finally control the TCP/IP connection before it can be declared to be the one who is in control of the host.

The network attacker and defender interaction, then, can be abstracted generally to be a conquering game. The attacker's ultimate goal determine the real name of the game. The attacker may have its main goal to control one host and flood the network with unnecessary communication packets. The attacker can possibly have its main goal set to grabbing as much assets as it can in order to control the most hosts for other purposes. The intent of the attacker cannot always be known ahead of time even if a library of attack signatures is being used.

The basic moves about how to defend a computer system is known in advance. The real name of the attack being deployed by the attacker is not known. These situations make defending a computer network system against unknown attackers a general game playing problem. In this chapter, the Monte-Carlo Tree Search fortified with the local regression modeling and the context-based opponent modeling, described previously, is used as a network defender.

5.4 Experiment

In this experiment, attack simulations were carried out on a network of 100 computers. The attacker's goal is to control as many computers as possible and utilize them for purposes other than just simple communication. The attacker is required to find an open port first, then access the operating system, and then control the TCP/IP communication before it can be declared to be in control of a computer. This attack pattern is a simplified version of the scenario 1 of the 2000 DARPA Intrusion Detection Scenario-Specific. As a response the defender can scan and close unwanted ports in each host in the network if the host is not already taken by the attacker. The defender can also instruct the operating system from communicating or using an attacked port if the operating system has not already being accessed by the attacker. In total, the attacker can perform 3 actions per host or computer while the defender can perform 2 actions per host. At the end of a simulation, the defender is declared in control of a host if the host was never attacked or the defender has closed all unwanted ports in the host and the operating system was never accessed by the attacker.

Each of the 100 computers in this experiment had an unwanted open port that the attacker can exploit. In the simulation, both the attacker and the defender were allowed to performed up to 150 actions. The simulation was ended when the attacker could no longer gain control of a new host. At the end of the simulation, a count was performed to check which one of the attacker and the defender controlled the most hosts. The attacker and the defender took turn to execute actions with the attacker always one action ahead of the defender. The defender was given a thinking or simulation time of 10 seconds before answering to the attacker. The thinking or simulation time represented the imposed amount of time that the communication took to reach the attacker's host from the attacked host. An action that helped the defender to remain in control of the majority of the network nodes received a reward

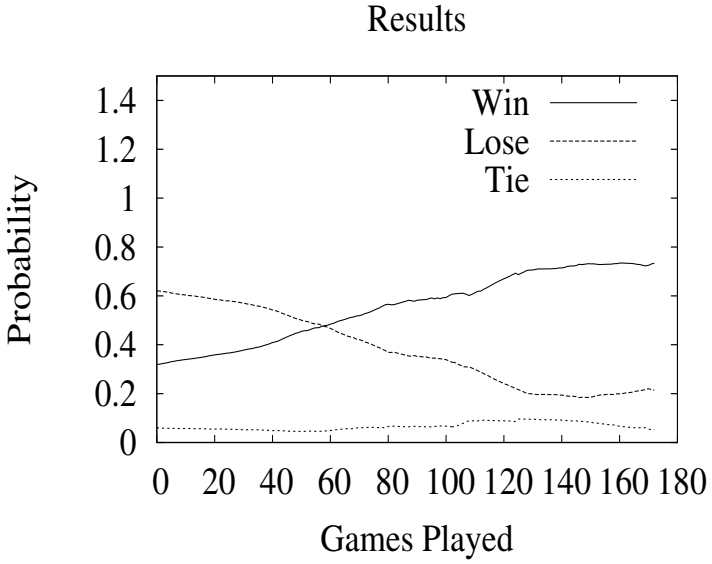


Figure 5.2: Defender's performance

of +20. An action that gave the attacker the possibility to control the majority of the network's nodes was rewarded a -50.

The data displayed in Figure 5.2 represent the performance of this Monte-Carlo, local regression, and context based network defender. These data are an average of a 1000 runs. Each run had 180 simulations. During one run, the defender kept improving the state-action values that it stored in the game tree. These state-action values were cleared and changed to random at the end of each run.

5.5 Key Findings

The general game defender's winning probability increases as the number of games played increases. This illustrates that the defender has the ability to learn from the interaction with the attacker. This situation also expresses that the defender is improving its defense strategy after each attack. As it can be seen from Figure 5.2, the proposed general game player can learn to prohibit the attacker from compromising

more than 50% of the hosts in the simulated network.

The losing probability can be lowered down to 20%. The probability for a tie game can be kept under 10%. The winning probability becomes greater than the loosing probability after the 60^{th} game. This suggests that the general game network defender needs multiple interactions with the attacker before it can develop an above average strategy to protect the network. This situation imposes a restriction that this general game approach is not suitable for a network or an organization that cannot survive multiple attacks.

Table 5.1: Table Comparing Defender's Performance to an Expected Result

Defeder's Maximum Performance	Expected Performance	One Sample p-value
73.45%	50%	p-value <0.0001

CHAPTER VI

CONCLUSIONS AND FUTURE WORKS

This research was aimed at exploring the possibilities of employing artificial intelligence in computer security. Chapter III showed that an artificial immune system and a Holland's classifier can be combined to create an intrusions detection system. Chapter IV expressed that Q-learning can be configured to provide software a temporary repair mechanism. Chapter V illustrated that a general game player based on the combination of a Monte-Carlo Search Tree, a local regression and a context-based opponent modeling is able to defend a network of computers from unknown attackers.

6.1 Summary

6.1.1 Detection: AIS-CS

By tuning the match points, the activation threshold and the tolerization parameter, the network intrusions detection system, presented in this research, is capable of achieving beyond 90% of accuracy and can provide a lower percentage of false negatives and a lower number of false positives. The experiment results show that there is a significant difference between the AIS-CS detection accuracy and the artificial immune system detection accuracy in Dasgupta and Gonzalez (2002), p-value is smaller than 0.0001 on a two tailed t-test. However, AIS-CS has not yet reach the 95% detection accuracy of the classifier systems such as in Lee and Stolfo (1999). The Holland's classifier encoding appears to add more generality and adaptivity to the intrusions detection system as it broadens the search space. It is possible with this new intrusions detection system to lower the false positives rate down to 0. This situation,

however, will impose some compromises in the accuracy of the detection. Creating a perfect intrusion detection system is a difficult task and each intrusions detector system has its down side. As in the case of using the detector system explained in this research, the network or system administrator will be faced with either trading off some of the detection accuracy for a low false positives number or accepting a high false positives number for a high detection accuracy.

The human involvement with this intrusions detection system depends on the setting and the parameters listed in the previous paragraph. If a human operator is used as a validator of the intrusion attacks, then this operator will be heavily involved after the completion of the tolerization or training period. The operator's tasks can greatly decrease when more memory detectors become available. The advantage of this approach is that it does not stop learning. It has the ability of updating itself with out waiting for the vendor of the intrusions detection system to provide the updates.

6.1.2 Repair: Q-learning Approach

Creating a software that can temporarily repair itself is possible with the Q-learning approach that was also explained in this research. Software users do not always have to wait for security patches to come from the software's vendors or creators. This situation is beneficial in terms of security and availability since it will shorten the shut down or lock up time on the software. The Q-learning approach, explained in this research, allowed the user to control the level of the temporary repairs to be applied. Users can decide if they only want to block the single action that triggered an alarm or some of the sequence of actions before the alarm was triggered.

