T H E U N I V E R S I T Y O F T U L S A

THE GRADUATE SCHOOL

CYBER ATTACK ANALYSIS BASED ON MARKOV PROCESS MODEL

by
Keming Zeng

A thesis submitted in partial fulfillment of

the requirements for the degree of Master of science

in the Discipline of Electrical Engineering

The Graduate School

The University of Tulsa

2017

T H E U N I V E R S I T Y O F T U L S A

THE GRADUATE SCHOOL

CYBER ATTACK ANALYSIS BASED ON MARKOV PROCESS MODEL

by
Keming Zeng

A THESIS

APPROVED FOR THE DISCIPLINE OF

ELECTRICAL ENGINEERING

By Thesis Committee

_____, Chair
Peter Hawrylak

_____
John Hale

_____
Peter LoPresti

COPYRIGHT STATEMENT

ABSTRACT

Keming Zeng (Master of Science in Electrical Engineering)

Cyber Attack Analysis Based on Markov Process Model

Directed by Dr. Peter Hawrylak

82 pp., Chapter 8:Conclusions

(151 Words)

Cyber security analysis is widely used in countering and analyzing cyber-attacks against important facilities such as commercial corporations, public infrastructures, financial institutions, and government agencies. A Markov process model based on an attack graph is an effective method to perform cyber security analysis on the attacker's strategy and to develop cyber defenses. Markov process representations of cyber systems allow the application of countermeasures to reduce the attacker's level of success. Two main strategies are investigated in this thesis, the trapping strategy and the regression strategy, both focusing on reducing the attacker's chances of reaching their intended goal.

However, these methods involves a huge amount of data, a normal CPU is not able to handle it. A high-performance heterogeneous computing platform utilizing CPUs and Intel Xeon Phis is used to solve this problem. The algorithms are implemented on this system and facilitate the evaluation of large Markov processes in a reasonable time.

ACKNOWLEDGEMENTS

I wish to express my gratitude to the committee members, Dr. Hawrylak, Dr. Hale. I really appreciate your effort and time in serving on my committee.

I would like to give a special thanks to Dr. Hawrylak for inviting me to join the research group and giving me a great help on developing algorithms and implementations and for offering me a chance to continue to pursue PhD degree.

I would like to acknowledge Dr. Hale on the routine conference and patience to listen to my weekly progress.

I also appreciate the support from my friend Ming Li, who has been always helpful not only during my thesis but also during other classes and research.

Finally, I would like to say thanks to my family, who found my tuition fees and always support my decisions. My journey to the USA has been a long dream since my childhood and it comes true because of their love and trust. This research is not only a gift but also a repayment to them.

Dedicated the thesis to my family

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

With the growing volume of cyber-attacks, corporations, public infrastructures, financial institutions, government agencies and military facilities are facing a potential danger of compromising confidential information, undermining personal privacy, hampering public order or threatening national security. The Computer Security Institution (CSI) and Federal Bureau of Intelligence (FBI) launched an investigation and found that 85 percent of corporations and government agencies were illegally invaded in 2001. Moreover, 64 percent of them were suffering from data recovery costs due to cyber-attacks in the last 12 months and the total amount of financial loss was approximately 300 million U.S. dollars. Cyber-crime continues to escalate and expand to a larger scale. In 2016, some hackers launched a large scale Distributed Denial of Service (DDoS) attack to a DNS supplier and temporarily paralyzed the majority of networks in the USA, resulting in inaccessibility to a myriad of websites and causing the loss of millions of dollars [9]. Thus, network security is becoming increasingly crucial and a reliable security system is needed to get rid of cyber-attacks.

Cyber security is a developing technique aimed at protecting computer software, hardware and data from theft or damage. The main threats proposed to cyber security systems include computer viruses, illegal access, electromagnetic radiation and hardware damage. One of the most common methods to harm a cyber-security system is illegal access. Illegal access allows attackers to fake a legal identity to enter the system and

manipulate data without any authorization. Such potential threats can be eliminated by restricting access critical data or by encrypting important information.

Multiple measures and strategies exist to help network engineers evaluate system security and establish reliable defense mechanisms. One way to analyze the reliability of a cyber-security system is to build an attack graph. The attack graph indicates all the possible ways for attackers to compromise a system [1]. The main interest of analyzing a security system is to evaluate its defenses and lower the chance of compromise [2]. An attack graph can be converted into a Markov process and used as a statistical model to make predictions about how attacks will evolve based on current observations and relevant data. Eventually, countermeasures can be taken and relevant resources can be assigned to protect the system from the attacker [3].

A common way to analyze attack graphs is to interpret them as Markov process models. Markov models were introduced by Andrei Markov in 1906 and this theory is widely used in probability models to describe a discrete-time process. They are able to predict the future states regardless of the last states. This feature allows a Markov process model to make its prediction based only on the current state and thus it also not need memory to store the previous results.

The Markov process model derived from an attack graph consists of several important matrices, including the probability matrix, the reward matrix and the steady state probability vector. The goal in this thesis is to calculate the steady state probability vector and the total expected reward by using the probability matrix and the reward matrix. This thesis will introduce an algorithm to determine the necessity of each state in the attack graph for the attacker to achieve a stated goal. Another algorithm will be

described that calculates and then optimizes the total expected reward for state transitions in the attack graph. The total expected reward can then be used to compare different countermeasures costs and effectiveness.

However, these two algorithms involve a large number of calculations due to the quantity of data and the complexity of matrix multiplication. Although a normal CPU is capable of handling complicated mathematical calculations, it is not capable of dealing with massive collections of such a computationally intensive problem. Thus, high-performance computing is needed to process such a large workload. Heterogeneous computing is one type of high-performance computing that provides users with multiple types of hardware platforms (e.g., cores and co-processors) to maximize system performance and efficiency. GPUs (graphic processing units) and CPUs have been combined to create heterogeneous computing platform. For instance, recently GPUs have become more efficient than CPUs in many aspects of linear algebra and matrix operations. A platform called the Compute Unified Device Architecture (CUDA) designed by Nvidia Corporation allows computer engineers to make full use of Nvidia GPUs specialized processing capabilities, which significantly enhances the overall performance.

In order to overcome the problem that a normal CPU does not possess enough capacity to process a large amount of data, this thesis will implement parallel algorithms to determine state necessity and calculate total expected reward for state transitions on a heterogeneous cluster. The heterogeneous cluster consists of many nodes, with each node containing two CPUs and two Intel Xeon Phis. An Intel Phi is a Many Integrated Core (MIC) processor that is good at parallel processing. Intel Phis are designed to handle parallel work while GPUs are specifically designed for graphic usage which provides

users with visible interfaces. Thus the Intel Phi has advantages over GPU in parallel computation. The CPU plays the role of host to transfer groups of data and subroutines to the Intel Phis and then aggregates the results to form the final result. The data transferred to Intel Phis will be broken down and handled by multiple massively-parallel cores on the Phi. Measuring the time spent by the programs to perform calculation (or algorithm) will be the metric used to determine how much the performance is improved when using the CPUs and Phis versus just using CPUs.

1.1 Research Objectives

The specific objectives of this thesis are:

1. Develop a parallel algorithm to determine the necessity of each state in an attack graph for reaching a particular goal state. This algorithm will support inclusion of weight (likelihood) describing importance of each transition in the algorithm.
2. Develop a parallel algorithm to determine the effectiveness of removal of a particular state based on the ranking developed in item 1.
3. Implement the algorithm in item 1 using the heterogeneous cluster consisting of CPUs, FPGAs, and Intel Phis. The target hardware is a cluster with nodes consisting of CPUs and Intel Phis.
4. Implement the algorithm in item 2 using the heterogeneous cluster consisting of CPUs, FPGAs, and Intel Phis. The target hardware is a cluster with nodes consisting of CPUs and Intel Phis.

The difference between the two algorithms relates to how the output of each is used. The algorithm in item 1 will concentrate on evaluating the necessity for each state in an attack graph by using a probability matrix and state probability matrix to calculate the probability of the attacker reaching a particular goal state. The algorithm in item 2 introduces a method to measure the total expected reward by adding up a reward to each transition in the attack graph which can be used to compare countermeasures. Both algorithms will be implemented in the heterogeneous cluster consisting of nodes containing CPUs and Intel Phis.

1.2 Document Layout

Chapter 2 introduces the basic principles and important terminologies of the Markov process model. It also explains methods to calculate the probability to reach a certain goal and the total expected reward gained by each transition. The algorithms for applying the two strategies to evaluate the necessity for each state are included in Chapter 3. Chapter 4 presents the algorithms for calculating the expected reward using the two defending strategies. Chapter 5 describes the implementation of the algorithms presented in Chapter 3, and presents the performance results when executed on the heterogeneous cluster. The implementation of the algorithms presented in Chapter 4, and their performance on the heterogeneous clusters are presented in Chapter 6. Future work is presented and discussed in Chapter 7. Finally, this thesis is summarized and concluded in Chapter 8.

1.3 Related Work

Shandilya, *et al.* [1] explain the definition of an attack graph and present a survey on the study of state of the art technologies in attack graph generation and use in security systems, which gives an overall instruction about establishing attack graphs. However, they are only working on identifying the general direction of using attack graphs while this thesis is specifically focused on analyzing a Markov process model derived from an attack graph. Sheyner and Wing interpret an attack graph as a Markov process model in [2] and they employ the value iteration method to calculate the probability for attackers to compromise a system. They not only present a method to generate attack graphs by using an existing model checker but also a method to analyze the attack graph and use minimization techniques to thwart and prevent attacks. The basic concepts and terminology of Markov process models is precisely elucidated in [4], providing the elementary ideas and essential algorithms for this thesis. One can clearly understand fundamental methods to evaluate the probability for a successful attack and measure the vulnerability of a security system by calculating the cost spent on the attack.

Franca, *et al.* [5] propose an interesting strategy called a lifting operation to significantly reduce time and cost spent on converging the steady state. A new Markov process model will be established by applying the lifting operation and this strategy is very helpful in reducing the number of iteration steps to calculate the steady state and thus lowers the complexity of matrix calculations. Although the lifting operation accelerates convergence speed, it is not closely related to necessity determination. It is merely an optimization for simplifying the necessity determination. A Markov-based attack graph for a power system is presented in [6]. Moreover, numerous experiments on

14-bus and 30-bus power systems are conducted to analyze attackers' invasion strategy. However, [6] does not derive an optimal defense strategy and apply it to their real time model. Ye, *et al.* [7] concentrate on building a Markov-based system according to historic data and detect cyber-attacks by comparing observed activity with historical normal activities. They also implement the detection technique to examine its performance.

Bylina and Potiopa [8] implement a Markov model with large matrix size to Intel Xeon Phi. They provide a two sparse matrix formats to deal with a parallel algorithm. However, [8] is aimed at investigating the effect on Intel Phi by using a different matrix format, while this research implements thread-level parallel algorithms to Intel Phis so as to accelerate the overall processing. There will be comparisons between the Intel Phi and a normal CPU for identifying performance improvement.

CHAPTER 2

OVERVIEW OF MARKOV PROCESS MODEL

This chapter provides an overview of Markov Processes and the techniques used in this thesis to process them. Then the mathematical basis is presented to use the Markov Process to quantify the cyber-security characteristics of a system. These methods use the steady state probability of the process and the calculation of total expected reward to quantify the cyber-security of a system and to identify key components within that system that contribute to its security.

There are two methods to figure out the steady state probability. The first one is pure iteration described by an equation. The second method is to first establish and then solve a set of equations. However, in some cases it is impossible to calculate the steady state probability by using the set of equations and the iteration method might be very expensive since iterating high-order matrices multiplication is a huge amount of work. However, the problem is well suited for a computer since computers are good at finishing simple and massive computation work. However, high-performance computing techniques are required to enable the computations to be completed in a timeframe that allows the findings to be useable when the state space is large. Nearly all cyber systems have large state spaces mandating the need to use high-performance computing to solve it.

2.1 Establish Markov Process Model

This section provides an introduction to the basic matrices which will be used to describe the Markov process model, determine state necessity, and to compute total expected reward. The probability matrix indicates the probability for each transition, and the reward matrix suggests reward gained from each transition, The steady state matrix states the static probability for the current location in attack graph.

2.1.1 Probability Matrix

The conversion of an attack graph to a Markov process is the first step in the process. This thesis assumes that the attack graph has already been converted into a Markov process and that the resulting Markov process is provided as an input at the start. The probability matrix is square matrix with dimensions equal to the number of states in the attack graph. Each element in the matrix represents the probability of transitioning from one state to another state or staying in the current state.

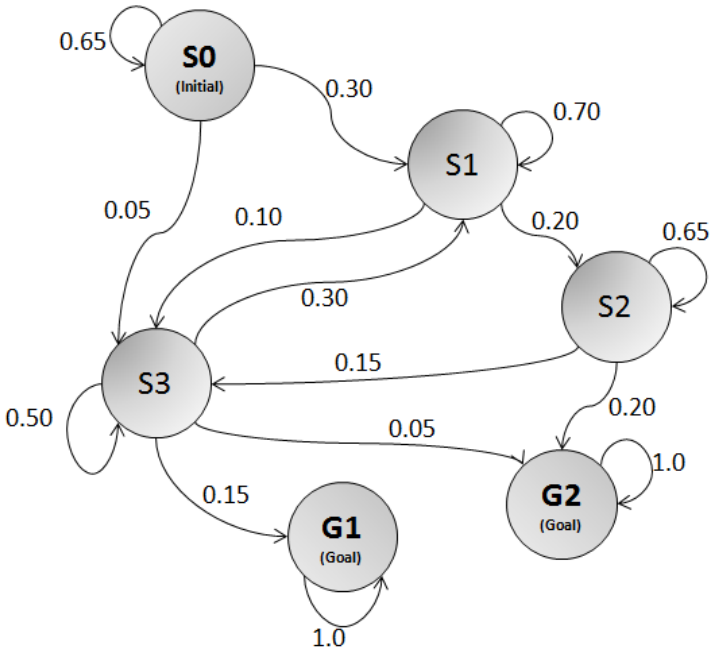

Figure 2.1: Attack Graph with Two Goal States

An attack graph with two goal states G1 and G2 is shown in Figure 2.1. There are six states in total and each state has a probability indicating how likely it is to enter other states or to stay at the current state. The arrows show orientations of the transitions (i.e., starting and ending states for a transition) while the numbers near the arrows refer to the probability to reach the next state by following the orientations that those arrows point to. Thus a probability matrix P can be obtained according to the arrows and numbers in the attack graph

$$P = \begin{bmatrix} 0.65 & 0.3 & 0 & 0.05 & 0 & 0 \\ 0 & 0.7 & 0.2 & 0.1 & 0 & 0 \\ 0 & 0 & 0.65 & 0.15 & 0 & 0.2 \\ 0 & 0.3 & 0 & 0.5 & 0.15 & 0.05 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad (2\text{-}1)$$

where P_{ij} refers to an element in i_{th} row and j_{th} column, which also represents the probability to transition from state i to state j. Diagonal elements of this matrix stand for the probability of state of entering itself. Note that the transition probabilities for state S0 is the first row of the probability matrix and that P_{1j} represents the probability of transitioning from state S0 to another state defined by column j. The sum of each row equals to 1 because it is either going to another state or staying at the same state.

2.1.2 Reward Matrix

Similar to the probability matrix, the reward matrix is also a square matrix, which has the same dimensions as the probability matrix, since each transition probability P_{ij} has a relative reward r_{ij}. What makes them different is that the elements in the reward matrix stand for the reward (gain or benefit) from each transition while elements in the

probability matrix represents the probability of each transition. Using the example Markov process shown in Figure 2.1, the reward matrix should be

$$R = \begin{bmatrix} 2 & 3 & 0 & 1 & 0 & 0 \\ 0 & 3 & 1 & 4 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 2 \\ 0 & 3 & 0 & 3 & 4 & 5 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}. \quad (2\text{-}2)$$

Matrix *R* represents the reward from state transitions. In other words, a certain amount of reward is gained if it successfully makes a state transition. The elements in the matrix *R* are r_{ij}, where r_{ij} refers to the reward gained from the transition from state i to state j For example, r_{12} stands for the reward that is received from the transition from state 1 to state 2, and r_{11} is the reward when the process to stays at state 1.

