EVOLUTIONARY STRATEGIES FOR SECURE MOVING TARGET CONFIGURATION DISCOVERY

BY

ROBERT WALTER SMITH

A Thesis Submitted to the Graduate Faculty of

WAKE FOREST UNIVERSITY GRADUATE SCHOOL OF ARTS AND SCIENCES

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

Computer Science

May, 2014

Winston-Salem, North Carolina

Approved By:

David John, Ph.D., Advisor

William Turkett, Ph.D., Chair

Daniel Cañas, Ph.D.

Errin Fulp, Ph.D.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1252551.

Table of Contents

Acknowledgments ii

List of Abbreviations v

List of Figures vi

List of Algorithms vii

Abstract viii

Chapter 1 Introduction 1
- 1.1 Cyber Attacks 1
- 1.2 Secure Computer Configuration Management 2
- 1.3 The Moving Target Strategy 3

Chapter 2 The Genetic Algorithm Moving Target Strategy 4
- 2.1 Automated Computer Reconfiguration 4
- 2.2 Evolutionary Strategies 4
- 2.3 Moving Target Defense 4
- 2.4 Machine Learning 5
- 2.5 Related Research 5

Chapter 3 Evolutionary Strategies 7
- 3.1 Genetic Algorithm 7
- 3.2 Simple Genetic Algorithms 10
  - 3.2.1 Simple Genetic Algorithm Definition 10
  - 3.2.2 Complexity Analysis 11
  - 3.2.3 The No Free Lunch Theorem 14
  - 3.2.4 The Fundamental Theorem of Simple Genetic Algorithms 15
  - 3.2.5 Conclusions from Schemata Theorem 17
- 3.3 Beam Search 18

Chapter 4 Classification Algorithms 19
- 4.1 Support Vector Machines 19
- 4.2 The Classification and Regression Tree Algorithm 20

Chapter 5 Moving Target Framework Details 23
- 5.1 System Assumptions 23
  - 5.1.1 Genetic Algorithm Constraints 24
- 5.2 Framework Overview 24
- 5.3 XML Configuration Description 27
  - 5.3.1 Parameter Types 28

Chapter 6 System Performance Measures 31
6.1 Diversity Measurements 31
6.2 CVSS Vector Vulnerability Classification 32
6.3 Attack Simulation For Fitness Estimation 34
6.4 Adaptation Rate 35

Chapter 7 Evolutionary Strategies For Configuration Generation 36
7.1 The Moving Target Genetic Algorithm 36
7.1.1 Selection 36
7.1.2 Crossover 44
7.1.3 Parameter Value Mutation 50
7.2 The Moving Target Beam Search 50

Chapter 8 Machine Learning Techniques for Domain Restriction 52
8.1 The Temporal Classifier 53
8.1.1 Parameter Domain Mutation 53
8.2 The Spatial Classifier 58
8.2.1 Configuration Classification 67

Chapter 9 Discussions 68
9.1 Evolutionary Strategies for Moving Target Configuration 68
9.2 Supervising Evolutionary Strategies with Machine Learning 69
9.3 Machine Learning for Moving Target Configuration 69
9.4 Research Results 70

Chapter 10 Future Work 73
10.1 Automated Feasibility Testing 73
10.2 Automated Attack Detection 73
10.3 Confounding Factors in System Assumptions 73
10.4 Diversity Measurements and Enforcement 74
10.5 Improvement of Classification Accuracy and ES Fitness Increase 75
10.6 Specialization to Types of Defended Networks 76

Bibliography 78

Appendix A Evolutionary Framework Code 80
A.1 Settings and Global Variables 80
A.2 Functions 84
A.3 Evolutionary Strategy Implementation 104

Appendix B Parameter Decriptions 118

Vita 134

List of Abbreviations

CART	Classification and Regression Tree
CIA	Confidentiality, Integrity, Assurance
CVSS	Common Vulnerability Scoring System
DDOS	Distributed Denial of Service
ES	Evolutionary Strategies
GA	Genetic Algorithm
IP	Internet Protocol
LAN	Local Area Network
MT	Moving Target
MTGA	Moving Target Genetic Algorithm
SGA	Simple Genetic Algorithm
SVM	Support Vector Machine