This Q-learning approach relied on the assumption that each action in a software or in an application can be uniquely identified. This approach also relied on the use of a fairly accurate misuses or intrusions detector that can trigger alerts.

Without the presence of the two details listed previously, it can be difficult to use this Q-learning approach. Depending on the software's size, this approach may also require a training time. At this point of the research, we cannot estimate the amount of training time needed with respect to the software's size.

6.1.3 Defense: General Game Player

Using a general game approach to defend a network is possible. The advantage of using a general game player is that it learns by itself to develop the rules and the policies needed to respond to security attacks. Using general game as a network defender requires that the network can survive multiple attacks. Similar to the setting in the experiment, the network to be protected by the the general game approach needs to support the ability of being restarted into a normal state when the defender happens to lose against the attacker.

The disadvantage involved with having to restart the network over and over into a normal state can be overcome by two options. The first option is to implement the general game defender on top of a rule-based system. In this case, the rule based system is handling the already known attacks, leaving the general game defender to only be concerned about the unknown attacks. The second option consists of training the general game defender on some known attacks. The defender, then, can calculate similarities between the unknown attacks and the already learned attacks. If similarities exist, then the defender will not start completely from scratch. This will possibly lower the number of interactions with the attacker that the defender has to perform to develop an above average defense strategy.

6.2 Limitations

All the experiments in this dissertation were simulated. No experiments on a real network infrastructure were performed. This limited the analysis to be only performed on the simulations' results. The network experiments only scaled up to

100 nodes due to resource limitations. This research did not consider networks of larger scale. In this research, all outsiders' intrusions were considered harmful. This research was not oriented to distinguishing between harmful and non-harmful intrusions.

In the artificial immune system based on Holland's classifier, it was assumed that the computers could be shielded from the outside connections for a period of time to allow the artificial immune system to train itself. If this condition was not met, then the performance of this approach would not be similar to the results presented in chapter III.

In the temporary repair approach that used the Q-learning approach, it was assumed that the actions, in the software, could always be uniquely identified. The Q-learning approach would not meet its intended goal if the actions could not be uniquely identified. This approach also relied on a fairly correct and accurate detector to trigger alarms when abnormal activities appeared. If this last condition was not met, then the performance of the approach would certainly degrade. There was a generalization disadvantage involved with this approach as well. Consider the following example. If each action in the set $\{3, 4, 5, 6, \ldots\}$ could be considered as a security violation, this approach would require that each action was met during the software execution.

In the general game network defender, it was assumed that the computer network can be abstracted to resemble a gaming environment. The attacker and the defender were assumed to be taking turns executing actions in the network. The attacker was set to be always one step ahead of the defender. This part of the research did not consider the situation where the attacker and the defender executed actions simultaneously. The general game approach, in this dissertation, assumed that the network of computers could endure multiple attacks without catastrophic losses.

6.3 From Theory to Practical

The artificial immune system based on Holland's Classifiers is intended to be employed as a filter that monitors the incoming packets into a computer. The filter's response time depends on the number of detectors employed and the data structure used. If the detectors are stored in an array-like structure without any order, then finding a match has to be done sequentially. Storing the detectors with some order will be more practical as it will improve the filter's response time. The amount of storage needed to store the detectors depends on two parameters: the number of detectors and the length of the detectors. For this research, each computer participating in the network had 100 detectors. Each detector was 30 bits long, therefore a 3000 bits were needed in each host to store the detectors. The length of the tolerization time will depend on the topology and the number of hosts in the network. The experiments illustrated that the accuracy of the detectors increases when the tolerization time increases. In a real world situation, the goal is not to find the optimal tolerization time but rather finding a tolerization time that satisfies the organization and the users' desired performance.

The temporary repair using the Q-learning approach was based on assuming that each action taken within the components and the connectors of the software can be uniquely identified. The amount of storage needed to store the Q-values depends on the number of components, the number of connectors and the number of actions. This research utilized a software prototype with 21 component and 75 connectors. The Q-values were stored as of type double requiring 8 bytes. Storing the Q-values requires $21 * 5 * 8 + 75 * 1 * 8 = 1440$ bytes. In a larger software, a generalization process like neural network or classification or a curve fitting can be used to extract an equation that can be used to estimates the Q-values. The generalization process will eliminate storing all the Q-values. Finding the Q-values for an action can be accomplished in a faster way if an ordered data structure is used to store the

Q-values. The back-propagation values can be set as return values for the function calls in the software. A problem may arises in this situation if the function calls are expected to return other computation values. This last situation requires that the back-propagation values are forwarded to the connectors and components by using a specified function call or by using a log. Due to the periodic assumption with the Q-learning approach, this temporary repair approach is not guaranteed to work well if the execution paths have circles. As previously specified, this approach can be incorporated in the TCP and UDP library. Studying the feasibility of this incorporation has been started at this point.

In the defender approach, the interaction between the defender, the network and the attacker was abstracted as a game. A total of 100 computers were participating in the network. The defender, if needed, was permitted to store up to $2^{100} * 2^{100} * 3$ Q-values dynamically. Each Q-value was 8 bytes long. Not all of these Q-values are necessary to render the system effective, pruning is needed to make the approach more practical. More complicated data structure can also be used to store the Q-values, the data structure used in this research was the simplest one. A sequential state or action search will slow the defender's simulation. This will limit the effectiveness of the approach. The probability table for the attacker's move prediction has to be stored as well. This probability table is created and updated dynamically and can occupy the same amount of storage as the Q-values. Similar to the Q-values, not the entire storage space is necessary and pruning is also required. In this research, after playing 180 games the defender generated 3MB, equivalent to 320400 states-action pairs, of Q-values and 169KB, equivalent to 21200 states-action pairs, of the attacker's move prediction probability values. The amount of time spend to find the local regression function depends on the matrix library used. Matrix libraries are most of the time of order $O(n^3)$, n is the amount of data used in the local regression. This research only used up to 10% of the Q-values stored when performing the local

regression step. In this research, the defender's simulation was limited to 10 seconds. More simulation will likely result in a better performance. In a real world situation, the goal is not to find the exact answer to the local regression or the exact amount of simulation time needed. The goal is to find a good enough local regression curve, a good enough simulation time, and enough Q-values that will guarantee defeating the attacker. A good enough solution is not expected to be the optimal solution. This defender approach is intended to develop into an automated network administrator. A real world experiments for this case will be included in the future works.

6.4 Future Works

The intrusions detection system, the temporary repair approach, and the general game network defender presented in this research need to be tested on a real and non-simulated infrastructure. Performing such tests will require time and resources.

More experiments need to be conducted to prove the scalability of the idea proposed here. The intrusions detector system needs to be tested on a computer network with thousands of computers. The Q-learning temporary repair system has to be tested with a software that is composed of thousands of components and connectors. The general game network defender shall also be tested with a network with more than a thousand computers.

In the intrusions detection system, more studies need to be conducted concerning the length of the strings that represent the detectors and the communication packets. More works need to be added on finding a better set of parameters where the compromises will be small and acceptable. Efforts also need to be provided to fully automate the learning process so that the humans' interventions can be reduced to zero.