2.1.3 Steady State Matrix

In addition to the probability matrix and the reward matrix, there is another matrix called the steady state matrix, which determines how likely the system is to reach a certain state in the attack graph after a long time. The steady state matrix is a $1 \times n$ matrix where *n* is the number of states in an attack graph. The steady state probability can be used to quickly determine the effectiveness of changes to the Markov process, and by extension to the attack graph, intended to prevent the attacker from reaching a goal state. Evaluation of many different modifications provides insight into how necessary each state is to the attacker reaching a goal state. Those states that are most necessary are key candidates to investigate for deploying security countermeasures to achieve the most benefit for the effort of deploying those countermeasures.

The steady state matrix can be written as

$$\pi(n) = [\pi_1 \quad \pi_2 \quad \cdots \quad \pi_n], \tag{2-3}$$

where each element π_i stands for the probability to end up in this state and

$$\pi_1 + \pi_2 + \cdots + \pi_n = 1 \tag{2-4}$$

since it is going to stay in one of the states among all the states in the attack graph.

If the initial state in Figure 2.1 is state S0, then the steady state probability vector at the beginning of the analysis is

$$\pi(n) = [1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]. \tag{2-5}$$

2.2 Calculate Steady State Probability and Total Reward

The probability matrix, *P* is essential for calculating the steady state probability vector, and the reward matrix, *R* is also required when trying to determine the total expected reward. This section describes two common methods to calculate the steady state probability. A method to compute the total expected reward is also included.

2.2.1 Markov Iteration Method

The preferred method to calculate the steady state probability is to compute a Markov iteration by following Equation (2-6), where n is the number of steps and *P* is the probability matrix.

$$\pi(n+1) = \pi(n) \times P \tag{2-6}$$

The process in Equation (2-6) is repeated until a stable value for the π vectors is obtained. Each value in the steady state probability vector π will converge to a certain number after many iterations, which represents the final probability to reach this state.

Using the example in Figure 2.1 and starting with $n = 1$ yields Equation (2-7) and the numeric values for π vector, as shown in Table 2.1.

$$\begin{aligned} \pi(2) &= \pi(1) \times P, \\ \pi(3) &= \pi(2) \times P, \\ \pi(4) &= \pi(3) \times P, \\ &\dots \\ \pi(n+1) &= \pi(n) \times P. \end{aligned} \tag{2-7}$$

n	$\pi(n)$
1	[1,0,0,0,0,0]
2	[0.6500, 0.3000, 0, 0.0500, 0, 0]
3	[0.4225, 0.4200, 0.0600, 0.0875, 0.0075, 0.0025]
4	[0.2746, 0.4470, 0.1230, 0.1159, 0.0206, 0.0189]
...	...
1000	[0.0000, 0.0000, 0.0000, 0.0000, 0.3214, 0.6786]

Table 2.1: The Steady State Probability in The Next n Runs

Eventually the system ends up in state G1 with approximately a 32.14% chance while there is a 67.86% chance it will stop in G2. The probability it will stay in any of the other four states is 0%. This is because the system cannot get back to those four states once it enters the goal state G1 or G2. The arrows and the probabilities in the attack graph determine the likelihood to enter either G1 or G2. Note that the probabilities always converge to 0.3214 and 0.6786 when n grows very large.

Markov processes have the characteristic that they do not consider the previous states and the next transition depends only on the current state and input. Thus the new steady state probability vector can always be obtained through the current steady state probability vector no matter how the attack graph changes (it is assumed that the attack graph does not change during the analysis of the Markov process derived from the attack graph). This feature will be crucial to the calculation and use of the steady state

probability vector with the trapping and regression strategy, which will be discussed in the next chapter.

2.2.2 Derivation of Markov Process Equations

Equation (2-6) is the universal formula for calculating the steady state probability vector of a Markov process. However, this method is usually very expensive in the majority of cases because when the matrix size grows very large, the work required to multiply the probability matrix by the steady state probability vector grows very rapidly. Thus, the method is not feasible for direct application to systems with large state spaces. Further, the process described in Equation (2-6) must be performed for many iterations before the steady state probability vector converges.

Thus, a more efficient method is introduced in this section. According to the matrix multiplication laws, a set of equations can be established

$$\begin{aligned} \pi_1 &= P_{11}\pi_1 + P_{21}\pi_2 \ldots + P_{n1}\pi_n, \\ \pi_2 &= P_{12}\pi_1 + P_{22}\pi_2 \ldots + P_{n2}\pi_n, \\ \text{and} \quad \pi_3 &= P_{13}\pi_1 + P_{23}\pi_2 \ldots + P_{n3}\pi_n, \\ &\vdots \\ \pi_n &= P_{1n}\pi_1 + P_{2n}\pi_2 \ldots + P_{nn}\pi_n. \end{aligned} \tag{2-8}$$

Solving this set of equations provides a means to obtain the steady state probability vector. This method is much simpler than the iteration method in the previous section. However, sometimes the method is unlikely to give a correct answer because this approach is not a universal method for calculating the steady state probability. When one or more trapping states exist in the attack graph, this method will inevitably fail, because the method can only yield an equation that suggests the system will eventually end up in one of the trapping states and it is unable to determine the probability to stay in a certain

state. Using the example Markov process shown in Figure 2.1, the following set of equations can be found,

$$\begin{aligned} \pi_1 &= 0.65\pi_1, \\ \pi_2 &= 0.3\pi_1 + 0.7\pi_2 + 0.3\pi_4, \\ \pi_3 &= 0.2\pi_2 + 0.15\pi_3, \\ \pi_4 &= 0.1\pi_2 + 0.15\pi_3 + 0.5\pi_4, \\ \pi_5 &= 0.15\pi_4 + \pi_5, \\ \pi_6 &= 0.2\pi_3 + 0.05\pi_4 + \pi_6, \\ \pi_1 + \pi_2 + \pi_3 + \pi_4 + \pi_5 + \pi_6 &= 1. \end{aligned} \quad \text{and} \quad (2\text{-}9)$$

These equations can be reduced to two expressions,

$$\begin{aligned} \pi_1 = \pi_2 = \pi_3 = \pi_4 &= 0, \\ \pi_5 + \pi_6 &= 1. \end{aligned} \quad \text{and} \quad (2\text{-}10)$$

The result shows that the process will stop in either G1 or G2 but it is not possible to determine the probability of being in G1 verses being in G2. Thus, this method is unable to determine the final steady state probability vector. In this case, the iteration method is needed even though it might be very complicated. However, solving the set of equations can identify that it is possible to reach both goal states.

Using the example from Figure 2.1, the related probability matrix, Equation (2-1), and repeatedly applying Equation (2-6) will yield the steady state probability vector. This will indicate how likely it is to be in state G1 verses G2. The probability vector is

$$\pi = [0 \quad 0 \quad 0 \quad 0 \quad 0.32 \quad 0.68]. \quad (2\text{-}11)$$

2.2.3 Total Expected Reward

Each successful transition has a certain amount of reward associated with it. This reward can be positive (benefit) or negative (penalty or cost). Summing the reward of each transition in a path from the first transition to the last transition gives the total expected reward. The calculation of the total reward for a given path requires the storage

of the reward obtained from the previous transitions. This introduces the need for memory in the Markov process is

$$v_i(n) = \sum_{j=1}^{N} p_{ij} r_{ij} + \sum_{j=1}^{N} p_{ij} v_j\,(n-1), \tag{2-12}$$

Where p_{ij} refers to the probability of the transition from state i to state j, and r_{ij} refers to the benefit or cost gained when the transition from state i to state j happens.

$v_i(n)$ is the total number of earnings (reward obtained) in the next n transitions, and i represents the initial (starting) state. $v_i(n)$ consists of two parts $p_{ij} r_{ij}$ and $p_{ij} v_j(n-1)$. The former part, $p_{ij} r_{ij}$ is immediate earnings if the system make transitions from state i to state j and the latter part, $p_{ij} v_j(n-1)$, represents the earnings from the previous transitions if this transition starts in state j. Equation (2-13) defines a new variable q_i

$$q_i = \sum_{j=1}^{N} p_{ij} r_{ij} \tag{2-13}$$

to represent immediate earnings. Using q_i from Equation (2-13) in Equation (2-12) allows Equation (2-12) to be simplified as

$$v(n) = q + Pv(n-1). \tag{2-14}$$

Example [4] illustrates the procedure to compute the total expected reward for each transition in a 5-transition execution. It is assumed that the initial accumulated reward is zero, $v_1(0) = 0$, $v_2(0) = 0$,

$$P = \begin{bmatrix} 0.5 & 0.5 \\ 0.4 & 0.6 \end{bmatrix},$$

and

$$R = \begin{bmatrix} 9 & 3 \\ 3 & -7 \end{bmatrix},$$

therefore

$$q = P * R^T = \begin{bmatrix} 0.5 & 0.5 \\ 0.4 & 0.6 \end{bmatrix} \begin{bmatrix} 9 & 3 \\ 3 & -7 \end{bmatrix}^T = \begin{bmatrix} 6 \\ -3 \end{bmatrix}. \tag{2-15}$$

After one transition, the total expected reward is given by

$$v_1(1) = 6 + 0 = 6,$$
$$v_2(1) = -3 + 0 = -3.$$

For the second transition,

and
$$v_1(2) = q_1 + p_{11}v_1(1) + p_{12}v_2(1) = 6 + 0.5 \times 6 + 0.5 \times -3 = 7.5$$
$$v_2(2) = q_2 + p_{21}v_1(1) + p_{22}v_2(1) = -3 + 0.4 \times 6 + 0.6 \times -3 = -2.4$$

For the third transition,

and
$$v_1(3) = q_1 + p_{11}v_1(2) + p_{12}v_2(2) = 6 + 0.5 \times 7.5 + 0.5 \times -2.4 = 8.55$$
$$v_2(3) = q_2 + p_{21}v_1(2) + p_{22}v_2(2) = -3 + 0.4 \times 7.5 + 0.6 \times -2.4 = -1.44$$

For the fourth transition,

and
$$v_1(4) = q_1 + p_{11}v_1(3) + p_{12}v_2(3) = 6 + 0.5 \times 8.55 + 0.5 \times -1.44 = 9.555$$
$$v_2(4) = q_2 + p_{21}v_1(3) + p_{22}v_2(3) = -3 + 0.4 \times 8.55 + 0.6 \times -1.44 = -0.444$$

For the fifth transition,

and
$$v_1(5) = q_1 + p_{11}v_1(4) + p_{12}v_2(4) = 6 + 0.5 \times 9.555 + 0.5 \times -0.444 = 10.555$$
$$v_2(5) = q_2 + p_{21}v_1(4) + p_{22}v_2(4) = -3 + 0.4 \times 9.555 + 0.6 \times -0.444 = 0.5556$$

Table 2.2 records the total expected reward for those 5 runs.

n	0	1	2	3	4	5
$v_1(n)$	0	6	7.5	8.55	9.555	10.5555
$v_2(n)$	0	-3	-2.4	-1.44	-0.444	0.5556

Table 2.2: The total expected reward in the first five runs.

The total expected reward for this process will grow to infinity. However, the change in total reward from transition to transition can be determined. In this example, the total expected reward will increase by 1 for v_1 and v_2.

CHAPTER 3

PARALLEL ALGORITHM FOR STATE NECESSITY

This chapter describes the parallel algorithm for the heterogeneous computing cluster to determine the necessity of each state in the attack graph. This thesis investigates two defensive options available to system managers (i.e., the defenders). The first strategy is termed the trapping strategy, which eliminates all transitions leaving a particular state (the transition probability of this state is 100% back to itself) preventing the attacker from progressing any further toward their goal. The second strategy is called the regression strategy, which forces attackers to go back to a previous state (one further away from the goal states). Both of these strategies are effective means to reduce the probability that the attackers reach a goal state and keep the system from being compromised.

3.1 Trapping Strategy

3.1.1 Algorithm for Trapping Strategy

The trapping strategy aims at trapping attackers in a certain state by converting this state into a closed state. Closed state is a state that one can enter but not leave. All the transition routes out of the state except for the transition back to the state are removed. The necessity of each state in the attack graph can be measured by converting a single state to a trapping state and recalculating the new steady state probability vector. This process is then repeated for each state in the Markov process. By comparing the old and

the new steady state probability, the necessity of each state can be evaluated. This strategy identifies key states on the paths between the starting state(s) and goal state(s), which can then be the focus of security hardening procedures.

Using the Markov process illustrated in Figure 3.1, there is no need to convert state S0, G1, and G2 to trapping states because they are either initial states (S0) or goal states (G1 and G2). Those intermediate states S1, S2, and S3 (states on the path from initial state to a goal state) will be converted to trapping states in order to evaluate the importance of each of those states in reaching one of the two goal states.

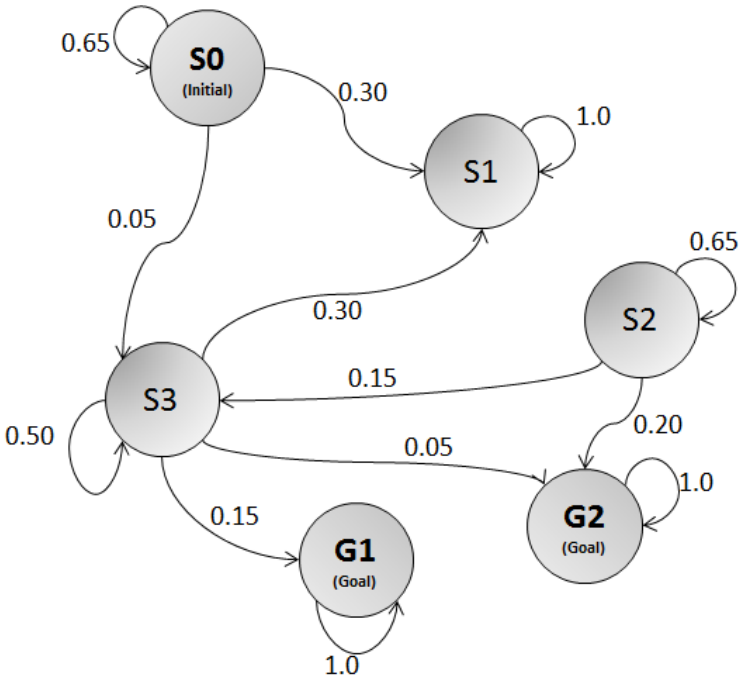

Figure 3.1: Attack Graph with Trapping State S1

To illustrate the process of computing the necessity of a state, the necessity of state S1 is calculated using the trapping strategy. The first step is to convert S1 (see Figure 2.1) into a trapping state as shown in Figure 3.1. At this point, if the attacker enters state S1, they will be stuck there. In order to calculate the new steady states probability vector, a new probability matrix is

$$P = \begin{bmatrix} 0.65 & 0.3 & 0 & 0.05 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.65 & 0.15 & 0 & 0.2 \\ 0 & 0.3 & 0 & 0.5 & 0.15 & 0.05 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \quad (3\text{-}1)$$

The steady state probability vector can be obtained by applying Equation (2-6) repeatedly. The final steady state probability vector matrix is

$$\pi = \left[0 \ \frac{33}{35} \ 0 \ 0 \ \frac{3}{70} \frac{1}{70}\right]. \quad (3\text{-}2)$$

Compared to the steady state probability of the Markov process shown in Figure 2.1, it is much less likely that the attacker will reach a goal state when state S1 is a trapping state. This difference is used to quantify the necessity of state S1.