List of Figures

3.1 Single Point Crossover And Mutation GA Operators 13

4.1 Illustration of SVMs 21

5.1 MT System Framework Overview 24
5.2 Option List Parameter XML Example 26
5.3 Numeric Parameter XML Example 26
5.4 Chromosome XML Example 26
5.5 Illustration of Two Sample Probability Functions 28

7.1 Roulette Wheel Selection Illustration 37
7.2 Selection Type Test Results Part 1 38
7.3 Selection Type Test Results Part 2 39
7.4 Selection Type Test Results Part 3 40
7.5 Tournament Selection Illustration 41
7.6 Three Point Crossover Illustration 46
7.7 Two Point Crossover Illustration 47
7.8 Crossover Type Test Results Part 1 48
7.9 Crossover Type Test Results Part 2 49
7.10 Uniform Crossover Illustration 49
7.11 Beam Search Test Results 51

8.1 Illustration Of Group Ranges for SVM Training Data 54
8.2 Temporal Classifier Test Results 59
8.3 Single Attack Per Generation Genetic Algorithm Results 60
8.4 Single Attack Per Generation Beam Search Results 61
8.5 Single Attack Per Generation Genetic Algorithm With Classifier Results 62
8.6 Single Attack Per Generation Beam Search With Classifier Results . . 63
8.7 Single Attack Per Generation Genetic Algorithm With Multiple Tree Classifier Results 64
8.8 Single Attack Per Generation Beam Search With Multiple Tree Classifier Results 65

9.1 Random Search Results 72

List of Algorithms

1 The Basic Genetic Algorithm 9
2 The Simple Genetic Algorithm 12

3 Deterministic Size Four Tournament Selection 41
4 Nondeterministic Size Four Tournament Selection 42
5 Roulette Wheel and Deterministic Size 4 Tournament Selection Type Blending 44
6 Elitism 44
7 Two Point Crossover 45
8 Three Point Crossover 47
9 Uniform Crossover 47

10 The Calculation Of Settings' Historical Fitness Impact Tallies. 53
11 Parameter Domain Mutation For Option List Parameters. 54
12 Parameter Domain Mutation For Bit Vector Parameters. 56
13 Parameter Domain Mutation For Numerical Parameters. 57
14 The Spatial Classifier 66

Abstract

Robert Walter Smith

Defense against many cyber security threats can be implemented with existing software on the machine, without requiring patches for current programs or the installation of specialized security software. There are certain operating system or program parameters which, if set properly, can close security vulnerabilities. Learning a way to securely configure computers to prevent attacks potentially allows organizations to defend their machines with a relatively low cost.

Regular machine reconfiguration could also be used to implement another concept in computer security: the moving target defense. In this strategy, the resource being defended is regularly modified in the hopes of spoiling the attacker's reconnaissance efforts. Reconnaissance is an important part of a cyber-attack. If a modification happens between reconnaissance and the attack, it is likely that the exploits based on the attacker's outdated information will no longer be effective. It is also possible to protect many similar objects at once.In this case, the security changes across all objects should be diverse. This way, even if an attacker can successfully compromise one machine, the same strategy will only be effective against a small portion of the others.

Both these ideas, learning secure parameters and moving target defense, can be implemented through the use of a genetic algorithm. A genetic algorithm naively mimics the process of evolution. For this problem, a network of computers is modeled as a genetic population of organisms. In place of DNA, a computer's genetic code is described by the settings of all of its parameters. The model then simulates the breeding of these computers using the operators of selection, in which computers decide on mating partners based on their measured fitness; crossover, in which the parents pass some of each of their genes to the child; and mutation, which in which random changes are applied to the child's genes. Once all the child configurations have been created, they can be implemented on the managed computers.

Attackers are imagined as predators for this population, regularly stalking the machines configured by the current generation and attacking the vulnerable computers within it. These attacks reduce the fitness of the targeted computers, reducing their probability of being chosen during selection. Thus, over many generations of breeding, the population will become more resistant to attacks, as the genes of computers which were not vulnerable to attack spread throughout the population.