In the Q-learning temporary repair system, experiments need to be conducted with other back propagation methods. Other back propagations can be faster and

more stable than the one employed in this research. The back propagation does not always have to wait until the end of the execution path. Rewards can be back forwarded at any point of the execution to accelerate and to shorten the learning phase. This research has not studied the effect of the discount factor γ which may play a role on the repair process as well.

Other ways have to be explored to distribute the general game player network defender to nodes in the network. The actual setting is centralized, therefore presents a single point of failure. The performances of this defender on other network attacks are also open for further explorations. Including the general game defender to be a part of an existing rule based attacks responder is also considered for future works.

The approaches that are presented in this research require more memory to be effective. Attempts to lower their memory use without losing their accuracy and effectiveness are also planned in any future works. There are numerous artificial intelligence areas that can be explored for a possible utilization in computer security. This research only touched a few of these areas. As long as the computer security and the cyber-security's problems are unsolved, more experiments will be carried on to attempt using more artificial intelligence methods in computer security.

BIBLIOGRAPHY

Auer, P., N. Cesa-Bianchi, and P. Fischer (2002, May). Finite-time analysis of the multiarmed bandit problem. *Machine Learning 47*, 235–256.

Barr, A. and E. A. Feigenbaum (Eds.) (1981). *The Handbook of Artificial Intelligence*, Volume 1. Reading, Massachusetts: Addison Wesley Longman Publishing Co., Inc.

Chen, K., Y. Lian, and Y. Zhang (2010). Automatically generating patch in binary programs using attribute-based taint analysis. In *Proceedings of the 12th international conference on Information and communications security*, Berlin, Heidelberg, pp. 367–382. Springer-Verlag.

Cleveland, R. B., W. S. Cleveland, J. E. Mcrae, and I. Terpenning (1990). Stl: A seasonal-trend decomposition procedure based on loess. *Journal of Official Statistics 6*(1), 3–73.

Cleveland, W. S. and S. J. Devlin (1988). Locally weighted regression: An approach to regression analysis by local fitting. *Journal of the American Statistical Association 83*(403), 596–610.

Corsava, S. and V. Getov (2003). Autonomous agents-based security infrastructure. In *Proceedings of the 2003 international conference on Computational science and its applications: PartII*, Berlin, Heidelberg, pp. 374–382. Springer-Verlag.

Cui, W., M. Peinado, H. Wang, and M. Locasto (2007, may). Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *IEEE Symposium on Security and Privacy*.

Dal, D., S. Abraham, A. Abraham, S. Sanyal, and M. Sanglikar (2008). Evolution induced secondary immunity: An artificial immune system based intrusion detection system. In *Proceedings of the 2008 7th Computer Information Systems and Industrial Management Applications*, Washington, DC, USA, pp. 65–70. IEEE Computer Society.

Dasgupta, D. and F. Gonzalez (2002). An immunity-based technique to characterize intrusions in computer networks. *Evolutionary Computation, IEEE Transactions on 6*(3), 281–291.

Deng, Y., J. Wang, J. J. P. Tsai, and K. Beznosov (2003, September). An approach for modeling and analysis of security system architectures. *IEEE Transactions on Knowledge and Data Engineering 15*, 1099–1119.

Du, Y., H. Wang, and Y. Pang (2004). A hidden markov models-based anomaly intrusion detection method. In *Fifth World Congress on on Intelligent Control and Automation, 2004. WCICA 2004.*, Volume 5, pp. 4348–4351.

Enzenberger, M. and M. Mueller (2009, Dec). an open-source framework for board games and go engine based on monte-carlo tree search. *Technical Report TR 09-08, Dept. of Computing Science. University of Alberta, Edmonton, Alberta, Canada*.

Finnsson, H. and Y. Bjornsson (2008). Simulation-based approach to general game playing. In *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*, pp. 259–264. AAAI Press.

Futoransky, A., L. Notarfrancesco, G. Richarte, and C. Sarraute (2010). Building computer network attacks. *Computing Research Repository abs/1006.1*.

Geib, C. W. and R. P. Goldman (2001). Plan recognition in intrusion detection systems. In *DARPA Information Survivability Conference and Exposition*.

Gelly, S. and D. Silver (2007). Combining online and offline knowledge in uct. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pp. 273–280.

Gelly, S., Y. Wang, R. Munos, and O. Teytaud (2006, November). Modification of uct with patterns in monte-carlo go. Technical report, INRIA, France.

Genesereth, M. and N. Love (2005). General game playing: Overview of the aaai competition. *AI Magazine 26(2)*, 62–72.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning* (1st ed.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Guttman, B. and E. A. Roback (1995). An introduction to computer security: The nist handbook. Technical report, Washington.

Hammersley, J. M. and D. C. Handscomb (1964). *Monte Carlo methods*. Methuen, London.

Harmer, P. K., P. D. Williams, G. H. Gunsch, and G. B. Lamont (2002). An artificial immune system architecture for computer security applications. *IEEE Transactions on Evolutionary Computation 6*, 252–280.

Harrop, W. and G. Armitage (2006). Modifying first person shooter games to perform real time network monitoring and control tasks. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, New York, NY, USA. ACM.

Henry, W. P. and P. H. Winston (2004). *Artificial Intelligence*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.

Hofmeyr, S. A. and S. Forrest (1999). Immunity by design: An artificial immune system. In *In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, Volume 2, pp. 1289–1296.

Hofmeyr, S. A. and S. A. Forrest (2000, December). Architecture for an artificial immune system. *Evolutionary Computation 8*, 443–473.

Holland, J. H. (1985). Properties of the bucket brigade. In *Proceedings of the 1st International Conference on Genetic Algorithms*, Hillsdale, NJ, USA, pp. 1–7. L. Erlbaum Associates Inc.

Holland, J. H. and J. S. Reitman (1977, June). Cognitive systems based on adaptive algorithms. *SIGART Bull.*, 49–49.

Huang, J. K. and B.-s. Chen (2007). Srml learning game theory with application to internet security and management systems. In *Proceedings of the 2007 IEEE International Conference on Granular Computing*, Washington, DC, USA. IEEE Computer Society.

Kim, D. and J. Park (2003). Network-based intrusion detection with support vector machines. In H.-K. Kahng (Ed.), *Information Networking*, Volume 2662 of *Lecture Notes in Computer Science*, pp. 747–756. Springer Berlin / Heidelberg.

Kim, J., P. J. Bentley, U. Aickelin, J. Greensmith, G. Tedesco, and J. Twycross (2007, December). Immune system approaches to intrusion detection — a review. *Natural Computing 6*, 413–466.

Koziol, J. (2003). *Intrusion Detection with Snort* (1 ed.). Indianapolis, IN, USA: Sams.

Kuhlmann, G. and P. Stone (2007). Graph-based domain mapping for transfer learning in general games. In *ECML '07: Proceedings of the 18th European conference on Machine Learning*, Berlin, Heidelberg, pp. 188–200. Springer-Verlag.

Kumar, G. P. and P. Venkataram (1997). Artificial intelligence approaches to network management: recent advances and a survey. *Computer Communications 20*(15), 1313 – 1322.