The process is repeated for state S2 (S2 is made a trapping state in the Markov process shown in Figure 2.1). Using the initial probability

$$P = \begin{bmatrix} 0.65 & 0.3 & 0 & 0.05 & 0 & 0 \\ 0 & 0.7 & 0.2 & 0.1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0.3 & 0 & 0.5 & 0.15 & 0.05 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (3\text{-}3)$$

yields a steady state probability vector

$$\pi = \left[0 \ 0 \ \frac{11}{14} \ 0 \ \frac{9}{56} \frac{3}{56}\right]. \quad (3\text{-}4)$$

The process is repeated for state S3 (S3 is made a trapping state in the Markov process shown in Figure 2.1). Using the initial probability matrix

$$P = \begin{bmatrix} 0.65 & 0.3 & 0 & 0.05 & 0 & 0 \\ 0 & 0.7 & 0.2 & 0.1 & 0 & 0 \\ 0 & 0 & 0.65 & 0.15 & 0 & 0.2 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (3\text{-}5)$$

yields a steady state probability vector

$$\pi = \left[0\ 0\ 0\ \frac{33}{49}\ \ 0\ \frac{16}{49}\right]. \tag{3-6}$$

Table 3.1 shows the steady state probability vector components for arriving in state G1 and G2 for the three cases discussed above. Comparing the steady state probability vectors for states S1, S2, and S3 (see Table 3.1) shows the impact of making each state a trapping state on the attacker's ability to reach a goal state (state G1 or G2). It can be seen that making S1 a trapping state lowers the probability of reaching a goal state more than making S2 a trapping state. Further, making state S3 a trapping state prevents the attacker from reaching goal state G1 ($\pi_5 = 0$ in Equation (3-6)). The differences in the probability of reaching one or more goal states is used to rank the necessity of each state.

Probability	S1	S2	S3
P(G1)	$\frac{3}{70}$	$\frac{9}{56}$	0
P(G2)	$\frac{1}{70}$	$\frac{3}{56}$	$\frac{16}{49}$

Table 3.1: Probability entering goal state by using trapping strategy on each state

According to the discussion above, Table 3.1 shows the probability to enter either goal state. It clearly points out that converting state S3 into a trapping state has the most significant impact on the probability to enter the goal state G1, while converting state S1 into a trapping state plays the most important role on reducing the probability to reach the goal state G2.

3.1.2 Pseudocode for the Trapping Strategy

The Pseudocode for the algorithm to calculate the steady state probability vector is shown in Figure 3.2.

```
1.  Define matrix P, T;
2.  Define a integer k;
3.  n=size of P
4.  if k=0 then
5.     for (i=0;i<1000;i++)
6.        T=T*P;
7.  end if;
8.  if k>0 then
9.     for (i=0;i<n;;i++)
10.       if i=k then
11.          P[k][i]=1;
12.       else
13.          P[k][i]=0;
14.       end if;
15. end
16. for (i=0;i<1000;i++)
17.     T=T*P;
18. end if;
19. Print T;
```

Figure 3.2: Pseudocode for finding the steady state probability vector.

Line 1 defines the input matrix, P and the output vector, T. P is the probability matrix associated with the Markov process. T is the vector used to hold steady state probability vector that is calculated by the routine.

Line 2 defines an integer *k*, to represent the number of the trapping state in the attack graph. Line 3 defines an integer *n*, which stands for the dimension of P and T. In the case that there is no trapping state, lines 4 to 7 will apply Equation (2-6) 1000 times to obtain the steady state probability vector. In this case, only 1000 iterations are used to converge the steady state matrix. Typically 1000 iterations are sufficient for the vector to converge in the case that there are only small number of states in attack graph. However, the number of iterations will inevitably grow larger as there are more states in the attack graph.

However, if a trapping state exists, lines 8 to 14 will adjust the probability matrix by converting its diagonal element on k_{th} row to 1 and other elements on k_{th} row to 0 (making the state represented by row k a trapping state). Finally, lines 16 to 18 substitute the modified probability matrix and apply Equation (2-6) 1000 times to obtain the steady state probability vector is obtained. Line 19 displays the steady state probability vector.

3.2 Regression Strategy

3.2.1 Algorithm for Regression Strategy

The regression strategy allows attackers to move backward to the last state. Similar to the trapping strategy, each of the intermediate states is converted to a regression state and then the steady state probability is recalculated. This is repeated for all intermediate states. However, in most cases the regression strategy can only decrease the probability to reach a certain set of goal states because the regression state cannot trap attackers, rather it is only able to force an attacker to change attack routes or to take a longer route to the goal state. Therefore, it may reduce the risk to get into some specific goal states but enhance the chance to reach other goal states. In other words, it sacrifices the fact that the attacker may reach some goal states to protect other, more important, goal states.

An example using state S1 in Figure 3.3 illustrates the process of computing the necessity of a state using the regression strategy. The regression strategy is applied to state S1 with a link back to state S0. The attacker can still proceed to the goal states by

going through S3. All the transition routes out of state S1 are removed and only one backward route to state S0 is added to the attack graph.

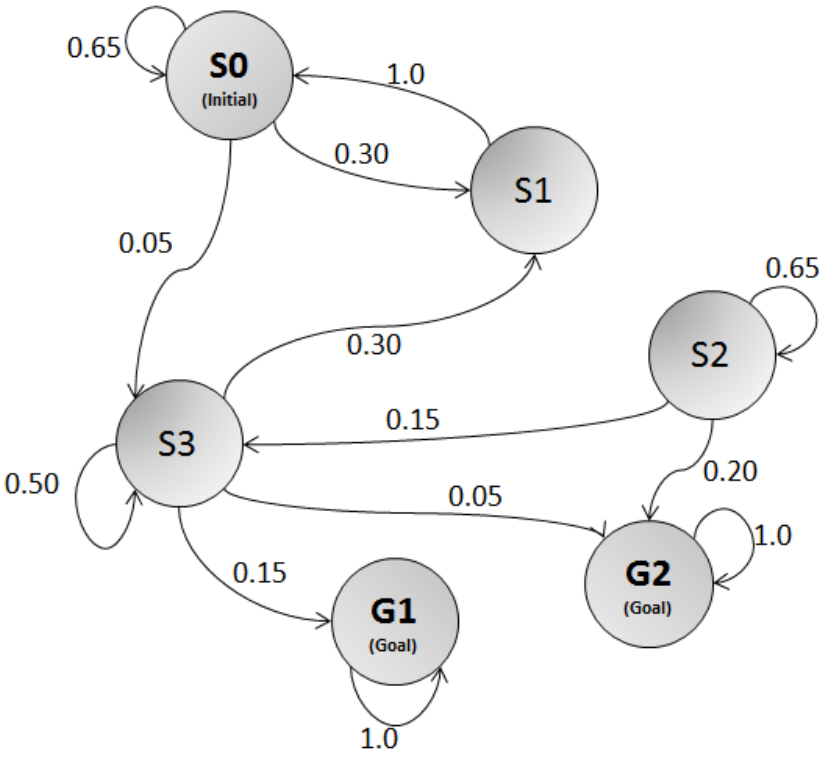

Figure 3.3: Attack graph with regression state S1.

According to Figure 3.3 the probability matrix for this process is

$$P = \begin{bmatrix} 0.65 & 0.3 & 0 & 0.05 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.65 & 0.15 & 0 & 0.2 \\ 0 & 0.3 & 0 & 0.5 & 0.15 & 0.05 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3-7}$$

Applying Equation (2-6) for many iterations yields the following steady state probability vector is

$$\pi = \left[0\ 0\ 0\ 0\ \frac{3}{4}\ \frac{1}{4}\right]. \tag{3-8}$$

Thus, the attacker is more likely (75% of the time) to reach goal state G1 and has only a 25% chance of reaching goal state G2. In this case, the attacker is destined to reach

one of the goal states but the probability of which state is entered is different from the original probability. The probability to reach G1 significantly rises from $9/28$ (original steady state probability) to $3/4$ and the probability to get into G2 drops from $19/28$ (original steady state probability) to $1/4$. Thus, one use of the regression strategy is to funnel the attacker into a specific group of states (goal or otherwise).

This process is repeated by making state S2 the regression state and the steady state probability vector is recomputed. The resulting probability matrix based on the process in Figure 2.1 modified to have S2 as the regression state (going to S1) is

$$P = \begin{bmatrix} 0.65 & 0.3 & 0 & 0.05 & 0 & 0 \\ 0 & 0.7 & 0.2 & 0.1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0.3 & 0 & 0.5 & 0.15 & 0.05 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3-9}$$

The resulting steady state probability vector is

$$\pi = \left[0\ 0\ 0\ 0\ \tfrac{3}{4}\tfrac{1}{4}\right]. \tag{3-10}$$

The result is the same as for the case where S1 was the regression state indicating that S1 and S2 affect the movement of the attacker toward the goal states in a similar fashion.

As a final example, consider state S3 as the regression state. There are multiple states (state S0, S1, and S2) that are able to be the regression state (state the attacker is forced into) from state S3. Thus, one of these states must be selected as the regression state for state S3. However, in this example, state S3 will move backward to the state whose state number is the smallest, so it will return to S0. In practice the steady state probability vector would be computed for each possibility and the best option (state S0,

S1, or S2) selected for the regression state. The resulting probability matrix based on the Markov process from Figure 2.1 with state S3 as the regression state is

$$P = \begin{bmatrix} 0.65 & 0.3 & 0 & 0.05 & 0 & 0 \\ 0 & 0.7 & 0.2 & 0.1 & 0 & 0 \\ 0 & 0 & 0.65 & 0.15 & 0 & 0.2 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3-11}$$

The steady state probability vector is

$$\pi = [0\ 0\ 0\ 0\ 0\ 1]. \tag{3-12}$$

There are three states S0 S1 and S2 that are able to enter S3, which means once S3 is converted to a regression state, the hacker will be sent back to one of these three states. However, S3 is the only intermediate state to enter G1 and the system will only end up in G1 or G2 after a long time because G1 and G2 are the only trapping states in the process. So the probability to get into G1 is zero no matter which state that the hacker is sent to from state S3. Now that it is impossible to enter G1, then the only option is for the system end up in the state G2 after a long time because there are no other trapping states in the process.

3.2.2 Pseudocode for the Regression Strategy

The pseudocode for the regression strategy is similar to the code for the trapping strategy (Section 3.1.2). The difference from the trapping strategy is how the probability matrix is modified because the alterations to the original Markov process are different: the trapping strategy makes a trapping state, while the regression strategy just move the attacker backwards in the path to a goal state or states.

```
1.  Define matrix P, T;
2.  Define a integer k;
3.  if k=0 then
4.      for (i=0;i<1000;i++)
5.      T=T*P;
6.  end if;
7.      if k>0 then
8.      breakflag=0;
9.  for (i=0;i<n;;i++)
10. if i!=k & P[k][i]!=0 & breakflag==0 then
11.     P[k][i]=1;
12.     breakflag=1;
13.     next
14. elsif breakflag==1 then
15.     P[k][i]=0;
16. end if;
17  end;
18. for (i=0;i<1000;i++)
19.     T=T*P;
20. end if;
21. Print T;
```

Figure 3.4: Pseudocode for the regression strategy algorithm.

Lines 1 and 2 define the probability matrix, the steady state probability vector, and the number of regression state. Lines 3 to 5 deal with the situation that no regression strategy is applied. Lines 7 to 17 modify the probability matrix to implement the regression state strategy. Note that P[k][i] refers to an element in the matrix which is located at k_{th} row and i_{th} column. Line 8 defines a flag to determine which state should be converted to the regression state. In other words, once the attackers are sent back to the state with the smallest number, this flag will make sure that the routes to other states are shut down. Linea 18 to 21 apply Equation (2-6) to obtain the steady state probability vector and display the final result.

CHAPTER 4

ALGORITHM FOR TOTAL REWARD WITH DEFENDING STRATEGY

Chapter 3 discusses two defending strategies and how to evaluate necessity of each state to be able to determine the vulnerability of an attack graph with respect to the probability of reaching the goal states. This chapter explores a method to calculate the total expected reward in the situation that the trapping strategy or the regression strategy is applied to defend the system. This chapter provides an expansion on the strategies presented in Chapter 3 by including other matrices, namely the rewards matrix, in the analysis of the quality of each policy.

Evaluation for the total expected reward with defending strategy is necessary because it denotes attributes such as the cost of defending each state. Other attributes, such as availability and quality-of-service, can also be included in the derivation of the reward values. If a system has a fixed amount of total cost to defend against attackers, this evaluation will figure out the optimal defending choice or minimize such cost.

4.1 Total Expected Reward with Trapping strategy

Lowering the total cost to reach a certain goal state might provide the attackers with a greater probability that they can compromise the system. On the other hand, it is much cheaper to construct the system for regular users since they can also get access to the goal state (e.g., the ability to legitimately access data) with lower cost (e.g., better quality of service or availability). The point is not only how to restrict attackers but also

to provide users with a certain level of quality-of-service. Thus, a compromise must be found that improves security while providing legitimate users with a system that is usable.

4.1.1 Algorithm for Total Reward with Trapping Strategy

Similar to the algorithm in Chapter 3, the first step is to remove a certain state from the attack graph and obtain a new probability matrix. Then the total expected reward matrix must be established and added to the system in order to compute total expected reward. This is because the reward matrix contains information about the amount of reward gained by each transition. The total expected reward can be calculated by summing up the immediate reward and the previous reward shown in Equation (2-13). Substituting the modified probability matrix and the reward matrix into the Equation (2-13) will yield the total expected reward. Note that the expected reward for the first transitions is 0 once the matrices are substituted into the equation, and the total expected reward in the next n runs can be obtained by continually updating v in Equation (2-13).

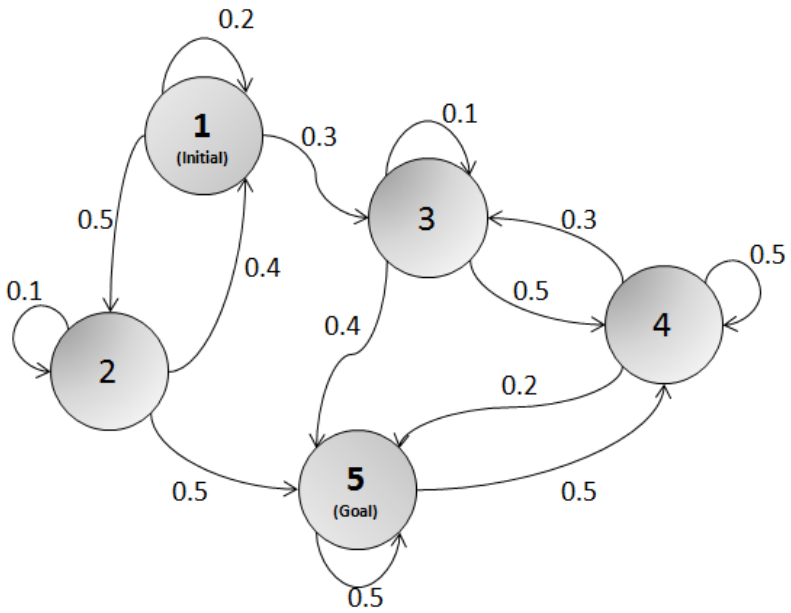

Figure 4.1: Markov process for an attack graph with one goal state.

A Markov process representation of an attack graph with only one goal state is shown in Figure 4.1. Assume state 1 is the initial state and state 5 is the goal state.

The probability matrix for the Markov process illustrated in Figure 4.1 is

$$P = \begin{bmatrix} 0.2 & 0.5 & 0.3 & 0 & 0 \\ 0.4 & 0.1 & 0 & 0 & 0.5 \\ 0 & 0 & 0.1 & 0.5 & 0.4 \\ 0 & 0 & 0.3 & 0.5 & 0.2 \\ 0 & 0 & 0 & 0.5 & 0.5 \end{bmatrix}. \quad (4\text{-}1)$$

The corresponding reward matrix for the Markov process shown in Figure 4.1 is

$$R = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 2 & 1 & 4 & 1 \\ 3 & 5 & 1 & 3 & 2 \\ 1 & 2 & 6 & 2 & 4 \\ 4 & 2 & 2 & 1 & 1 \end{bmatrix}. \quad (4\text{-}2)$$

The values in the reward matrix R can be zero, positive, or negative and the total expected reward can be interpreted as the reward gained by transitions or the total cost spent on getting access to the system. In this case, the total expected reward refers to the cost for attackers to enter (gain access or compromise) the system. The greater the cost, the more difficulty for the attacker to compromise the system. Thus, the defenders need to find a proper strategy to increase the total expected reward so that the attackers will not easily gain access to the system.