In this thesis, a genetic algorithm achieving both the goals stated above was developed. The genetic algorithm has been shown to evolve more secure computer configurations which when used with an automated reconfiguration system immunize the machines to attacks. This population of configurations is diverse and changes with every generation, creating a moving target defense as a consequence of its operation, using no additional resources. The system was demonstrated to work using only 40 generations, a much smaller number of generations than is normal for use with genetic

algorithms but is reasonable given the time constraints of this problem. Evolutionary strategies other than genetic algorithms have also been considered. Beam search is a greedy algorithm, accepting only the highest scoring configurations of the current generation into the next. In this thesis, a beam search based system prototype was developed. It was also more successful than the genetic algorithm in increasing average configuration fitness, while still maintaining a high amount of diversity.

This thesis also introduces the novel approach of using the artificial intelligence strategies support vector machines and classification and regression trees to supervise the evolutionary strategy. In the naive implementation, this was done by recording the changes in computers' observed security and correlating this information to changes between parents and children over many generations. This allowed for the classification of some settings as insecure. These insecure settings were forcibly removed from the population and the genetic algorithm modified so that they could not be reintroduced.

In this thesis, an implementation of this artificial intelligence based idea was added to both the genetic algorithm and beam search based systems. In both cases, it resulted in significant increases in the amount of total fitness gain, with very little impact on diversity.

A more complex implementation of this idea addressed the fact that, in the real world, only a small number of possible exploits will be used in any generation. This problem would normally prevent accurate fitness evaluation, as, in each generation, chromosomes are only graded based on the subset of attacks used against them. This separate classification scheme compared chromosomes within a single generation, searching for commonalities which would explain which traits the exploit required to succeed.

A more complex classification scheme can achieve results in only a single generation. Instead of measuring at score changes over many generations, it compares attacked and uncompromised computers in the same generation. Commonalities within these groups will allow for the classification of traits responsible for opening the exploited vulnerability.

Once such traits were identified, they were used to create a profile describing what sort of chromosome is vulnerable to that exploit. Chromosomes matching that profile were no longer allowed into the population. This approach has the added benefit of aiding in the understanding of why the system created certain kinds of configurations and not others, providing clues to help understand the exploit used against the managed computers.

A prototype system making use of these strategies was developed. Experimentation has shown that, for small numbers of attacks per generation, it was often successful in creating a new generation of chromosomes which is immune to the attacks from the previous generation. These results demonstrated that this technique has merit for improving computer security and providing greater understanding of the cyber security landscape.

Chapter 1: Introduction

Many cyber attacks exploit a misconfiguration in the targeted computer. These threats can be defended against simply by applying correct settings to some combination of parameters, such as a program's settings or permissions on a given file. However, attackers are capable of discovering new attacks faster than security experts can identify such problems and apply a new configuration rule. An automated approach to the discovery and implementation of secure configurations helps address this problem. Evolutionary Strategies (ES), which discover better solutions based on known good ones, are one possible basis for such a configuration management system. As an additional benefit, this approach creates a Moving Target (MT) defense incidentally, with no additional overhead, by ensuring that one machine will be configured differently than others and from previous versions of itself. Ideally, this will reduce the portion of the managed computers vulnerable to any given attack.

1.1 Cyber Attacks

In a cyber attack, the opponent will begin by searching the network for vulnerable machines. This search can involve finding Internet Protocol (IP) addresses with assigned machines or focus on gaining information about the computers, such as what operating system they are running and which versions of what software they have installed. Once complete, the attacker uses this information to design an exploit, a series of steps which will allow the attacker to achieve their goal on the targeted machine. Then the attack is actually launched. The attacker proceeds through the steps of the constructed exploit in order to interfere with the target computer in some way, such as by causing it to change or divulge stored data.