Landwehr, C. (2008). Cyber security and artificial intelligence: from fixing the plumbing to smart water. In *Proceedings of the 1st ACM workshop on Workshop on AISec*, AISec '08, New York, NY, USA, pp. 51–52. ACM.

Lee, W. and S. J. Stolfo (1999). A data mining framework for building intrusion detection models. *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 120–132.

Li, W. (2004). Using genetic algorithm for network intrusion detection. In *Proceedings of the United States Department of Energy Cyber Security Group 2004 Training Conference*, pp. 24–27.

Lin, Z., X. Jiang, D. Xu, B. Mao, and L. Xie (2007). Autopag: towards automated software patch generation with source code root cause identification and repair. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, New York, NY, USA. ACM.

Lu, W. and I. Traore (2004). Detecting new forms of network intrusion using genetic programming. *Computational Intelligence*, 475–494.

Magee, J. and J. Kramer (1996). Dynamic structure in software architectures. In *In Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

Marpe, D., H. Schwarz, and T. Wiegand (2003). Context-based adaptive binary arithmetic coding in the h.264/avc video compression standard. *IEEE Transactions on Circuits and Systems for Video Technology 13*, 620–635.

Mhat, J. and T. Cazenave (2011, March). A parallel general game player. *KI journal 25*(1), 43–47.

Nathan, P. and M. W. Erwin (2004). On the rules of engagement for information warfare. Symbiot, Inc Publication. `http://www.symbiot.com`.

Nathan, P. and W. Hurley (2004). Non-equilibrium risk models in enterprise network security. Symbiot, Inc Publication. `http://www.symbiot.com`.

Oppliger, R. (2000). *Security technologies for the World Wide Web*. Norwood, MA, USA: Artech House, Inc.

Pachghare, V., P. Kulkarni, and D. M. Nikam (2009, july). Intrusion detection system using self organizing maps. In *Intelligent Agent Multi-Agent Systems, 2009. IAMA 2009. International Conference on*, pp. 1–5.

Perkins, J. H., S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard (2009). Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, New York, NY, USA, pp. 87–102. ACM.

Portnoy, L., E. Eskin, and S. Stolfo (2001). Intrusion detection with unlabelled data using clustering. In *Proceedings of ACM CSS Workshop on Data Mining Applied to Security (DMSA-2001)*, pp. 5–8.

Ren, J. and R. N. Taylor (2005). A secure software architecture description language. In *In Workshop on Software Security Assurance Tools, Techniques, and Metrics*.

Rich, E. and K. Knight (1990). *Artificial Intelligence* (2nd ed.). McGraw-Hill Higher Education.

Rubinstein, B. I. P. (2010, May). *Secure Learning and Learning for Security: Research in the Intersection*. Ph. D. thesis, EECS Department, University of California, Berkeley.

Russell, D. and G. T. Gangemi, Sr. (1991). *Computer security basics*. Sebastopol, CA, USA: O'Reilly & Associates, Inc.

Russell, S. J., P. Norvig, J. F. Candy, J. M. Malik, and D. D. Edwards (1996). *Artificial intelligence: a modern approach*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Schwetman, H. (2009). User's guide: Article reprints : Csim 18 - the simulation engine. Mesquite Software, Inc. `http://www.mesquite.com`.

Shapiro, S. C. (1992). Editor's foreword. *Encyclopedia of Artificial Intelligence 1*, 54–57.

Solomon, M. G. and M. Chapple (2005). *Information Security Illuminated*. USA: Jones and Bartlett Publishers, Inc.

Sun, F., X. Han, J. Wang, F. Sun, X. Han, and J. Wang (2010). An immune danger theory inspired model for network security monitoring. In *Proceedings of the 2010 International Conference on Challenges in Environmental Science and Computer Engineering - Volume 02*, CESCE '10, Washington, DC, USA. IEEE Computer Society.

Sutton, R. S. and A. G. Barto (1998, June). *Reinforcement Learning: An Introduction*. Cambridge,Massachusetts: The MIT Press.

Taylor, R. N., N. Medvidovic, and E. M. Dashofy (2009, jan). *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing.

Tesauro, G., J. O. Kephart, and G. B. Sorkin (1996, Aug). Neural networks for computer virus recognition. In *IEEE Expert Magazine*, Volume 11, pp. 5–6.

TippingPoint. Tippingpoint. `http://dvlabs.tippingpoint.com/`.

Wang, W., X. H. Guan, and X. L. Zhang (2004). Modeling program behaviors by hidden markov models for intrusion detection. Volume 5.

Watkins, C. and P. Dayan (1992). Technical note: Q-learning. *Machine Learning 8*, 279–292.

Webber, B. L. and N. J. Nilsson (Eds.) (1981). *Readings in artificial intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Weimer, W., S. Forrest, C. Le Goues, and T. Nguyen (2010, May). Automatic program repair with evolutionary computation. *Commun. ACM 53*, 109–116.

Winands, M. H., Y. Bjornsson, and J.-T. Saito (2008). Monte-carlo tree search solver. In *Proceedings of the 6th international conference on Computers and Games*, CG '08, Berlin, Heidelberg, pp. 25–36. Springer-Verlag.

Wireshark. Wireshark. `http://www.wireshark.org/`.

Yuan, S., Q.-j. Chen, and P. Li (2009). Design of a four-layer ids model based on immune danger theory. In *Proceedings of the 5th International Conference on Wireless communications, networking and mobile computing*, WiCOM'09, Piscataway, NJ, USA, pp. 4502–4505. IEEE Press.

APPENDIX A

DATA FOR CHAPTER III

Table A.1: Match Points Data $\pm$ Standard Deviation

Match Points	Detection	False Positives x1000	False Negatives
3	0.306959$\pm$0.01	473.0319$\pm$1000	0.693041$\pm$0.01
6	0.306959$\pm$0.01	473.0319$\pm$1000	0.693041$\pm$0.01
9	0.306173$\pm$0.01	472.461668$\pm$1000	0.693827$\pm$0.01
12	0.402450$\pm$0.01	463.7006$\pm$1000	0.597550$\pm$0.01
15	0.925890$\pm$0.01	313.2941$\pm$1000	0.074110$\pm$0.01
18	0.899362$\pm$0.01	2.272576$\pm$1000	0.100638$\pm$0.01
21	0.889335$\pm$0.01	0.000000$\pm$0.00	0.110665$\pm$0.01
24	0.814654$\pm$0.01	0.000000$\pm$0.00	0.185346$\pm$0.01
27	0.282533$\pm$0.01	0.000000$\pm$0.00	0.717467$\pm$0.01
30	0.000000$\pm$0.00	0.000000$\pm$0.00	1.000000$\pm$0.00