If neither the trapping nor regression strategies are applied, the total expected reward can be obtained and is shown in Table 4.1, where $v_i(n)$ represents the total expected reward starting from state i and n is the number of iteration. Note that Table 4.1 shows only the result for the first five transitions starting with each state. The result for more iterations will be shown later.

n	1	2	3	4	5
$v_1(n)$	2.1000	3.9900	5.8450	8.0007	10.2928
$v_2(n)$	1.5000	2.9900	5.0450	7.2475	9.5955
$v_3(n)$	2.4000	4.8400	7.3640	9.8964	12.4296
$v_4(n)$	3.6000	6.3200	8.8720	11.4072	13.9407
$v_5(n)$	1.0000	3.3000	5.8100	8.3410	10.8741

Table 4.1: Total expected reward with no trapping state.

$v_1(n)$ is the total expected reward starting with state 1 in the next n runs and increase with each transition because the total reward is the accumulation of the reward from the previous transitions.

Consider converting state 2 into a trapping state and modifying the probability matrix to reflect this change. Then apply Equation (2-14) to find the total expected reward for the next five transitions. The total expected reward for the Markov process in Figure 4.1 with state 2 converted to a trapping state is shown in Table 4.2. The data in Table 4.2 show that v_1 (the total expected reward starting with state 1) increases at a greater rate in the first five transitions when compared to v_1 in the Table 4.1. Thus converting state 2 into trapping state might have an effect on increasing the reward.

n	1	2	3	4	5
$v_1(n)$	2.1000	4.2400	6.4000	8.5892	10.7868
$v_2(n)$	2.0000	4.0000	6.0000	8.0000	10.0000
$v_3(n)$	2.4000	4.8400	7.3640	9.8964	12.4296
$v_4(n)$	3.6000	6.3200	8.8720	11.4072	13.9407
$v_5(n)$	1.0000	3.3000	5.8100	8.3410	10.8741

Table 4.2: Total expected reward with state 2 as a trapping state.

The process is repeated by computing the total expected reward when state 3 is converted into a trapping state. The total expected reward when state 3 is a trapping state is shown in Table 4.3. Compared to the result when state 2 was the trapping state, it indicates that the total expected reward (starting from state 1) is 7.9597 while it is 10.2928 and 10.7868 in the original case and the last case (state 2 as the trapping state),

which means it takes less cost to reach the goal state. So removing state 3 might be a bad choice since attackers can easily gain access to the goal state (state 5) at a low cost.

n	1	2	3	4	5
$v_1(n)$	2.1000	3.5700	4.9090	6.4203	7.9597
$v_2(n)$	2.0000	2.9900	4.8770	6.7513	8.5957
$v_3(n)$	1.0000	2.0000	3.0000	4.0000	5.0000
$v_4(n)$	3.6000	5.9000	8.8720	9.5250	11.1035
$v_5(n)$	1.0000	3.3000	5.8100	7.7050	9.6150

Table 4.3: Total expected reward with state 3 as the trapping state.

The process is repeated with state 4 as the trapping state and Table 4.4 shows the total expected reward for the first five transitions.

n	1	2	3	4	5
$v_1(n)$	2.1000	3.9900	5.6050	7.2847	9.0268
$v_2(n)$	2.0000	2.9900	4.6450	6.3315	8.1095
$v_3(n)$	1.0000	4.0400	5.8040	7.6804	9.6180
$v_4(n)$	2.0000	4.0000	6.0000	8.0000	10.0000
$v_5(n)$	1.0000	2.5000	4.2500	6.1250	8.0625

Table 4.4: Total expected reward with state 4 as the trapping state.

The total cost (starting from state 1) for state 4 being trapping state is 9.0268 after five runs, which is higher than the total reward 7.9597 in the case that state 3 being a trapping state, but it is lower than the total reward in the other two cases (10.2928 in the original case and 10.7868 in the case that state 2 as the trapping state). So this policy can be a secondary choice because it is more expensive than the case of state 3 being a trapping state, and cheaper than the original case and state 2 being a trapping state.

4.1.2 Analysis for Trapping Strategy Reward

This section deals with the total expected reward after first five transitions from the example in Section 4.1.1. It is not sufficient to identify which state has the most significant impact on the total cost by looking only at five transitions. The long-term

behavior of the system must be identified. Thus, a further extension of the number of iterations (transitions) is needed to elucidate the trend of removing each state. The graph for the total expected reward for the first 50 transitions is shown in Figure 4.2.

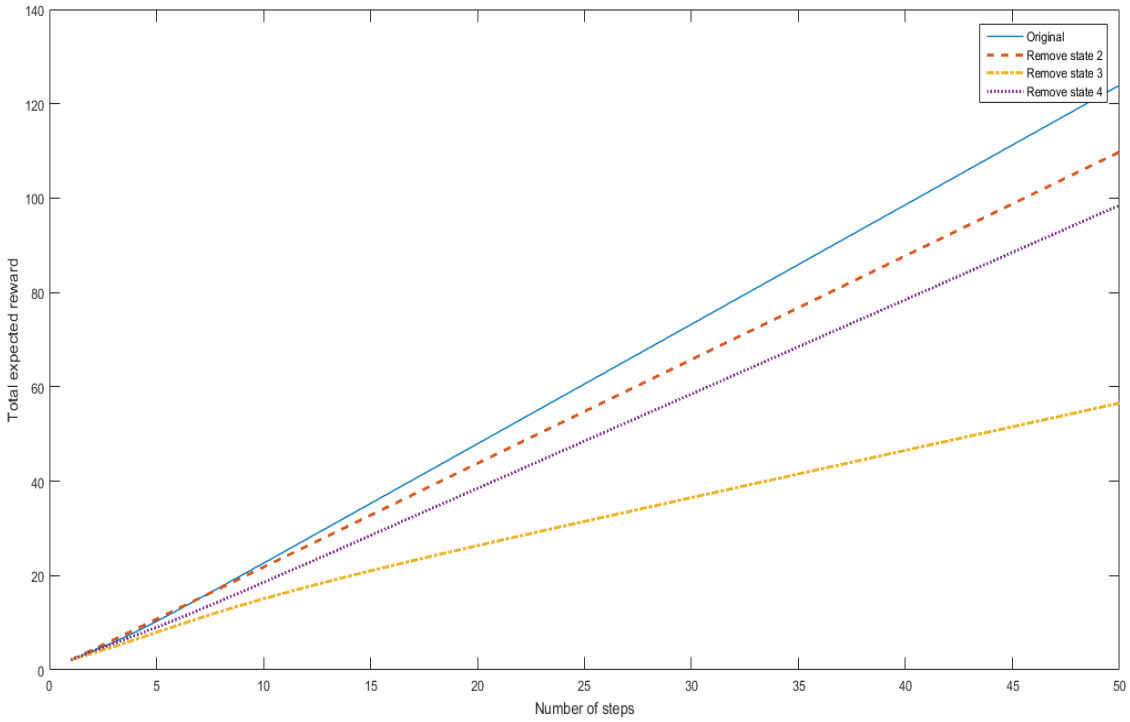

Figure 4.2: Total reward in the first 50 runs of removing each state

Each curve in the graph represents a different condition (e.g., no trapping states, or one of the states being the trapping state). The original reward curve (no trapping state) becomes larger than the curve for removing state 2 (making state 2 a trapping state) after 7 transitions. The curves for states 3 and 4 being the trapping state are also below the original curve, so the trapping strategy does indeed reduce the total cost and it will be easier for the attack to compromise the system. Moreover, the curve for removing state 3 is always less than the curve for removing state 4. This is because when state 3 is converted to be a trapping state, the attack is destined to end up in state 3 after running for many times. However, it could go to any state before being trapped in state 3, and this

is not linear at the beginning (resulting in the change of slope shown in Figure 4.2). This explains why the curve consists of linear and nonlinear parts.

4.1.3 Pseudocode for trapping strategy reward computation

The pseudocode for calculating the total reward with trapping strategy consists of matrix modification and value iteration. The matrix modification is the same as the pseudocode in Section 3.12, which is aimed at obtaining a new probability matrix. The value iteration calculates the total expected reward starting with each state by applying Equation (2-13).

```
1.  Input matrix P, R;
2.  Inputa integer k;
3.  Inputa integer n;
4.  N=size(P);
5.  if k=0 then
6.      P=P
7.  end if;
8.  if k>0 then
9.      for (i=0;i<N;;i++)
10. if i=k then
11.     P[k][i]=1;
12. else
13.     P[k][i]=0;
14. end if;
15. q=P*R
16. V_buffer=V
17. for i=1:1:n
18.      V[i]=q[i]+P[i,:]*V_buffer
19. end
20. Print V;
```

Figure 4.3: Pseudocode for computing the reward for the trapping state strategy.

Line 1 inputs the probability matrix and the reward matrix. Line 2 defines an integer k to determine which state is trapping state. Integer n in line 3 is the number of transitions to calculate the reward. Lines 5 to 7 address the case that there is no trapping state, while lines 8 to 14 modify the probability matrix for the trapping state. Lines 15 to

19 determine q and V from Equation (2-14) and apply this equation n times to compute the reward at each transition. Note that in line 18, the symbol ":" represents all the elements in the i_{th} row.

4.2 Total Expected Reward with Regression strategy

This section computes the total expected reward when applying regression strategy on each transient state in Figure 4.1. Comparing those total expected reward values obtained by the regression strategy on each state yields an optimal choice to increase the cost to get into the goal states and thus enhancing the difficulty for the attackers to compromise the system.

4.2.1 Algorithm for computing the total reward using the regression strategy

Each state should be converted into a regression state and the total expected reward computed to evaluate how important the state is. The primary difference between the trapping strategy and the regression strategy is how the probability matrix is modified.

Consider the Markov process of the attack graph from Figure 4.1 and apply the regression strategy to state 2. The resulting total expected reward is shown in Table 4.5, where $v_i(n)$ is the total expected reward whose initial state is i.

n	1	2	3	4	5
$v_1(n)$	2.1000	4.2400	6.4500	8.7192	11.0378
$v_2(n)$	2.0000	4.1000	6.2400	8.4500	10.7192
$v_3(n)$	2.4000	4.8400	7.3640	9.8964	12.4296
$v_4(n)$	3.6000	6.3200	8.8720	11.4072	13.9407
$v_5(n)$	1.0000	3.3000	5.8100	8.3410	10.8741

Table 4.5: Total expected reward with state 2 as the regression state.

The total expected reward in Table 4.5 when starting in state 1 is slightly higher than the reward for the original case shown in Table 4.1 (no regression states). They are very close to each other and an extended graph is necessary for illuminating the trend of increasing reward.

The process is repeated by selecting state 3 as the regression state and the total expected reward for the first five transitions when converting state 3 into a regression state is shown in Table 4.6.

n	1	2	3	4	5
$v_1(n)$	2.1000	4.1700	5.9590	8.0013	10.0606
$v_2(n)$	2.0000	2.9900	5.1170	7.3453	9.6700
$v_3(n)$	2.4000	5.1000	7.1700	8.9590	11.0013
$v_4(n)$	3.6000	6.5000	9.0400	11.4510	13.7072
$v_5(n)$	1.0000	3.3000	5.9000	8.4700	10.9605

Table 4.6: Total expected reward with state 3 as the regression state.

Table 4.6 suggests that the total reward is lower than when state 2 is the regression state (see Table 4.5). State 1 and state 2 are no longer transient states since state 3 can make transition to state 1.

The process is repeated by selecting state 4 as the regression state and the total expected reward for the first five transitions is shown in Table 4.7. State 4 yields the highest reward among four cases.

n	1	2	3	4	5
$v_1(n)$	2.1000	3.9900	6.2050	8.8647	11.6988
$v_2(n)$	2.0000	2.9900	5.6450	8.2715	11.2455
$v_3(n)$	2.4000	6.0400	9.0040	12.3004	15.4300
$v_4(n)$	3.6000	8.4000	12.0400	15.0040	18.3004
$v_5(n)$	1.0000	4.5000	7.4500	10.7450	13.8745

Table 4.7: Total expected reward with state 4 as the regression state.

4.2.2 Analysis for the Regression Strategy Reward

A further extension on the number of transitions is required in order to determine the trend of the total reward to reach the goal state and this is shown in Figure 4.4.

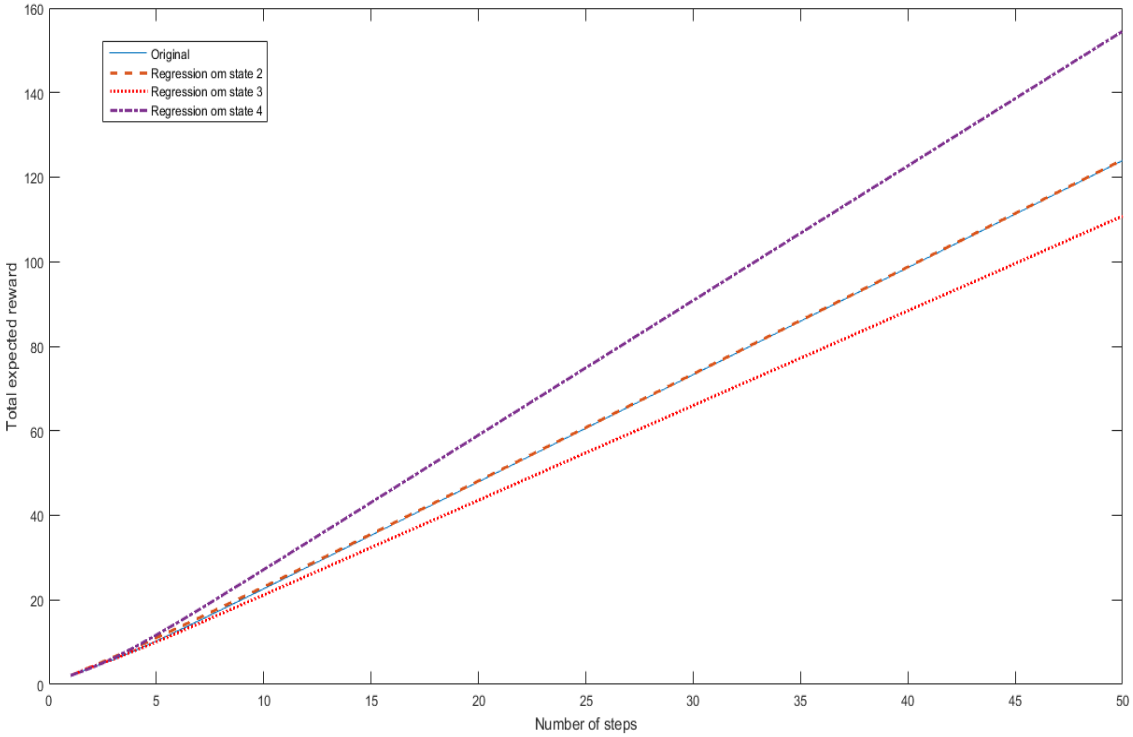

Figure 4.4: Total reward in the first 50 transitions of using regression on each state.

Figure 4.4 indicates that when state 2 is a regression strategy, v_1 (The total expected reward with state 1 as the starting state) in the Table 4.5 is almost the same as v_1 in the original case (no trapping state, see Table 4.1). This is because in Figure 4.1, state 1 and state 2 are transient states, which means the system find it almost impossible to stay in one of these two states after a number of transitions. So the process will end up in states 3, 4, or 5 eventually whether state 2 is a regression state or not because the system can never move back to state 1 or state 2 after a number of transitions. In conclusion, regression strategy on state 2 almost has no effect on the total expected reward and thus the regression on state 2 curve nearly overlaps the original reward curve.