The first step described above, reconnaissance, is vital to the attacker. If the information it provides is inaccurate, then it is likely that the exploit based on them will fail. Take, for example, an attacker who finds a machine running a certain version of a web server program. The attacker knows a security flaw in this version of the

software, and prepares the exploit it. Suppose the target's software is updated before the attacker has time to design and execute the exploit. If this new version of the program no longer features the same security flaw, the hacking attempt designed to capitalize on that flaw will fail.

1.2 Secure Computer Configuration Management

A computer configuration is a set of instructions governing how a machine is to be operated. A configuration can be represented as a set of parameters and their associated settings. These parameters may be of a variety of types. For example, one parameter may be the permissions for a given file, while the associated setting is the bit vector describing the way those permissions are set. Other types of parameters include numbers in a given range or a list of possible options from which to select. A configuration consists of a selected setting for each parameter, such as assigning a length 9 bit vector to each filer's permissions. A configuration's security can be measured by the number and severity of the attacks to which a machine which implements it would be vulnerable.

In general, the discovery of a secure configuration given a description of all possible attacks is NP-complete. This can be shown by casting a configuration as a series of Boolean variables describing each of its parameters' settings. A secure configuration can then be defined by a Boolean expression which is true if and only if each vulnerability is closed by the configuration's adherence to the corresponding security rule. This problem can then be reduced to CIRCUIT-SAT [2].

In practice, this process of finding secure parameter settings is additionally hampered because the objective function associating configurations to computer security is not known beforehand. The security of a configuration can only be estimated via configuring a machine with it and determining if a successful attack will occur. Even when an attack occurs, it is not necessarily possible to determine which subset of parameters were responsible for opening the exploited vulnerability. Even worse, the utility of a given configuration can actually change over time as new exploits are discovered, rendering previously secure solutions insecure.

1.3 The Moving Target Strategy

A Moving Target (MT) defense's goal is to disrupt attacks by changing the state of the defended item, such as a computer or a specific program's memory. This will cause the attacker's knowledge of the target to become obsolete before it can be acted upon. This can act as a deterrent, because it has been reported that 45 percent of an attacker's time is spent in the reconnaissance phase [7]. Ideally, a change in configuration during the construction of an exploit will alter the computer such that the machine no longer contains the same vulnerabilities discovered during reconnaissance, thereby rendering the constructed attack ineffective. This concept is known as temporal diversity. A MT defense can also provide spatial diversity. Spatial diversity is the amount of difference between various machines at the same point in time. High spatial diversity limits any potential exploit to only a subset of the managed computers, as some will not exhibit the parameter settings which open the associated security vulnerability.

Chapter 2: The Genetic Algorithm Moving Target Strategy

In this thesis, a strategy for bolstering computer security is developed. This strategy combines a variety of ideas from diverse fields of computer science, such as security, biologically inspired algorithms, and machine learning. These ultimately allow for a network of computers to automatically adapt to the attacks they face.

2.1 Automated Computer Reconfiguration

A system framework has been developed which oversees the operation of a network of computers. [9] These machines are regularly assigned new configuration descriptions, and are automatically reconfigured to adhere to the policy described by their new configuration. Furthermore, the managed computers are monitored for security incidences, which are logged and assessed. Based on this collected data, each configuration is given a score representing its security.

2.2 Evolutionary Strategies

Evolutionary Strategies (ES) are a class of algorithms inspired by the process of natural selection. They model candidate solutions breeding to create a new set of solutions, which in turn breeds itself to create even further generations. Over time, this process ideally allows the population of solutions to adapt to its environment, discovering highly fit candidates. Here, the solutions are configurations and the environment is defined by the attacks the automatically configured machines face. This will allow for the discovery of more secure configurations, thereby preventing future attacks.

2.3 Moving Target Defense

The regular reassignment of configurations to machines creates a Moving Target (MT) environment. A common criticism of MT strategies is that it is not effective as a sole

defense, because it relies on cloaking security flaws instead of addressing them. This obfuscation can be penetrated by an attacker, and the cloaked flaw exploited. [16] However, the Moving Target Genetic Algorithm (MTGA) strategy generates a MT defense as a consequence of its normal operation. Therefore, it can be used to provide a MT defense essentially for free while pursuing its actual goal of finding secure solutions.