Table A.2: Tolerization Data $\pm$ Standard Deviation

Tolerization	Detection	False Positives x1000	False Negatives
2	0.885673$\pm$0.01	363.1135$\pm$1000	0.114327$\pm$0.01
4	0.891077$\pm$0.01	339.2190$\pm$1000	0.108923$\pm$0.01
6	0.901845$\pm$0.01	350.3220$\pm$1000	0.098155$\pm$0.01
8	0.910718$\pm$0.01	340.1438$\pm$1000	0.089282$\pm$0.01
10	0.905636$\pm$0.01	345.3784$\pm$1000	0.094364$\pm$0.01
12	0.919364$\pm$0.01	322.9316$\pm$1000	0.080636$\pm$0.01
14	0.917657$\pm$0.01	305.5381$\pm$1000	0.082343$\pm$0.01
16	0.924680$\pm$0.01	294.7294$\pm$1000	0.075320$\pm$0.01
18	0.931691$\pm$0.01	294.6614$\pm$1000	0.068309$\pm$0.01
20	0.929247$\pm$0.01	273.1172$\pm$1000	0.070753$\pm$0.01

Table A.3: Number of Nodes Data $\pm$ Standard Deviation

Number of Nodes	Detection	False Positives x1000	False Negatives
5	0.967269$\pm$0.01	12.4546$\pm$1000	0.032731$\pm$0.01
10	0.972755$\pm$0.01	13.3547$\pm$1000	0.027245$\pm$0.01
20	0.947731$\pm$0.01	44.5264$\pm$1000	0.052269$\pm$0.01
30	0.961138$\pm$0.01	109.2071$\pm$1000	0.038862$\pm$0.01
40	0.914415$\pm$0.01	185.0241$\pm$1000	0.085585$\pm$0.01
50	0.925890$\pm$0.01	313.2941$\pm$1000	0.074110$\pm$0.01
60	0.862883$\pm$0.01	395.0750$\pm$1000	0.137117$\pm$0.01
70	0.907904$\pm$0.01	490.7463$\pm$1000	0.092096$\pm$0.01
80	0.861086$\pm$0.01	573.8131$\pm$1000	0.138914$\pm$0.01
90	0.830678$\pm$0.01	668.8815$\pm$1000	0.169322$\pm$0.01
95	0.606111$\pm$0.01	721.2241$\pm$1000	0.393889$\pm$0.01

Table A.4: Activation Data $\pm$ Standard Deviation

Activation	Detection	False Positives x1000	False Negatives
1	0.944317$\pm$0.01	321.8254$\pm$1000	0.055683$\pm$0.01
2	0.937543$\pm$0.01	313.2930$\pm$1000	0.062457$\pm$0.01
3	0.931777$\pm$0.01	312.2694$\pm$1000	0.068223$\pm$0.01
4	0.925890$\pm$0.01	313.2941$\pm$1000	0.074110$\pm$0.01
5	0.920406$\pm$0.01	310.9574$\pm$1000	0.079594$\pm$0.01
6	0.914399$\pm$0.01	313.2923$\pm$1000	0.085601$\pm$0.01
7	0.908633$\pm$0.01	312.0332$\pm$1000	0.091367$\pm$0.01
8	0.902544$\pm$0.01	306.5924$\pm$1000	0.097456$\pm$0.01
9	0.896778$\pm$0.01	312.6210$\pm$1000	0.103222$\pm$0.01
10	0.891012$\pm$0.01	313.0421$\pm$1000	0.108988$\pm$0.01

Table A.5: Crossover Data $^{\pm}$ Standard Deviation

Crossover	Detection	False Positives x1000	False Negatives
0.1	0.919820$^{\pm}$0.01	306.9583$^{\pm}$1000	0.080180$^{\pm}$0.01
0.2	0.920250$^{\pm}$0.01	307.3316$^{\pm}$1000	0.079750$^{\pm}$0.01
0.3	0.928505$^{\pm}$0.01	314.7778$^{\pm}$1000	0.071495$^{\pm}$0.01
0.4	0.921150$^{\pm}$0.01	310.2244$^{\pm}$1000	0.078850$^{\pm}$0.01
0.5	0.934728$^{\pm}$0.01	311.8753$^{\pm}$1000	0.065272$^{\pm}$0.01
0.6	0.931777$^{\pm}$0.01	313.2935$^{\pm}$1000	0.068223$^{\pm}$0.01
0.7	0.922850$^{\pm}$0.01	313.1491$^{\pm}$1000	0.077150$^{\pm}$0.01
0.8	0.922243$^{\pm}$0.01	305.8848$^{\pm}$1000	0.077757$^{\pm}$0.01
0.9	0.930277$^{\pm}$0.01	315.0467$^{\pm}$1000	0.069723$^{\pm}$0.01
1	0.930419$^{\pm}$0.01	313.2397$^{\pm}$1000	0.069581$^{\pm}$0.001

Table A.6: Mutation Data $^{\pm}$ Standard Deviation

Mutation	Detection	False Positives x1000	False Negatives
0.001	0.931777$^{\pm}$0.01	312.2694$^{\pm}$1000	0.068223$^{\pm}$0.01
0.01	0.926547$^{\pm}$0.01	313.0053$^{\pm}$1000	0.073453$^{\pm}$0.01
0.1	0.935716$^{\pm}$0.01	311.9911$^{\pm}$1000	0.064284$^{\pm}$0.01
0.5	0.928010$^{\pm}$0.01	315.9618$^{\pm}$1000	0.071990$^{\pm}$0.01

APPENDIX B

DATA FOR CHAPTER IV

Table B.1: Training Time Data ± 0.01

25000 iterations	15000 iterations	5000 iterations	0 iteration
0.959347	0.95892	0.959133	0.834053
0.994113	0.994081	0.993887	0.971615
0.996349	0.996333	0.996717	0.985443
0.997398	0.997487	0.997466	0.99006
0.997813	0.997814	0.99787	0.992776
0.998127	0.998106	0.998031	0.994086
0.998221	0.998214	0.998235	0.995146
0.998364	0.998401	0.99837	0.995738
0.998466	0.99839	0.99847	0.996401
0.998478	0.998565	0.998532	0.996538
0.998593	0.998643	0.998525	0.996716
0.99859	0.998624	0.998646	0.997039
0.99856	0.998582	0.998668	0.997165
0.998611	0.998639	0.998729	0.997376
0.998752	0.998686	0.9987	0.99741
0.998704	0.998755	0.998762	0.997595
0.998676	0.998734	0.998669	0.997608
0.998719	0.998744	0.998754	0.997717
0.998684	0.998805	0.998823	0.997768
0.998795	0.998769	0.998812	0.997722
0.998763	0.998787	0.998794	0.997874
0.998728	0.998794	0.99883	0.997902
0.998792	0.998805	0.998866	0.997953
0.998763	0.998884	0.998841	0.998116
0.998774	0.998823	0.998853	0.998127
0.998814	0.998859	0.998809	0.998097
0.998789	0.998863	0.998848	0.998128
0.998817	0.998878	0.998852	0.998007
0.998806	0.998874	0.998884	0.998057
0.998825	0.998903	0.998866	0.99825
0.99882	0.998883	0.99891	0.99822