This conclusion can also be examined from Table 4.1 and Table 4.5. The total expected reward (v_1) in these two tables is very close to each other.

The curve for regression on state 3 is below all the curves after a number of transitions. So using regression strategy on state 3 might not be a good strategy for improving system security since the attackers can break through the system security at such a low cost. The curve for regression on state 4 is above any other curve. It has the most significant cost and thus sets up more barriers against attack on the system.

4.2.3 Pseudocode for Calculation of the Regression Strategy Reward

The Pseudocode for computing the total expected reward when applying regression strategy on each state is presented in Figure 4.5.

```
1.  Input matrix P, R;
2.  Inputa integer k;
3.  Inputa integer n;
4.  N=size(P);
5.  if k=0 then
6.      P=P
7.  end if;
8.  if k>0 then
9.      for (i=0;i<N;;i++)
10.         if i≠k and P[k][i]≠0 then
11.             P[k][i]=1;
12.             break;
13.         else
14.             P[k][i]=0;
15.         end if;
16.      end;
17. q=P*R
18. V_buffer=V
19. for i=1:1:n
20.      V[i]=q[i]+P[i,:]*V_buffer
21. end
22. Print V;
```

Figure 4.5: Pseudocode for computing the reward for the regression strategy.

The pseudocode for computing the reward for the regression strategy is similar to the pseudocode for computing the reward for the trapping strategy (see Section 4.1.3). Modification of the probability matrix is the first step since the attack graph for regression is slightly different from the trapping case. The trapping strategy forces the state to remain at a single state so the element on the k_{th} row,k_{th} column (the probability to stay in state k) should be 1 and the rest of others on k_{th} row (the probability to enter other states except state k) should be 0. However, the matrix modification of the regression strategy is a little bit different from the trapping strategy modification. Since the policy applied here is for the regression state to move back to the last reachable state with the smallest number, lines 10 to 16 set the probability to get back to the recently reachable state as 1. The other transitions leaving the state on the k_{th} row (the probability to get into other state from state k) are set to 0 as soon as any one of the elements in k_{th} row is 1 (see line 11). Lines 17 to 21 apply the value iteration process to calculate the final total expected reward according to the modified probability matrix and unchanged reward matrix.

CHAPTER 5

IMPLEMENTATION OF DETERMINING NECESSITY

This chapter provides implementation of the algorithm to determine the necessity of each state in an attack graph by converting each state into a trapping state or regression state, measuring the importance of each state by comparing the probability of entering certain goal states (See Chapter 3).

The implementation is built in a Unix system by using the C programming language. The algorithm involves a huge amount of computational work and requires a high-performance computing approach. Thus, a heterogeneous cluster consisting of many Intel Phis and CPUs is employed to handle the massive number of computations required by the algorithms developed in this thesis. The runtime of the program on the heterogeneous cluster (CPUs and Intel Phis) is compared to the runtime of the program running on the same cluster, but using just the CPUs. This yields a comparison between the heterogeneous implementation and the CPU-only implementation and thus improvement of the program performance can be quantified.

5.1 Necessity for Using Heterogeneous Cluster

Both of the parallel algorithms for determining the state necessity and computing the total expected reward involve iterating matrix multiplication (see Chapters 3 and 4). For the algorithm of determining state necessity, Equation (2-6) shows that the steady

state probability vector has to be multiplied by the probability matrix to compute the steady state probability vector for the next iteration. This multiplication equation is

$$[a_{11} \quad \dots \quad a_{1n}] \times \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix} = [c_{11} \quad \dots \quad c_{1n}]. \tag{5-1}$$

Based on Equation (5-1), a set of equations can be derived and these are

$$\begin{gathered} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + \cdots + a_{1n}b_{n1} = c_{11}, \\ a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} + \cdots + a_{1n}b_{n2} = c_{12}, \\ \vdots \\ a_{11}b_{1n} + a_{12}b_{2n} + a_{13}b_{3n} + \cdots + a_{1n}b_{nn} = c_{1n}. \end{gathered} \tag{5-2}$$

For each equation in the set shown in Equation (5-2), the row elements in the state probability vector ($a_{11} \quad \dots \quad a_{1n}$) should multiply the column elements in the probability matrix ($b_{11} \quad \dots \quad b_{nn}$) in order to obtain a single element in the new state probability vector ($c_{11} \quad \dots \quad c_{1n}$). For example, in order to obtain c_{11}, the state transition vector a should multiply the first column elements in the probability matrix b one by one and this procedure involves n computations. Moreover, each product from the previous multiplication step has to be summed (accumulated) to obtain the value of c_{11} and this procedure takes $n-1$ addition operations. Therefore, a single element c_{11} will take $n+n-1=2n-1$ arithmetic (multiplication or addition) operations. Here multiplication and addition are treated as requiring the same amount of computational effort, however, this work is easily extendable to include cases where they require different computational workloads. Since the state probability transition matrix is a matrix with n dimensions, Equation (5-1) will take $n \times (2n-1) = 2n^2 - n$ arithmetic operations to obtain the new steady state probability matrix.

Moreover, Equation (5-1) has to be iterated many times until the steady state probability vector converges to the steady state probabilities of the steady state Markov

process. Assume that $m$iterations are required. Then it will require$(2n^2 - n)m$ arithmetic computations to obtain the final steady state probability vector. Thus, the computational workload with respect to arithmetic operations grows quickly with the size of the number of states, *n*, O(n^2); and with the number of iterations, *m*, O(*m*). For realistic systems and attack graphs the *n* and *m* values are too large for computation in a traditional desktop or laptop environment. Further, note that as the number of states, *n*, grows, the number of iterations, *m*, must also grow to ensure that the steady state probability vector converges. This further increases the computational workload growth rate.

Such intensively massive work requires a great amount of resources. Although the CPU in the heterogeneous cluster has multiple cores and it is good at logic control and serial computing, it is not efficient for dealing with such large data sets. However, the Intel Phi has even more cores than the CPU, though there are less processing units in those cores. Intel Phi shares some characteristics with the GPU and is capable of finishing independent and simple work involving large amounts of data. However, communication issues also exist in Intel Phi. For example, a thread cannot execute the next task until all the other threads complete their work. So a great amount of time and resources are consumed and wasted on waiting. Therefore, the performance of Intel Phi is affected by the number of cores and communication, and sometimes cannot achieve expected performance. In order to lower the cost spent on communication, one can reduce the number of threads being used and optimize algorithms by combing parallelism with serialization.

This heterogeneous cluster is consisting of 12 nodes. Each node has got two CPUs and two Intel Phis. The hardware specification for each node is shown in Figure 5.1.

model type: CPU
vendor_id: GenuineIntel
model name: Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
cpu base frequency: 2399.969 MHz
cpu max frequency: 3200.000 MHz
cpu cores: 16
cache size: 15360 KB
address sizes: 46 bits physical, 48 bits virtual

model type: Coprocessor
model name: Intel(R) Xeon Phi(TM) Coprocessor 7120P
processor base frequency: 1.238 GHz
processor max frequency: 1.333 GHz
processor cores: 61
cache size: 30.5 MB L2
max memory size: 16MB

Figure 5.1: Hardware specification for a single node in the heterogeneous cluster

C language is used in the implementation of both algorithms. OpenMP is used to achieve parallelism on Intel Phi. The code should be compiled by Intel compiler and the program will be executed on either CPU or heterogeneous cluster by using different compiling command shown in Table 5.1. Note that the programming code is saved as file 'test.cpp' and it will be compiled into a file 'test' which contains computer language.

Compiling command	Function
icc –fopen test.cpp –o test	Run the program on CPU only
icc –fopen test.cpp –o test –qno-offload	Run the program by heterogeneous cluster

Table 5.1: Compiling command for different device target

Slurm is a very useful management tool to schedule jobs for clusters and 'sbatch' is a Slurm command for submitting script files to Slurm so that the program can be executed. The script file also known as shell contains configuration and information for the program execution. The shell file that will be used in this thesis is shown in Figure 5.2.

```
#SBATCH --job-name=helloOMP
#SBATCH --error=job.%J.err
#SBATCH --output=job.%J.out
#SBATCH --ntasks=16
#SBATCH --cpus-per-task=16
#SBATCH --threads-per-core=2
#SBATCH --time=24:00:00
#SBATCH --mem-per-cpu=3840
module load icc
./test
```

Figure 5.2: Configuration for the execution of the program

The line contains ‘cpu-per-task=16’ defines that 16 CPUs are used for a single tasks while ‘ntasks=16’ indicates that there are totally 16 tasks, which means 256 threads are needed. Each core has got two threads according to line ‘threads-per-core=2’, so there are 32 threads for each node. Therefore, eight nodes will be used to engage process. OpenMP will automatically assign tasks to multiple nodes once configuration is done.

5.2 Program for Determining State Necessity

The program for the state necessity determination algorithm consists of two subroutines and a main program. This section describes the functionality of each program and explains how each is built. The code for each program is shown separately in this section in order to achieve a better observation and explanation.

5.2.1 Subroutine for Generating Matrices

The first subroutine is aimed at generating an initial steady state probability vector and a probability matrix as the input for the main program. This procedure can also be interpreted as a method to generate a random attack graph based on the Markov process model and the code is shown in Figure 5.3.

```
#include <stdio.h>
#include <stdlib.h> // Head file for rand()
#include <omp.h> // Head file for OpenMP
#include <time.h>// Head file for Clock()
#include "offload.h"
#define N 10
double ProbabilityMatrix[N][N];
double StateMatrix[1][N];
double Result[1][N];
void AssignValues()
{
    double sum = 0.0;
    for (int i = 0; i < N; i++)
    {
            Result[0][i] = 0;
    }
    for (int i = 0; i < N; i++)
    {
            StateMatrix[0][i] = 0;
    }
    StateMatrix[0][0] = 1;
    for (int i = 0; i < N; i++)
    {
            for (int j = 0; j < N; j++)
            {
                    srand(i + j);
                    ProbabilityMatrix[i][j] =((double)rand()) / RAND_MAX;
            }
    }
    for (int i = 0; i < N; i++)
    {
            for (int j = 0; j < N; j++)
            {
                    sum = sum + ProbabilityMatrix[i][j];
            }
            for (int k = 0; k < N; k++)
            {
                    ProbabilityMatrix[i][k] = ProbabilityMatrix[i][k]/sum;
            }
            sum = 0.0;}
    printf("Steady state probability vector:\n");
    for (int i = 0; i < N; i++)
    {
            printf("%f ", StateMatrix[0][i]);
    }
    printf("Probability matrix:\n");
    for (int i = 0; i < N; i++)
    {
            for (int j = 0; j < N; j++)
            {
                    printf("%f ", ProbabilityMatrix[i][j]);
            }
            printf("\n");}}
```

Figure 5.3: Code for Generating Random Input Matrices.

Line 1 to line 5 defines header files for the program. The headerfile <Time.h> includes the Clock() function, which returns the number of clock ticks that have passed since the program was launched. This function is able to record the time spent on the program execution and this time (with some basic arithmetic to convert clock ticks into seconds) is very important for measuring the performance of the program. Line 6 determines the number of states in the attack graph and this number is also closely related to the amount of workload (see Section 5.1). In this case, the number of states equals to 10 in order to obtain a better observation on the generated matrices. Line 7 to line 9 defines target matrices that are going to be generated. Matrix Result is a buffer (of type double) to store the sum of products obtained by matrix multiplication. Line 13 to line 16 resets the buffer Result, while line 17 to line 20 resets the steady state probability vector. This implementation assumes that the initial state is the very first state of the attack graph so the first element in the initial steady state probability vector should be 1 and the other elements should be 0. Line 21 assigns 1 to the first element of the steady state probability vector. Line 22 to 29 generates a random probability matrix and all the elements in this matrix are between 0 and 1. However, for each row the sum of row elements must be 1 (see Chapter 2). Thus, line 32 to line 35 compute the sum of each row, and line 36 to line 39 makes sure that each element in the row is divided by the sum of the row elements (reduce each element by a factor equal to the original sum) so that in new matrix the sum of each row equals to 1. Line 42 to line 53 displays the steady state probability vector and probability matrix.

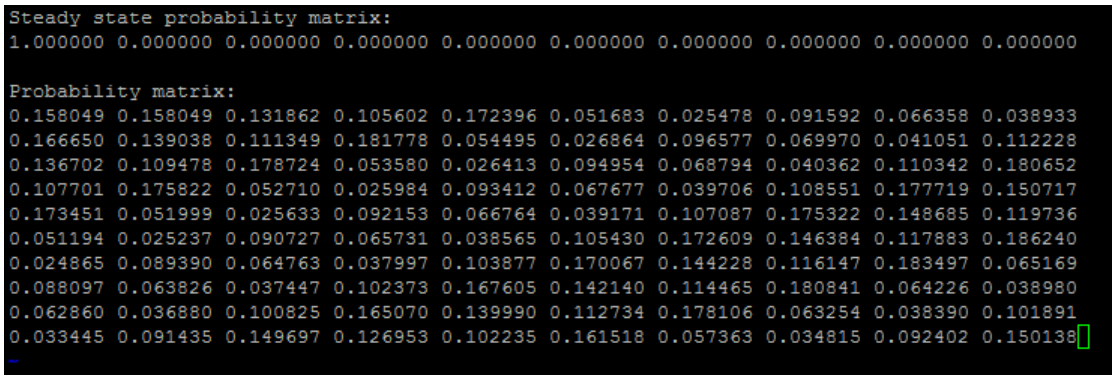

Figure 5.4: The result for the random matrices generating program.

The random steady state probability vector and transition probability matrix are shown in Figure 5.4. The first element in the initial steady state probability vector is 1and other elements are 0, which indicates the first state is the initial state. As for the transition probability matrix, all the elements are randomly distributed and the sum of each row equals to 1.

5.2.2 Subroutine for Matrix Modification

This subroutine is able to modify the transition probability matrix (also referred to as the probability matrix) using different strategies by replacing a certain row in the probability matrix with a new row vector. Either of the two strategies (trapping or regression) can be used to modify the probability matrix.

```
1.void TrappingModification()
2.{
3.      int trapping_state = 2;
4.      for (int i = 0; i < N; i++)
5.      {
6.              ProbabilityMatrix[trapping_state][i] = 0;
7.      }
8.      ProbabilityMatrix[trapping_state][trapping_state] = 1;
9.}
```

Figure 5.5: Program for trapping state modification.

The code for modifying probability matrix by using trapping strategy is shown in Figure 5.5. This program is aimed at converting one of the states in attack graph to a trapping state, modifying the probability matrix that has been already generated by the program in Section 5.1.1. Line 3 determines a specific state as the trapping state. Line 4 to line 8 generates a new row vector whose element on the k_{th} row and k_{th} column is 1 while other elements are 0, taking the place of old row vector.

```
1.void RegressionModification()
2.{
3.      int regression_state = 2;
4.      int stop_flag = 0;
5.      for (int i = 0; i < N; i++)
6.      {
7.  if (stop_flag == 0 && ProbabilityMatrix[regression_state][i] != 0 && i != regression_state)
8.              {
9.                      ProbabilityMatrix[regression_state][i] = 1;
10.                     stop_flag = 1;
11.             continue;
12.             }
13.             else
14.             {
15.                     ProbabilityMatrix[regression_state][i] = 0;
16.             }
17.     }
18.}
```

Figure 5.6: Program for regression state modification.

The code for modifying the probability matrix by using the regression strategy is shown in Figure 5.6. Line 3 determines a state as regression state. Line 4 defines a register stop_flag. Line 5 to line 18 set the probability to return to the recently reachable state as 1 and set the other elements on the k_{th} row (the probability to get into other state from state k) to 0 as soon as any one of the elements in the k_{th} row equal 1.