Table B.2: Learning Factor Data $^{\pm}0.01$

alpha=0.4	alpha=0.6
0.950131	1.27593
0.993374	1.31391
0.995957	1.31987
0.996869	1.32277
0.997333	1.31956
0.997711	1.32029
0.997968	1.3227
0.998104	1.3198
0.998171	1.32042
0.998417	1.32178
0.998399	1.32031
0.998528	1.32368
0.998567	1.32173
0.998539	1.32097
0.998547	1.32203
0.998651	1.32242
0.998643	1.32195
0.998642	1.32096
0.998662	1.32076
0.998675	1.32203
0.998653	1.32359
0.998747	1.32088
0.998686	1.3238
0.998758	1.32143
0.998805	1.32075
0.998762	1.32209
0.998802	1.32232
0.998805	1.31948
0.998781	1.32115
0.998827	1.32216
0.998794	1.32277
0.998805	1.32089
0.99883	1.31901
0.9988	1.32211
0.998779	1.32116
0.998852	1.32236
0.998874	1.3209
0.998812	1.32174
0.998849	1.32181
0.998816	1.32027
0.998863	1.32269
0.998856	1.32091
0.998863	1.31961
0.998852	1.32343
0.998848	1.32237
0.998892	1.32039
0.99887	1.32317
0.998851	1.32074
0.998892	1.32089
0.99887	1.32161
0.998906	1.31932
0.998917	1.32059
0.99887	1.32239
0.998859	1.31825
0.998921	1.32207
0.998899	1.32294
0.998881	1.32313
0.99891	1.32372
0.998928	1.32469
0.998906	1.3234
0.998938	1.31884
0.998905	1.32037
0.998862	1.32135
0.998906	1.31986
0.998902	1.32221
0.998927	1.3245
0.998906	1.32353
0.998916	1.32494

Table B.3: Threshold Data ± 0.01

Threshold=-100	Threshold=-15	Threshold=0	Threshold=20	Threshold=30
0.137787 ± 0.06	0.833691	0.833057	0.834345	2.8062
0.444081 ± 0.03	0.971311	0.972151	0.971492	3.02021
0.52302 ± 0.02	0.98563	0.985134	0.985112	3.01496
0.555838	0.990289	0.990162	0.990351	3.02521
0.576265	0.992762	0.992664	0.992793	3.01021
0.590749	0.994317	0.994166	0.994292	3.00975
0.598367	0.995135	0.995265	0.995104	3.02486
0.602888	0.995818	0.995825	0.995666	3.011
0.606585	0.996145	0.996218	0.996114	3.01049
0.608532	0.996533	0.996482	0.996722	3.01613
0.609753	0.996736	0.996757	0.996738	3.01964
0.610513	0.99718	0.997087	0.996956	3.01283
0.61274	0.997203	0.997346	0.997181	3.0136
0.612165	0.997071	0.997385	0.997414	3.01713
0.612114	0.997412	0.99742	0.997348	3.00946
0.613762	0.997437	0.99752	0.997358	3.01551
0.614147	0.997571	0.997535	0.99767	3.02133
0.614824	0.99765	0.997831	0.997708	3.01024
0.61462	0.997761	0.997565	0.997701	3.0165
0.614361	0.9977	0.997859	0.997786	3.01359
0.616765	0.997965	0.997812	0.99793	3.01236
0.616533	0.99789	0.998012	0.998054	3.01464
0.61488	0.997937	0.99798	0.997915	3.00846
0.615079	0.997997	0.997995	0.997964	3.00982
0.61576	0.998195	0.997985	0.998128	3.00793
0.615923	0.998139	0.998151	0.998034	3.03049
0.617176	0.998093	0.998172	0.998111	3.0071
0.616102	0.998143	0.998104	0.998104	3.01749
0.615868	0.998101	0.998208	0.998064	3.01602
0.617204	0.998199	0.998049	0.998218	3.01379
0.616781	0.998274	0.998139	0.99821	3.01352
0.615664	0.99827	0.998225	0.998197	3.01723
0.617028	0.998295	0.998231	0.998224	3.01475
0.615804	0.998197	0.998298	0.998263	3.02221
0.61589	0.998279	0.998315	0.998246	3.02915
0.616733	0.998223	0.998273	0.998275	3.00973
0.616539	0.998292	0.998253	0.998345	3.01496
0.616344	0.998347	0.998385	0.998363	3.01226
0.616892	0.998342	0.998272	0.998329	3.00981
0.617194	0.998342	0.998262	0.998334	3.01196
0.618348	0.998337	0.998367	0.998313	3.01871
0.61686	0.9985	0.998368	0.998356	3.01365
0.617683	0.998363	0.998447	0.998342	3.01678
0.617437	0.998348	0.998437	0.998349	3.01623
0.61773	0.998479	0.998388	0.998414	3.01738
0.617215	0.998399	0.99846	0.998486	3.018
0.61718	0.998429	0.998397	0.998421	3.02964
0.61671	0.998402	0.998446	0.998472	3.01146
0.617083	0.998472	0.998422	0.99849	3.02466
0.618625	0.998474	0.998464	0.998487	3.00806
0.617428	0.998527	0.998458	0.998306	3.01301
0.617362	0.998415	0.998468	0.998471	3.02004
0.616663	0.99859	0.998483	0.998446	3.01551
0.61774	0.998508	0.998542	0.998472	3.02585
0.618429	0.998511	0.998472	0.998504	3.02231
0.617612	0.998509	0.998548	0.998468	3.00691
0.617183	0.998486	0.99853	0.998495	3.00988
0.617053	0.998566	0.998552	0.998523	3.00273
0.617441	0.998555	0.99854	0.998515	3.00998
0.617256	0.998529	0.998569	0.998526	3.02229
0.617071	0.998581	0.99853	0.998599	3.00952
0.616838	0.998555	0.998548	0.998524	2.99962
0.617446	0.99856	0.998599	0.998559	3.00975
0.61736	0.998518	0.998592	0.998548	3.01996
0.618064	0.998541	0.998613	0.998566	3.01081
0.617662	0.998558	0.998594	0.99859	3.02082
0.618711	0.998566	0.998592	0.998606	3.00835
0.617478	0.998639	0.998595	0.99862	3.0272
0.619399	0.998584	0.998565	0.998555	3.02468
0.618176	0.998635	0.998555	0.998664	3.00535

APPENDIX C

DATA FOR CHAPTER V

Table C.1: Defender's Performance

Win	Lose	Tie
0.31931711 ± 0.46	0.62051279 ± 0.48	0.06017003 ± 0.24
0.32085557 ± 0.46	0.61962146 ± 0.48	0.0595229 ± 0.23
0.32312664 ± 0.46	0.61775333 ± 0.48	0.05911997 ± 0.23
0.3249093 ± 0.46	0.6167399 ± 0.48	0.05835074 ± 0.23
0.32730246 ± 0.46	0.61462757 ± 0.48	0.05806991 ± 0.23
0.33067243 ± 0.46	0.61166054 ± 0.48	0.05766698 ± 0.23
0.33257719 ± 0.47	0.60967031 ± 0.48	0.05775245 ± 0.23
0.33497035 ± 0.47	0.60780218 ± 0.48	0.05722742 ± 0.23
0.33638671 ± 0.47	0.60642244 ± 0.48	0.05719079 ± 0.23
0.33816937 ± 0.47	0.6050427 ± 0.48	0.05678786 ± 0.23
0.33970784 ± 0.47	0.60354087 ± 0.48	0.05675123 ± 0.23
0.3411242 ± 0.47	0.60130644 ± 0.48	0.05756929 ± 0.23
0.34254056 ± 0.47	0.60041511 ± 0.48	0.05704426 ± 0.23
0.34407902 ± 0.47	0.59927958 ± 0.49	0.05664133 ± 0.23
0.34598378 ± 0.47	0.59741145 ± 0.49	0.0566047 ± 0.23
0.34776644 ± 0.47	0.59578752 ± 0.49	0.05644597 ± 0.23
0.349427 ± 0.47	0.59465199 ± 0.49	0.05592094 ± 0.22
0.35169807 ± 0.47	0.59266176 ± 0.49	0.05564011 ± 0.22
0.35445753 ± 0.47	0.59030523 ± 0.49	0.05523718 ± 0.22
0.35666274 ± 0.47	0.58812842 ± 0.49	0.05520878 ± 0.22
0.358621 ± 0.47	0.58644549 ± 0.49	0.05493344 ± 0.22
0.36045579 ± 0.48	0.58439215 ± 0.49	0.05515198 ± 0.22
0.36117934 ± 0.48	0.58320312 ± 0.49	0.05561748 ± 0.22
0.36356696 ± 0.48	0.58133499 ± 0.49	0.055098 ± 0.22
0.36530987 ± 0.48	0.57997798 ± 0.49	0.0547121 ± 0.22
0.36620846 ± 0.48	0.57943131 ± 0.49	0.05436018 ± 0.22
0.36826745 ± 0.48	0.57785236 ± 0.49	0.05388013 ± 0.22
0.37009896 ± 0.48	0.57620401 ± 0.49	0.05369698 ± 0.22
0.37273632 ± 0.48	0.57433588 ± 0.49	0.05292775 ± 0.22
0.37496365 ± 0.48	0.57254976 ± 0.49	0.05248655 ± 0.22
0.37813274 ± 0.48	0.56979542 ± 0.49	0.0520718 ± 0.22
0.38058695 ± 0.48	0.56756099 ± 0.49	0.05185202 ± 0.21
0.38281731 ± 0.48	0.56528586 ± 0.49	0.05189679 ± 0.21
0.38466974 ± 0.48	0.56330559 ± 0.49	0.05202463 ± 0.22
0.38776498 ± 0.48	0.56029277 ± 0.49	0.05194221 ± 0.21
0.38992456 ± 0.48	0.55818575 ± 0.49	0.05188965 ± 0.21
0.39271043 ± 0.48	0.55616911 ± 0.49	0.05112042 ± 0.21

Table C.2: Defender's Performance (continue)

Win	Lose	Tie
0.39664062 ± 0.48	0.55268495 ± 0.49	0.0506744 ± 0.21
0.4003082 ± 0.48	0.54978659 ± 0.49	0.04990517 ± 0.21
0.40418625 ± 0.48	0.54632329 ± 0.49	0.04949042 ± 0.21
0.40917841 ± 0.48	0.54210036 ± 0.49	0.04872119 ± 0.21
0.41203555 ± 0.49	0.53957289 ± 0.49	0.04839152 ± 0.21
0.41558866 ± 0.49	0.53587325 ± 0.49	0.04853804 ± 0.21
0.42133162 ± 0.49	0.53042175 ± 0.49	0.04824659 ± 0.21
0.42646648 ± 0.49	0.52605611 ± 0.49	0.04747736 ± 0.21
0.4318511 ± 0.49	0.52144073 ± 0.49	0.04670813 ± 0.20
0.43693046 ± 0.49	0.51713059 ± 0.5	0.0459389 ± 0.20
0.44194878 ± 0.49	0.51181924 ± 0.5	0.04623194 ± 0.20
0.44660079 ± 0.49	0.50793646 ± 0.5	0.04546271 ± 0.20
0.44971434 ± 0.49	0.50474965 ± 0.5	0.04553597 ± 0.20
0.45492627 ± 0.49	0.50008716 ± 0.5	0.04498652 ± 0.20
0.45728107 ± 0.49	0.49705958 ± 0.50	0.04565932 ± 0.20
0.45829026 ± 0.49	0.4950412 ± 0.50	0.04666851 ± 0.20
0.46224942 ± 0.49	0.49090954 ± 0.50	0.04684102 ± 0.20
0.46677786 ± 0.49	0.48716602 ± 0.50	0.04605609 ± 0.20
0.46952293 ± 0.49	0.48476081 ± 0.50	0.04571623 ± 0.20
0.46990056 ± 0.49	0.48437374 ± 0.50	0.04572567 ± 0.20
0.47499963 ± 0.49	0.47896248 ± 0.49	0.04603785 ± 0.20
0.47810069 ± 0.49	0.47497688 ± 0.49	0.04692239 ± 0.21
0.47985589 ± 0.49	0.47222963 ± 0.49	0.04791445 ± 0.21
0.48389163 ± 0.5	0.46697289 ± 0.49	0.04913545 ± 0.21
0.48758034 ± 0.5	0.46130488 ± 0.49	0.05111475 ± 0.21
0.49256877 ± 0.5	0.45528985 ± 0.49	0.05214136 ± 0.22
0.49744153 ± 0.5	0.44950615 ± 0.49	0.05305229 ± 0.22
0.50043253 ± 0.5	0.44506098 ± 0.49	0.05450647 ± 0.22
0.50377055 ± 0.5	0.4406158 ± 0.49	0.05561363 ± 0.23
0.50791829 ± 0.5	0.43524523 ± 0.49	0.05683647 ± 0.23
0.51183467 ± 0.5	0.43213856 ± 0.49	0.05602675 ± 0.23
0.5137185 ± 0.51	0.42914756 ± 0.49	0.05713392 ± 0.23
0.51740355 ± 0.5	0.42313253 ± 0.49	0.05946391 ± 0.23
0.51894036 ± 0.5	0.4207199 ± 0.49	0.06033973 ± 0.24
0.52227838 ± 0.49	0.41720011 ± 0.49	0.0605215 ± 0.24
0.52788031 ± 0.49	0.41095373 ± 0.49	0.06116596 ± 0.24
0.53087131 ± 0.49	0.40743394 ± 0.48	0.06169476 ± 0.24
0.53568216 ± 0.49	0.40333205 ± 0.48	0.06098579 ± 0.24
0.54053963 ± 0.49	0.39712087 ± 0.48	0.0623395 ± 0.241
0.54663136 ± 0.49	0.39186525 ± 0.48	0.06150339 ± 0.24
0.55113048 ± 0.49	0.38784391 ± 0.48	0.06102561 ± 0.24
0.55781945 ± 0.49	0.38139384 ± 0.48	0.06078672 ± 0.24
0.5636723 ± 0.49	0.37589933 ± 0.48	0.06042838 ± 0.24
0.56570288 ± 0.49	0.36897147 ± 0.48	0.06532565 ± 0.24
0.56386132 ± 0.49	0.36891109 ± 0.48	0.0672276 ± 0.24
0.56416322 ± 0.49	0.36909223 ± 0.48	0.06674457 ± 0.24
0.56781616 ± 0.49	0.36640536 ± 0.48	0.0657785 ± 0.24
0.57122759 ± 0.49	0.36359773 ± 0.48	0.06517471 ± 0.24
0.57451826 ± 0.49	0.36054857 ± 0.48	0.06493319 ± 0.24
0.57805045 ± 0.49	0.35762018 ± 0.48	0.0643294 ± 0.24