5.2.3 Main Program for the Necessity Determining Algorithm

The main program consists of two subroutines (see Sections 5.2.1 and 5.2.2) and a program for iterating matrix multiplication. The generated and modified steady state

probability matrix and probability matrix will be used in the iteration of matrix multiplication program and obtain the final steady state probability matrix. The main program is shown in Figure 5.7.

```
1.void main()
2.{
3.      AssignValues();
4.      TrappingModification();
5.      clock_t t1 = clock();
6.#pragma offload target(mic)
7.      {
8.              for (int iteration = 0; iteration < 2 * N; iteration++)
9.              {
10.#pragma omp parallel for shared(ProbabilityMatrix,StateMatrix,Result) private(i,j,k)
11.                     for (int j = 0; j < N; j++)
12.                     {
13.                             for (int k = 0; k < N; k++)
14.                             {
15.       Result[0][j] = Result[0][j] + StateMatrix[0][k] * ProbabilityMatrix[k][j];
16.                             }
17.                     }
18.                     for (int i = 0; i < N; i++)
19.                     {
20.                             StateMatrix[0][i] = Result[0][i];
21.                             Result[0][i] = 0;
22.                     }
23.             }
24.     }
25.     clock_t t2 = clock(); //End counting
26.     printf("Time = %f\n", (double)(t2 - t1) / CLOCKS_PER_SEC);
```

Figure 5.7: Main program for determine state necessity algorithm.

Line 3 randomly generates the initial steady state probability matrix and probability matrix (see section 5.2.1). Line 4 modifies the probability matrix by using either trapping strategy or regression strategy. Line 5 returns the number of clock ticks from the system and this is the starting time of the execution. In line 6, OpenMP clause "#pragma offload target(mic)" forces the CPU to offload input matrices and matrix multiplication program to any available Intel Phis and let the Phis deal with the iterations. The content in braces following this clause is what will be executed on Intel Phi. Line 10 gives another OpenMP clause "#pragma omp for" to coerce the subsequent code to be

executed in parallel. Thus, Lines 10 to 17 contain the parallel algorithm for matrix multiplication. Line 18 to 23 ensures that the steady state probability matrix is updated at the end of each iteration. Line 25 returns clock ticks at the end of the program. Line 26 computes time spent on the program by calculating the interval ticks between two clock ticks and converts the tick interval ticks to a real time value. The real time spent on the program is an essential parameter to measure the performance of the program.

5.3 Performance of the Heterogamous Clusters on the program

This section presents the results of running the complete program (consisting of subroutines and the main program) for determining state necessity on the heterogeneous cluster using (1) CPUs and Intel Phis and (2) just CPUs separately. The runtimes are recorded and used to compare the CPU-only performance with the CPU and Intel Phi option on the heterogeneous cluster performance. The complete code for the determining state necessity with timing information reported is shown in Figure 5.8.

The program in Figure 5.6 is also using some OpenMP clauses beginning with "#pragma omp" to achieve parallelism. More detail about combination use of Intel Phi and OpenMP is in [10].

The command "icc -fopenmp test.cpp -o test" was used to compile the parallel program which will be running on both CPU and Phi. In the case that the program is only running on CPU, parameter "-qno-offload" should be added to the compiling command so that "#pragma offload target(mic)" can be neglected by the compiler and the program will not be executed on the Intel Phi. Note that OpenMP clause "#pragma omp parallel for" should also be deleted so that it is a completely serial program on CPU.

```
#include <stdio.h>
#include <stdlib.h> // Head file for rand()
#include <omp.h> // Head file for OpenMP
#include <time.h>// Head file for Clock()
#include "offload.h"
#define N 1000
double ProbabilityMatrix[N][N];
double StateMatrix[1][N];
double Result[1][N]; int i, j, k, iteration;
void AssignValues()
{       double sum = 0.0;
        for (int i = 0; i < N; i++){
                Result[0][i] = 0;}
        for (int i = 0; i < N; i++){
                StateMatrix[0][i] = 0;}
        StateMatrix[0][0] = 1;
        for (int i = 0; i < N; i++){
                for (int j = 0; j < N; j++){
                        srand(i + j);
                        ProbabilityMatrix[i][j] = ((double)rand()) / RAND_MAX;}}
        for (int i = 0; i < N; i++){
                for (int j = 0; j < N; j++){
                        sum = sum + ProbabilityMatrix[i][j];}
                for (int k = 0; k < N; k++){
                        ProbabilityMatrix[i][k] = ProbabilityMatrix[i][k] / sum;}
                sum = 0.0;}}
void TrappingModification()
{       int trapping_state = 2;
        for (int i = 0; i < N; i++){
                ProbabilityMatrix[trapping_state][i] = 0;}
        ProbabilityMatrix[trapping_state][trapping_state] = 1;}
void main()
{       AssignValues();
        TrappingModification();
        clock_t t1 = clock();
#pragma offload target(mic)
        {               for (int iteration = 0; iteration < 2 * N; iteration++){
#pragma omp parallel for shared(ProbabilityMatrix,StateMatrix,Result) private(i,j,k)
                        for (int j = 0; j < N; j++){
                                for (int k = 0; k < N; k++){
          Result[0][j] = Result[0][j] + StateMatrix[0][k] * ProbabilityMatrix[k][j];}}
                        for (int i = 0; i < N; i++){
                                StateMatrix[0][i] = Result[0][i];
                                Result[0][i] = 0;}}}
        clock_t t2 = clock(); //End counting
        printf("Time = %f\n", (double)(t2 - t1) / CLOCKS_PER_SEC);}
```

Figure 5.8: The complete code for determining state necessity.

The program is executed on (1) CPUs and Intel Phis and (2) CPUs only for a variety of input matrix sizes and the runtimes recorded for each experiment. The runtime performance for each input size is shown in Table 5.2. The term "X" in Table 5.2

indicates that the time spent on the program is over 24 hours and the run was terminated without finishing because it was taking excessive time. Note that size means the number of states in the attack graph.

Note that in these experiments, the number of iterations has already been reduced to as twice as the number of states in order to shorten the execution time. However, it is not sufficient to just find steady state probability vector and this execution time problem becomes more serious as the matrix size grows larger. Reducing the number of iterations is able to enhance speed for analyzing the system, but its accuracy cannot be guaranteed due to the lack of iterations.

Size (number of rows)	500	1000	2000	3000	4000	5000	6000
CPU time(s)	0.08	0.72	9.82	40.36	96.76	190.44	439.12
PHI&CPU time(s)	1.95	4.02	15.6	46.53	94.42	175.70	262.70
Speedup	0.04	0.18	0.63	0.87	1.02	1.08	1.67
Size (number of rows)	7000	8000	9000	10000	15000	20000	30000
CPU time(s)	518.51	768.13	1106.42	1826.89	7601.37	19417.51	X
PHI&CPU time(s)	374.71	591.63	833.72	1038.59	2810.14	5567.67	16659.13
Speedup	1.38	1.30	1.33	1.76	2.70	3.49	X

Table 5.2: Time spent on the state necessity determination program.

The Speedup is computed by dividing the time required when running on CPUs only by the time required running on CPUs and Intel Phis (Speedup = $Time_{CPU}$ / $Time_{CPU\&Phi}$). The performance of Intel Phi is as good as the CPU when the matrix size is below 4000 due to the communication and the competition issues existing in the parallel structure. A single thread might not be able to execute the next task and needs to wait until all of the other threads have completed their tasks. Moreover, the data in the

program sometimes inevitably has competition against each other. Thus, there is a huge cost in the parallel algorithm and the performance of the Intel Phi & CPU is not as good as the CPU only implementation for small input sizes. However, this situation is reversed when the matrix grows large, more than 4000 elements per row. For these large matrices, the majority of the time is spent on the parallel computation and the cost of the communication and competition issues is becoming more negligible. Therefore, the performance of Intel Phi will be better than the CPU only as the matrix size grows larger. According to Table 5.2, the performance of Intel Phi is better than the performance of the CPU only when the matrix size is above 4000. Figure 5.9 plots a graph for the performance of these two devices (term "X" in Table 5.2 are neglected and not shown in the graphs in Figures 5.9 and 5.10).

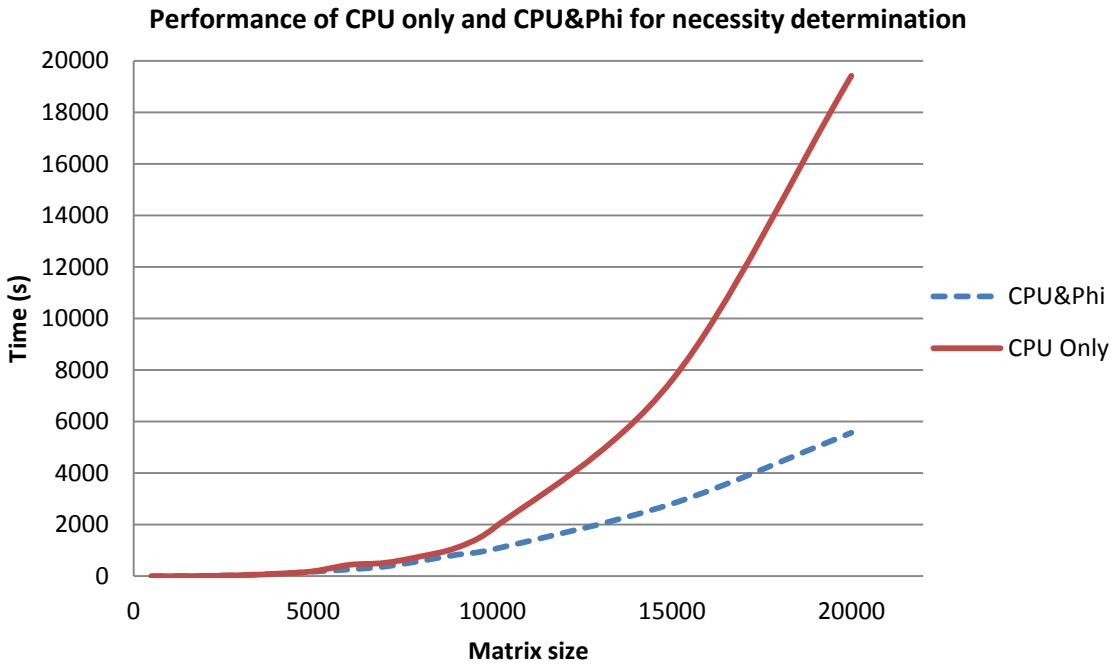

Figure 5.7: Performance of CPU&Phi and CPU only options for necessity determination.

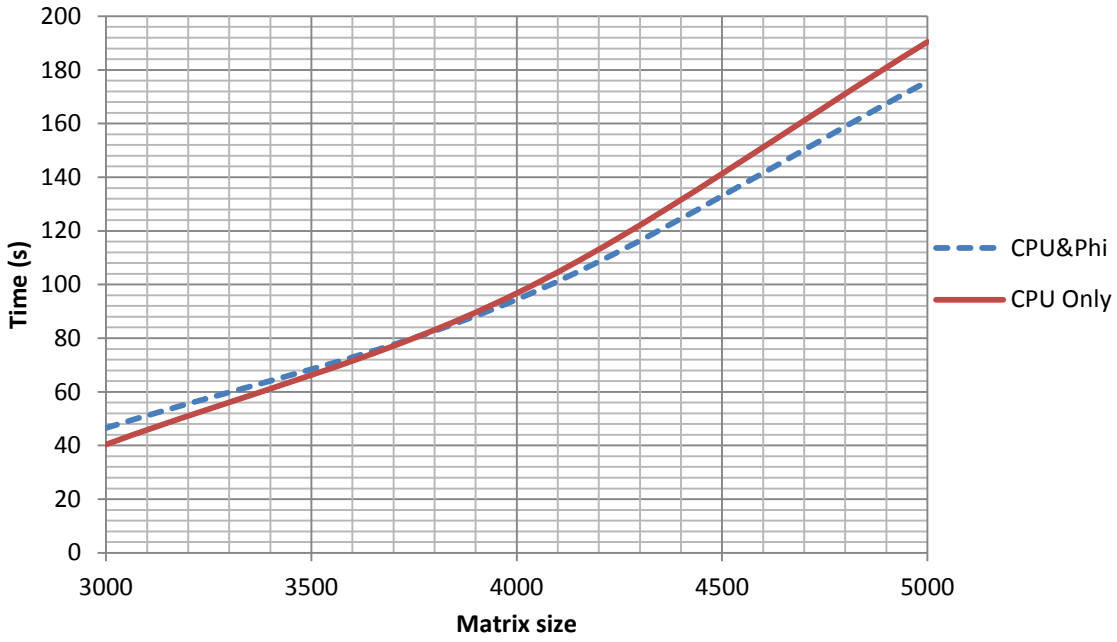

Figure 5.8: Zoomed-in view of Figure 5.7 for matrix size of 3000 to 5000.

CHAPTER 6

IMPLEMENTATION OF COMPUTING REWARD

This chapter presents the implementation of the algorithm for calculating total expected reward using the heterogeneous cluster. This algorithm is capable of computing the total expected reward for either the trapping strategy or the regression strategy.

6.1 Program for Computing Total Expected Reward

The purpose of this program is to compute the total expected reward according to Equation (2-14). It also includes the subroutines for generating a random probability matrix (see Section 5.2). The output is generated in a similar fashion as that for the program described in Chapter 5, however, the program presented in this chapter will include the reward matrix in the calculations to derive the total expected reward. The number of clock ticks for the program will be recorded and used to compute the runtime of the program running on the heterogeneous cluster using (1) CPUs and Intel Phis, and (2) CPUs only.

6.1.1 Subroutine for Input Matrices

This program uses subroutines to randomly generate the probability matrix and the state transition probability matrix and then modifies the generated matrices by using either the trapping or regression strategy. The subroutine for the matrix modification is the same as the subroutine described in Section 5.2.2. However, the subroutine for

generating the input matrices is different from the subroutine in Section 5.2.1 and it is shown in Figure 6.1.

```
1.void AssignValues()
2.{
3.      double sum = 0.0;
4.      for (int i = 0; i < N; i++)
5.      {
6.              for (int j = 0; j < N; j++)
7.              {
8.                      srand(i + j);
9.                      ProbabilityMatrix[i][j] = ((double)rand()) / RAND_MAX;
10.             }
11.     }
12.     for (int i = 0; i < N; i++)
13.     {
14.             for (int j = 0; j < N; j++)
15.             {
16.                     sum = sum + ProbabilityMatrix[i][j];
17.             }
18.             for (int k = 0; k < N; k++)
19.             {
20.                     ProbabilityMatrix[i][k] = ProbabilityMatrix[i][k] / sum;
21.             }
22.             sum = 0.0;
23.     }
24.     for (int i = 0; i < N; i++)
25.     {
26.             for (int j = 0; j < N; j++)
27.             {
28.                     srand(i + j);
29.                     RewardMatrix[i][j] = rand() % 10;
30.             }
31.     }
32.}
```

Figure 6.1: Subroutine for generating random input matrices.

Line 3 to line 23 creates a random probability matrix which is the same as the previous matrix generating subprogram (see Section 5.2.1). However, there is no code for generating a steady state probability vector since the steady state probability vector no longer exists in the algorithm to compute the total expected reward. Line 24 to line 31 generates a random reward matrix whose element values are between 0 to 9. These values can be any numbers. In this case they are assumed to be between 0 to 9.