Table C.3: Defender's Performance (continue)

Win	Lose	Tie
0.58182415 ± 0.49	0.35420875 ± 0.48	0.06396712 ± 0.24
0.5801758 ± 0.49	0.35335405 ± 0.48	0.06647018 ± 0.24
0.57779485 ± 0.49	0.3558571 ± 0.48	0.06634808 ± 0.24
0.5823736 ± 0.49	0.35213305 ± 0.48	0.06549337 ± 0.24
0.5833504 ± 0.491	0.35176675 ± 0.48	0.06488287 ± 0.24
0.58444931 ± 0.49	0.35078995 ± 0.48	0.06476077 ± 0.24
0.5848156 ± 0.49	0.35078995 ± 0.48	0.06439447 ± 0.24
0.58841756 ± 0.49	0.34718799 ± 0.47	0.06439447 ± 0.24
0.5914090 ± 0.491	0.34468494 ± 0.47	0.06390607 ± 0.24
0.58792915 ± 0.49	0.34578384 ± 0.47	0.06628703 ± 0.24
0.59140901 ± 0.49	0.34315868 ± 0.47	0.06543233 ± 0.24
0.58890596 ± 0.49	0.34315868 ± 0.47	0.06793538 ± 0.24
0.59287421 ± 0.49	0.33992303 ± 0.47	0.06720278 ± 0.24
0.59372891 ± 0.49	0.33943464 ± 0.47	0.06683648 ± 0.24
0.60032232 ± 0.48	0.33357384 ± 0.47	0.06610388 ± 0.24
0.60703782 ± 0.48	0.32844563 ± 0.46	0.06451658 ± 0.24
0.60825882 ± 0.48	0.32746883 ± 0.46	0.06427238 ± 0.24
0.60935773 ± 0.48	0.32124172 ± 0.46	0.06940058 ± 0.25
0.61009033 ± 0.48	0.31574721 ± 0.46	0.07416249 ± 0.26
0.61094503 ± 0.48	0.31000851 ± 0.46	0.07904649 ± 0.26
0.60642733 ± 0.48	0.3092759 ± 0.46	0.0842968 ± 0.27
0.60129912 ± 0.48	0.309398 ± 0.46	0.0893029 ± 0.27
0.60667153 ± 0.48	0.30439189 ± 0.46	0.0889366 ± 0.27
0.61259819 ± 0.48	0.29920607 ± 0.46	0.08819577 ± 0.27
0.61877179 ± 0.48	0.29352635 ± 0.45	0.08770188 ± 0.27
0.61957436 ± 0.48	0.28821706 ± 0.45	0.09220861 ± 0.28
0.62698268 ± 0.48	0.28204345 ± 0.45	0.09097389 ± 0.28

Table C.4: Defender's Performance (continue)

Win	Lose	Tie
0.63272693±0.48	0.27654895±0.44	0.09072414±0.28
0.63911589±0.48	0.27054332±0.44	0.0903408±0.28
0.64634916±0.48	0.26372735±0.43	0.0899235±0.27
0.65242204±0.48	0.25794366±0.43	0.08963431±0.27
0.65872241±0.48	0.25222937±0.43	0.08904823±0.27
0.66421691±0.47	0.24673487±0.43	0.08904823±0.27
0.67012145±0.47	0.24115835±0.42	0.0887202±0.27
0.67508423±0.47	0.23655006±0.42	0.08836572±0.27
0.68076189±0.46	0.23142185±0.42	0.08781627±0.27
0.6866634±0.46	0.22633434±0.41	0.08700227±0.27
0.69294284±0.46	0.22095197±0.41	0.0861052±0.27
0.68699045±0.46	0.21660215±0.41	0.0964074±0.28
0.69296274±0.46	0.21134654±0.40	0.09569073±0.28
0.69830874±0.46	0.20600054±0.40	0.09569073±0.28
0.70422745±0.45	0.20040503±0.40	0.09536752±0.28
0.70573401±0.45	0.19925296±0.39	0.09501304±0.28
0.70612647±0.45	0.1988605±0.39	0.09501304±0.28
0.70865395±0.45	0.19677258±0.39	0.09457347±0.28
0.71048545±0.44	0.19585684±0.39	0.09365772±0.28
0.71012711±0.44	0.19669296±0.39	0.09317994±0.28
0.71037686±0.44	0.19644321±0.39	0.09317994±0.28
0.71037686±0.44	0.19644321±0.39	0.09317994±0.28
0.71068211±0.44	0.19613796±0.39	0.09317994±0.28
0.71159786±0.44	0.1959548±0.39	0.09244734±0.28
0.71178101±0.44	0.19577165±0.39	0.09244734±0.28
0.71416197±0.44	0.1941233±0.39	0.09171474±0.28
0.71455443±0.44	0.19373084±0.39	0.09171474±0.28
0.71651675±0.44	0.19255344±0.39	0.09092981±0.28
0.72083386±0.44	0.18980619±0.38	0.08935995±0.27
0.72231314±0.44	0.18917221±0.38	0.08851464±0.27
0.72210181±0.44	0.18938354±0.38	0.08851464±0.27
0.72442641±0.44	0.18705895±0.38	0.08851464±0.27
0.72900516±0.44	0.18339595±0.38	0.08759889±0.27
0.72825591±0.44	0.1851442±0.38	0.08659989±0.27
0.72950466±0.44	0.18489445±0.38	0.08560089±0.27
0.73164141±0.43	0.1839787±0.38	0.08437989±0.27
0.73133615±0.43	0.18550496±0.38	0.08315889±0.27
0.7310309±0.43	0.18825221±0.38	0.08071689±0.27
0.72965727±0.44	0.19099946±0.39	0.07934326±0.26
0.72828365±0.44	0.19374671±0.39	0.07796964±0.26
0.72867611±0.44	0.19492411±0.39	0.07639978±0.26
0.72906857±0.44	0.19610151±0.39	0.07482992±0.26
0.72946103±0.44	0.1972789±0.39	0.07326007±0.26
0.72985349±0.44	0.19688644±0.39	0.07326007±0.26
0.73181581±0.43	0.19649398±0.39	0.07169021±0.26
0.73220827±0.43	0.19924123±0.39	0.06855049±0.25
0.73417059±0.43	0.19884877±0.39	0.06698064±0.24
0.73456305±0.43	0.20002616±0.40	0.06541078±0.24
0.73338566±0.43	0.20277342±0.40	0.06384092±0.24
0.73377812±0.43	0.20395081±0.40	0.06227107±0.24
0.73260073±0.43	0.20512821±0.40	0.06227107±0.24
0.73168498±0.43	0.20787546±0.40	0.06043956±0.24
0.72893773±0.44	0.21062271±0.40	0.06043956±0.24
0.72802198±0.44	0.21153846±0.40	0.06043956±0.24
0.72527473±0.44	0.21428571±0.40	0.06043956±0.24
0.72252747±0.44	0.21703297±0.41	0.06043956±0.24
0.72527473±0.44	0.21978022±0.41	0.05494505±0.22
0.73076923±0.43	0.21703297±0.41	0.0521978±0.22
0.73351648±0.43	0.21428571±0.41	0.0521978±0.22