6.1.2 Main Program for Reward Computing Algorithm

The main program for computing the total expected reward is shown in Figure 6.2.

```
1.void main()
2.{
3.      AssignValues();
4.      TrappingModification();
5.      clock_t t1 = clock(); //Start counting;
6.      for (i = 0; i < N; i++)
7.      {
8.              for (j = 0; j < N; j++)
9.              {
10.     ImmediateBuffer = ImmediateBuffer + ProbabilityMatrix[i][j] * RewardMatrix[i][j];
11.             }
12.             ImmediateReward[i][0] = ImmediateBuffer;
13.             ImmediateBuffer = 0.00000;
14.     }
15.#pragma offload target(mic)
16.     {
17.             for (int iteration = 0; iteration < 10000; iteration++)
18.             {
19.                     for (int k = 0; k < N; k++)
20.                     {
21.                             V_buffer[k][0] = RewardBuffer[k][0];
22.                     }
23.#pragma omp parallel for shared(V_buffer,ProbabilityMatrix,RewardBuffer,Result) private(i,j)
24.                     for (int i = 0; i < N; i++)
25.                     {
26.                             for (int j = 0; j < N; j++)
27.                             {
28.             Result = Result + ProbabilityMatrix[i][j] * V_buffer[j][0];
29.                             }
30.                             RewardBuffer[i][0] = Result + ImmediateReward[i][0];
31.                             Result = 0.0;
32.                     }
33.             }
34.     }
35.     clock_t t2 = clock(); //Start counting;
36.     printf("First Time = %f\n", (double)(t2 - t1) / CLOCKS_PER_SEC);
37.}
```

Figure 6.2: Main program for Calculating the expected reward.

Line 3 obtains input matrices and line 4 modifies the probability matrix by applying one of the two strategies. Line 5 retunes the number of system ticks and this is the very beginning of the time interval which will be used to evaluate the performance of the heterogeneous cluster. Line 6 to line 14 computes immediate reward q according to Equation (2-12). Line 15 copies data and program from the host (CPU) and transfers

them to the Intel Phi, assigning the Phis to perform the computation. Line 17 to line 34 keeps iterating Equation (2-13) and updating total expected reward $v_i(n)$. Line 35 obtains the number of clock ticks and this is the end of the time interval. Line 36 calculates the difference between two clock ticks and converts it to the real time spent on the program in order to measure the performance of the heterogeneous clusters.

6.2 Performance of the Heterogamous Clusters on the program

This section explores the performance of the CPU and Intel Phi for the reward computation program. Two programs to implement this algorithm are described. The first program deals with the situation when the number of states in the Markov process is small. As the Markov process grows, the large number of states will inevitably result in the shortage of memory and the second program adopts memory allocation to properly use memory.

6.2.1 Performance Reward Computing Program without Memory Allocation

The complete code for the program is shown in Figure 6.3. This section compares the performance of heterogeneous cluster when using (1) CPUs and Intel Phis and (2) CPUs only.

```
#include <stdio.h>
#include <stdlib.h> // Head file for rand()
#include <omp.h> // Head file for OpenMP
#include <time.h>// Head file for Clock()
#include "offload.h"
#define N 1000
double ProbabilityMatrix[N][N]; double RewardMatrix[N][N];
double ImmediateReward[N][1] = { { 0.0 } }; double RewardBuffer[N][1] = { { 0.0 } };
double TotalReward[N][1] = { { 0.0 } }; double V_buffer[N][1] = { { 0.0 } };
double Result = 0.0; double ImmediateBuffer = 0.0;
void AssignValues()
{       double sum = 0.0;
        for (int i = 0; i < N; i++){
                for (int j = 0; j < N; j++){
                        srand(i + j);
                        ProbabilityMatrix[i][j] = ((double)rand()) / RAND_MAX;}
        for (int i = 0; i < N; i++)
        {     for (int j = 0; j < N; j++){
                sum = sum + ProbabilityMatrix[i][j];}
                for (int k = 0; k < N; k++){
                        ProbabilityMatrix[i][k] = ProbabilityMatrix[i][k] / sum;}
                sum = 0.0;}
               for (int i = 0; i < N; i++){
                for (int j = 0; j < N; j++){
                        srand(i + j);
                        RewardMatrix[i][j] = rand() % 10;}}}
void TrappingModification()
{              int trapping_state = 2;
        for (int i = 0; i < N; i++){
                ProbabilityMatrix[trapping_state][i] = 0;}
        ProbabilityMatrix[trapping_state][trapping_state] = 1;}
void main()
{       int i,j;
        AssignValues();
        TrappingModification();
        clock_t t1 = clock(); //Start counting;
        for (i = 0; i < N; i++){
                for (j = 0; j < N; j++){
        ImmediateBuffer = ImmediateBuffer + ProbabilityMatrix[i][j] * RewardMatrix[i][j];}
                ImmediateReward[i][0] = ImmediateBuffer;
                ImmediateBuffer = 0.00000;}
#pragma offload target(mic){
                for (int iteration = 0; iteration < 10000; iteration++){
                        for (int k = 0; k < N; k++){
                                V_buffer[k][0] = RewardBuffer[k][0];}
#pragma omp parallel for shared(V_buffer,ProbabilityMatrix,RewardBuffer,Result) private(i,j)
                        for (int i = 0; i < N; i++){
                                for (int j = 0; j < N; j++){
                Result = Result + ProbabilityMatrix[i][j] * V_buffer[j][0];}
                                RewardBuffer[i][0] = Result + ImmediateReward[i][0];
                                Result = 0.0;}}}
        clock_t t2 = clock(); //Start counting;
        printf("First Time = %f\n", (double)(t2 - t1) / CLOCKS_PER_SEC);}
```

Figure 6.3: The complete code for computing total expected reward.

Table 6.1 shows the results for running the program on the heterogeneous (1) CPUs and Intel Phis and (2) CPUs only for a variety of input matrix sizes ranging from 500 to 30000. The Markov process created from an attack graph for a typical system would generally have at least 500 states.

Size (number of rows)	500	1000	2000	3000	4000	5000	6000
CPU time(s)	0.59	3.17	22.69	60.22	106.62	167.44	241.01
PHI&CPU time(s)	2.01	2.69	3.74	6.36	9.86	14.85	20.19
Speedup	0.29	1.18	6.06	9.47	10.81	11.28	11.94
Size (number of rows)	7000	8000	9000	10000	15000	20000	30000
CPU time(s)	326.53	433.42	546.13	682.87	1760.51	4020.96	7482.57
PHI&CPU time(s)	26.36	34.57	43.24	52.34	125.48	301.49	542.83
Speedup	12.39	12.54	12.63	13.05	13.31	13.34	13.78

Table 6.1: Execution time for the total expected reward computation.

Table 6.1 indicates that the CPU only implementation is faster than CPU and Intel Phi implementation when the input matrix size is less than 1000 but the situation is reversed when the input matrix is greater than 1000. Compared to the performance of the program presented in Chapter 5 (see Table 5.1), the time spent on this program is significantly less. The reason is that the last program was required to repeat a process for many times until the steady state probability vector converges. The number of times for this iteration process must be conducted increases as the matrix size grows. Thus the larger the matrix is, the more time is spent on the computation. However, the program presented in this chapter is aimed at calculating an accumulated expected reward. There is no need to iterate a process and reach its limit.

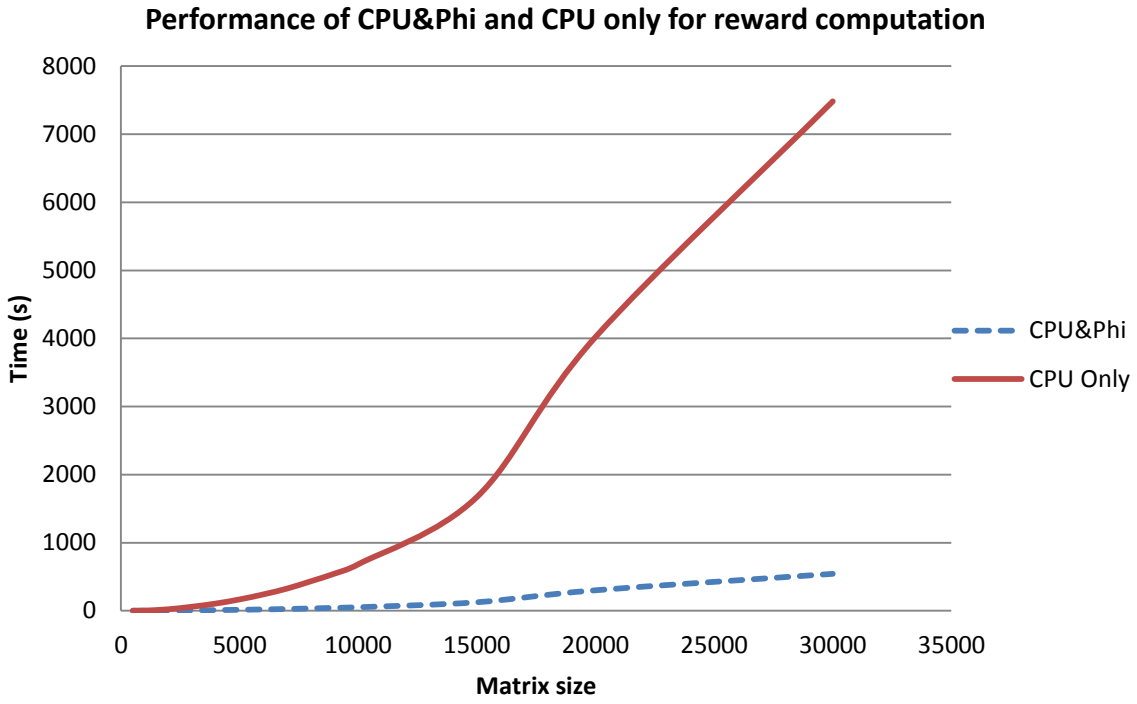

Figure 6.4: Performance of CPU&Phi and CPU only options for reward computation.

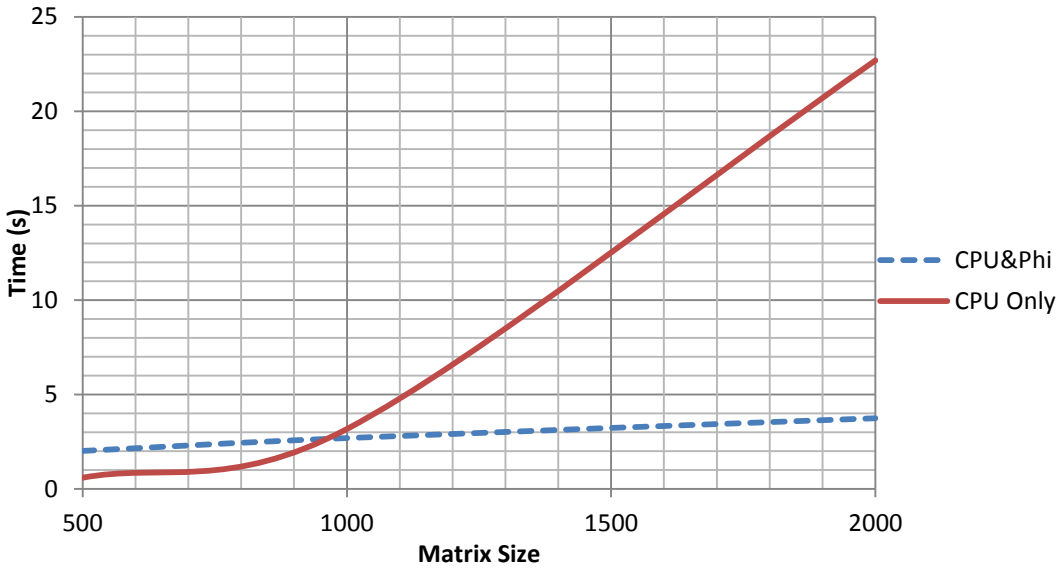

Figure 6.5: Zoomed-in view of Figure 6.4 for matrix size of 500 to 2000.

In both programs, the result of each iteration is needed for the next iteration. This means that executing iterations in parallel will result in data mismatch and yield incorrect results. In other words, each iteration must be executed in order and this is a serial

structure. Thus, the number of iterations required is the main determinant of the execution time of the program. Compared to the state necessity determination program, the reward computation algorithm is more complicated, so it seems that the state necessity program will take less execution time if both of the algorithms have the same iteration times. However, this is not true because there is a small difference between parallelism of these two algorithms. In the necessity determination algorithm (see Chapter 5), the process that a row vector times a matrix (also known as row-majored order) is iterated while the process in the reward computation (see Chapter 6) algorithm is a matrix times a column vector. The former process takes much more time than the latter one, because the CPU can only get a small group of consecutive data items from the cache line each time. The latter process makes full use of each single cache line because it uses neighboring memory locations (data items) while the former process spends a large amount of time on waiting for memory to transfer data that is not contained in the current cache line (not neighboring data). Thus the time spent on the program presented in this chapter is, in general, less than the time spent on the one presented in Chapter 5.

6.2.2 Performance Reward Computing Program using Memory Allocation

The program in Section 6.2.1 is for the case that the matrix size is below 30000. However, when the matrix size grows very large, a large amount of data causes memory leak due to the lack of memory, which will eventually result in the termination of the program. One way to solve this problem is to use "new" and "delete" functions to apply and release dynamic memory. Moreover, the input matrices should be broken into smaller pieces and these pieces have to be sent to the program one by one so that all the data can

be digested. However, it takes a huge amount of time to allocate and release memories, thus the performance of the program will inevitably decrease ever though memory leak problem is solved.

The main program (main() function of the program) for the case that the matrix size is above 30000 is shown in Figure 6.6. In order to reduce the execution time, the number of iterations times is now reduced to 5 while in the last program (Figure 6.3) it was 10000. The reason for choosing five iterations is to reduce execution time. If it is iterated ten times, the result is still valid but the time required will be doubled as the number of iterations is doubled, which is not practiced for testing performance based on the runtimes obtained for the 5-iteration case (over 2 hours in one case).

There are multiple ways to input a probability matrix and a reward matrix. In the necessity determination program, they are generated and modified by subroutines. However, it is not wise to use the subroutines in this program because matrices will occupy a very large amount of memory once they are generated. The ideal way to avoid abusing memory is to cut the matrices into small pieces and input them one by one. Line 6 defines n which represents the length of each data piece. In other words, the matrix will be cut to n times N dimensions. Line 7 defines the number of pieces that the matrices are broken into. Lines 11 to 12 define two pointers which point to the input files. Line 15 to 16 applies for memories in order to hold data for each piece. These pieces will be sent into Intel Phi and executed in parallel since each piece of data is independent. Line 20 to 23 read the data from input file. Lines 24 to 26 allocate more memory to the vectors which will participate in the reward computation. Line 27 to line 40 is for computing the total expected reward. Line 41 to 45 releases memories at the end of each iteration to

make sure that more memories is available in the next iteration. The combination use of functions "new" and "delete" can efficiently prevent memory leak and further crash.

```
#include <stdio.h>
#include <stdlib.h> // Head file for rand()
#include <omp.h> // Head file for OpenMP
#include <time.h>// Head file for Clock()
#define N 1000000
#define n 100
#define block_number N / n
using namespace std;
void main()
{   int x,y,u;
     FILE *fp = fopen("probability.txt", "r");
     FILE *fr = fopen("reward.txt", "r");
     clock_t t1 = clock();
     for (int iteration = 0; iteration < 5; iteration++){
                    float *ProbabilityMatrix = new float[n*(long)N];
                    float *RewardMatrix = new float[n*(long)N];
#pragma offload target(mic) in(ProbabilityMatrix, RewardMatrix:length(n*N)){
#pragma omp parallel for
                    for (u = 0; u < block_number; u++)
                           {for (x = 0; x < n*N; x++){
                                  fscanf(fp, "%f", ProbabilityMatrix[x]);}
                            for (y = 0; y < n*N; y++){
                                  fscanf(fr, "%f", RewardMatrix[y]);}
                            float *ImmediateReward = new float[n*(long)N];
                            float *RewardBuffer = new float[(long)N];
                            float *V_buffer = new float[(long)N];
#pragma omp parallel for
                            for (int i = 0; i < n; i++){
                                  float ImmediateBuffer = 0.0;
                                  for (int j = 0; j < N; j++){
     ImmediateBuffer = ImmediateBuffer + ProbabilityMatrix[N*i + j] * RewardMatrix[N*i + j];}
                                  ImmediateReward[u + i] = ImmediateBuffer;}
                                  for (int k = 0; k < N; k++){
                                  V_buffer[k] = RewardBuffer[k];}
#pragma omp parallel for
                            for (int m = 0; m < n; m++){
                                   float Result = 0.0;
                                   for (int q = m*N; q < (m + 1)*N; q++){
                            Result = Result + ProbabilityMatrix[q] * V_buffer[q - m*N];}
                            RewardBuffer[u + m] = Result + ImmediateReward[u + m];}
                                         delete[] ImmediateReward;
                                         delete[] V_buffer;
                                         delete[] RewardBuffer; }}
             delete[] ProbabilityMatrix;
             delete[] RewardMatrix;}
     clock_t t2 = clock();
     printf("Total Time = %f\n", (double)(t2 - t1) / CLOCKS_PER_SEC);
     printf("done");}
```

Figure 6.6: Main program when the states number is over 30000.

Table 6.2 shows the results for running the program on the heterogeneous cluster using (1) CPUs and Intel Phis and (2) CPUs only.

Size (number of rows)	10000	20000	30000	50000	80000	100000	150000
CPU time(s)	0.49	1.83	5.58	21.15	59.95	94.93	212.72
PHI&CPU time(s)	0.91	1.49	2.11	3.54	6.41	15.87	27.43
Speedup	0.51	1.23	2.64	5.97	9.35	5.98	7.76
Size (number of rows)	200000	300000	400000	500000	600000	800000	1000000
CPU time(s)	378.71	849.19	1584.28	2689.12	3196.24	5169.29	7952.92
PHI&CPU time(s)	41.42	83.61	138.65	223.35	331.61	590.52	950.69
Speedup	9.14	10.16	11.43	12.04	9.64	8.75	8.36

Table 6.2: Execution time for the reward computation when state number is large

Comparing Table 6.2 and Table 6.1, it can be observed that the execution time is much less in Table 6.2. This is because there are only five iterations in the program used to generate the results for Table 6.2 and 10000 in the program used to generate the results in Table 6.1. In fact, the program presented in this section generally needs much more time ever though both of the two programs (see Section 6.2.1 for the other program) implement the same algorithm. The main reason is that allocating and releasing memories takes some time and the time spent on memory operation increases as the number of iterations grows, since each iteration requires memory to be allocated and released again. Thus, the larger the input matrix the more time required for the iterations.

The graph of the performance of the program in this section executing on the heterogeneous cluster using (1) CPUs and Intel Phis and (2) CPUs only is shown in Figure 6.7. When the matrix size is greater than 18000, the Intel Phi is getting faster than CPU. Parallel computation (CPU&Phi) gains better performance over serial computation

(CPU) when the number of states grows very large even though communication issue exists.

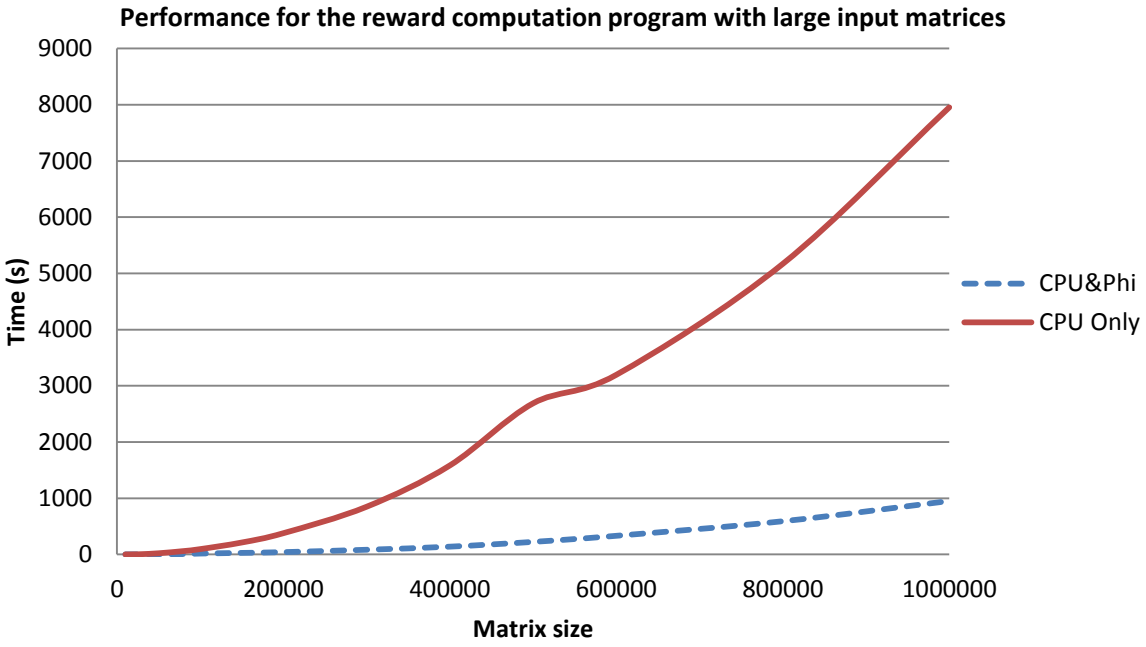

Figure 6.7: Performance for the reward computation program with large input matrices.

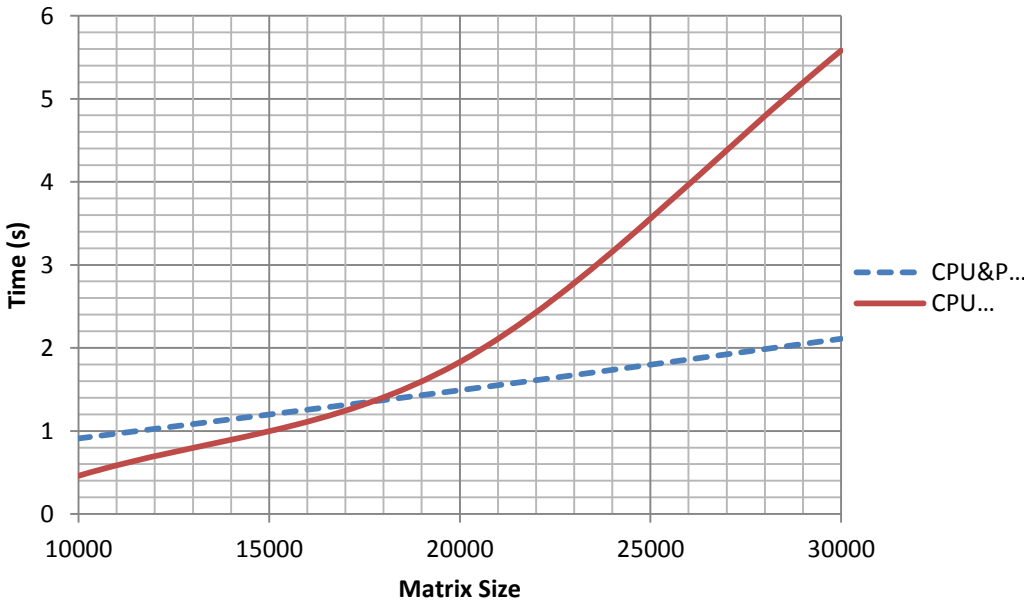

Figure 6.8: Zoomed-in view of Figure 6.7 for matrix size of 10000 to 30000.

CHAPTER 7

FUTURE WORK

This chapter presents future work and future research directions for the research presented in this thesis. There are three aspects of the future research, (1) the number of steps for the steady state probability vector to converge, (2) the defending strategy on multiple states, and (3) using different reward policy.

7.1 Algorithm for Convergence of Steady State Probability Matrix

In the first algorithm for determining necessity for each state in an attack graph, Equation (2-6) has to be iterated many times. The number of the iterations increases as the matrix size grows since a case with a larger matrix size needs more iterations to ensure that steady state probability vector converges.

However, this thesis does not give any instructions to compute the exact number of iterations needed for the steady state probability vector to converge (this thesis uses a large enough number of iterations to ensure convergence in most cases, however in certain cases fewer iterations could be used to reduce runtime), which might result in inaccuracy of the final result (i.e., the number the iterations is not enough for the convergence) or wasted effort as the steady state probability vector converged with fewer iterations (i.e., too many iterations and effort is wasted on iterations once the steady state probability vector converges). Therefore, developing an algorithm to determine the number of iterations for the steady state probability vector to converge is necessary in

order to obtain a correct answer at the cost of reasonable iteration numbers. The tradeoff of whether the additional time to determine the minimum number of iterations verses using a fixed number must be investigated as well.

Although this new approach would be of great help in ensuring accuracy and efficiency, a huge cost is also needed to compute the number of iterations or to determine when the steady state probability vector has converged. Thus, the total time spent on computing the steady state probability vector may increase due to the added computations, which would negate the savings by only iterating for the minimum number of times.

In conclusion, the algorithm to compute the number of iterations can yield an accurate and correct iteration numbers for the convergence, but it still needs further study to determine its cost.

7.2 Defending Strategy on Multiple States

Both of the two defending strategies presented in Chapters 3 and 4 convert only one state in the attack graph to a trapping state or regression state. It is reasonable to apply the defending strategy on a single state when trying to evaluate the necessity for each state. However, this single state has a very limited impact on the attack graph with a large number of states. Groups of several states might have a greater influence on the overall attack graph and more study on applying defending strategy on multiple states. This approach may also yield a reduction in the computational complexity of the process due to having to evaluate fewer permutations of states.

There are two methods to analyze the defending strategy on multiple states. The first one is to simply convert multiple states to trapping or regression states and modify

the probability matrix according to those converted states. The second one is to find out a way to simplify the attack graph. The former method is very simple since the algorithm is already built (see Chapter 3) and all that is needed is to update the algorithm to modify the probability matrix for multiple states and then iterate using Equation (2-6). The problem for this method is that there are too many choices when applying strategy on multiple states. The number of permutations grows rapidly as one considers all of the different possible groupings of states. It is unlikely to be possible to determine which combination has the most significant impact on the attack graph in a reasonable time. One approach that may help is to restrict groups as those states that are neighbors.

The second algorithm is aimed at simplifying a complicated attack graph to a simpler (smaller) one. For instance, multiple states can be converted to a single combined state (e.g., a super-state) so that a new attack graph with fewer states is obtained. This method requires a complex computation to determine the super-states and to update the state transitions in the process. Then the super-states can participant in determining state necessity.

It is believed that the second method is the better option because fewer states are required resulting in a smaller matrix and fewer computations to determine state necessity. Further research into identify the super-states and modifying the process to work with these super-states is required to determine if the approach provides a savings with respect to runtime.

7.3 Optimize total expected reward by using multiple reward matrices

In Chapter 4, the reward matrix is a matrix whose values is fixed and is not modified even if the attack graph has changed. Thus, increasing or lowering the total expected reward only depends on the transition probability matrix, which has very limited influence on manipulating the total expected reward because the transition probability matrix can only be modified according to defending strategy on each state. However, if there are multiple choices for the reward matrix at each transition, it will be more flexible in maximizing or minimizing the total expected reward. At each transition, the total expected reward based on a different reward matrix should be separately calculated and the maximum reward should be found out by comparing all of them. Thus, optimal strategy can be determined to optimize the total expected reward. This approach allows exploration of multiple attacker strategies or behaviors and assumes that the attacker will select the strategy that provides them with the greatest reward at each step (i.e., the attacker is following a greedy algorithm).

This procedure is more expensive because having multiple choices for reward matrices (multiple strategies) means more computation work. However, it is very helpful to control the total expected reward and thus determines the difficulty to enter the goal states.

CHAPTER 8

CONCLUSIONS

The growing trend of cyber-attacks proposes a potential threat to society, jeopardizing personal privacy and public interests. Studies for cyber security are very essential to analyze cyber-attacks and eliminate illegal access to important data. This thesis presents research that builds capabilities to perform cyber-analysis model based on Markov process model, and developed parallel algorithms to analyze attack graphs using these techniques. The algorithms are able to measure the necessity (importance) of each state in the attack graph and they have been implemented on a heterogeneous computing cluster consisting of Intel Phis and CPUs to achieve a superior performance.

The necessity of each state in the attack graph is evaluated by converting each of state into either a trapping or a regression state. The steady state probability vector indicates the chance to get to goal states and it is going to change due to the modification of transition probability matrix. The defending strategy is capable of analyzing the impact describing the importance of each state and it can be used to eliminate cyber-attacks by lowering the chance of the attacker reaching one of the goal states. This thesis successfully implemented parallel programs to apply the two defending strategies (trapping state and regression state strategies) to the necessity determination algorithm in order to evaluate the importance of each state in attack graph. However, in the case that the number of states is very large, this method is unlikely to prevent cyber-attacks efficiently since a single state has limited impact and groups of states must be

investigated. Other methods aimed at reconfiguring the Markov process into a more suitable form to reduce or combine the goal states into fewer states is one possible approach to addressing this issue. However, the technique and implementation presented in this thesis is valid in defending against specific attacks that target a specific goal state.

Computing the total expected reward is another method to evaluate the importance of transitions in the attack graph. The defending strategy on each state results in fluctuation of the total expected reward. This reward may represent the difficulty to compromise the system. The thesis creates and evaluates the performance of an algorithm to compute the total expected reward and uses two defending strategies to increase the total expected reward to interrupt or terminate the attack (e.g., make it more costly to continue the attack). A more flexible way to control the total expected reward is to choose optimal policy when there are multiple choices for the reward matrix.

Both of the algorithms involve a large amount of computational work. Software implementations of theses algorithms targeting a heterogeneous cluster consisting of CPUs and Intel Phis is presented to handle simple computation work involving a large quantity of data. The Intel Phi is a powerful coprocessor, which is able to assist the CPU dealing with massive and intensive computation work. This thesis used both CPUs and Intel Phis to implement the developed algorithm and to collect experimental results of the runtime of the implementations. These results show that Intel Phis is indeed very good at massive-data computation and that the CPU and Phi implementation is overwhelmingly superior to serial algorithm. Moreover, increasing parallelism in the program is of great help to improve the efficiency since elements in matrix computation is usually independent.

BIBLIOGRAPHY

[1] Vivek Shandilya, Chris B. Simmons, and Sajjan Shiva, "Use of Attack Graphs in Security Systems," Journal of Computer Networks and Communications, vol. 2014, Article ID 818957, 13 pages, 2014.

[2] S. Jha, O. Sheyner and J. Wing, "Two formal analyses of attack graphs," Proceedings 15th IEEE Computer Security Foundations Workshop. CSFW-15, 2002, pp. 49-63.

[3] Kaluarachchi, P. , Tsokos, C. and Rajasooriya, S, "Cybersecurity: A Statistical Predictive Model for the Expected Path Length," Journal of Information Security, 7, pp.112-128, 2016

[4] R. A. Howard, Dynamic programming and Markov processes (technology press research monographs), 7th ed. Cambridge, MA: M.I.T. Press, 1960

[5] G. Franca; J. Bento, "Markov Chain Lifting and Distributed ADMM," in IEEE Signal Processing Letters , vol.PP, no.99, pp.1-1, Jan.2017

[6] Y. Hao; M. Wang; J. H. Chow, "Likelihood Analysis of Cyber Data Attacks to Power Systems with Markov Decision Processes," in IEEE Transactions on Smart Grid , vol.PP, no.99, pp.1-1, Nov 2016

[7] Nong Ye, Yebin Zhang and C. M. Borror, "Robustness of the Markov-chain model for cyber-attack detection," in IEEE Transactions on Reliability, vol. 53, no. 1, pp. 116-123, March 2004

[8] B. Bylina and J. Potiopa, "Data structures for Markov chain transition matrices on Intel Xeon Phi," 2016 Federated Conference on Computer Science and Information Systems (FedCSIS), Gdansk, 2016, pp. 665-668.

[9] "Massive DDoS Attacks Take Down Major DNS Service Provider." FARS News Agency. N.p., 22 Oct. 2016. Web. 15 Mar. 2017.

[10] J. Jeffers, J. Reinders and A. Sodani, Intel Xeon Phi coprocessor high-performance programming, 1st ed. .