2.4 Machine Learning

Additional research has been completed to combine machine learning techniques with the ES. The machine learning leverages characteristics of the MTGA problem to identify insecure settings or configurations and forcibly removing them from the population. This speeds the elimination of insecure configurations from the population, increasing the security of the managed machines. It also allows for the collection of information about the ES's functioning, such as why it made certain decisions for a given parameter's setting, to be extracted in an understandable format.

2.5 Related Research

Prior research [4] [5] has addressed the idea of using GAs to manage computer configuration for a MT defense. However, this thesis improves upon this research in several key areas. Each configuration is made up of real parameters, with a variety of types reflecting the actual settings available when configuring a machine and each associated with possible real attacks which misconfiguration could allow. The prior research instead used simulated configurations comprised of single bit string, the bits of which notionally represented various parameter settings. Previous research also used identical, completely vulnerable configurations as the initial generation, but even the naive method of random configuration generation creates a significantly more secure and diverse population with which to initialize the GA. The system implemented in the previous papers also accounted for uncertainty in security evaluation by adding a small, random perturbation in the reported fitness score, whereas this thesis directly

simulates attackers targeting the system. This approach allows for very large discrepancies between observed and true security values, as would be expected in a real deployment.

Chapter 3: Evolutionary Strategies

The Moving Target Genetic Algorithm (MTGA) system relies on Evolutionary Strategies (ES) for the discovery of more secure configurations. ES mimic evolution by iterating through many generations of solution sets, basing each on the last. The goal is to gradually cause the population to adapt to its environment, thereby producing good solutions. In this thesis, each generation of the ES corresponds to a single reconfiguration cycle of the MTGA system. Genetic Algorithms (GA) and Beam Search are the two ES which have been implemented within the prototype.

3.1 Genetic Algorithm

Genetic Algorithms (GA) are a search and optimization heuristic inspired by the biological process of natural selection. A candidate solution is represented as a chromosome, a list of various possible traits of the solution, each with an associated value corresponding to the specific trait exhibited by that particular solution. To begin the algorithm, an initial population of n such chromosomes is randomly generated (Algorithm 1, Line 1). Each iteration of the algorithm then simulates the creation of a new generation, also of size n, allowing the current population to breed. This is accomplished via the processes of selection, crossover, and mutation. These three operators are the means by which the algorithm produces the next generation from the current. Selection (Algorithm 1, Lines $5-6$) chooses pairs of chromosomes from the current generation to serve as parents for chromosomes of the next. Selection usually prefers chromosomes of higher fitness. This causes highly fit solutions to be more likely to survive and multiply in subsequent generations, while still allowing chromosomes of lower fitness to be chosen, maintaining diversity in the population. Crossover (Algorithm 1, Lines $8-12$) exchanges parts of two selected parent chromosomes to create a child for the next generation. This is done so that different combinations of traits from parent chromosomes may be aggregated into a single child chromosome. Selection biases parent choice towards fit chromosomes. These

highly fit chromosomes likely have many high fitness traits, and ideally the child will exhibit even more of these than the parents. Mutation (Algorithm 1, Lines $14-17$) randomly changes a given trait, allowing the GA to introduce new trait values which were not in the current population or to reintroduce ones which were eliminated in previous generations. Crossover and mutation are associated with probabilities p_c and p_m respectively, which determine whether or not they will occur. The constant p_m can be defined in one of two ways, either the probability that each gene in each chromosome will be mutated or the probability that one chromosome will have some number of genes chosen to be mutated.

There must also exist a fitness function, which measures the value of each chromosome. This function takes as input a chromosome and returns a nonnegative real number. The higher this number, the better the chromosome functions as a solution to the problem. If $f(A) > f(B)$, the chromosome A strictly is of higher utility than chromosome B. Thus solution A is considered superior to solution B, according to the criteria laid out when defining the fitness function. The goal at each iteration is to produce a new generation with chromosomes of higher fitness and ultimately to produce a chromosome with as high a fitness score as possible which serves as the algorithm's solution to the problem the fitness function represents [14]. If the fitness function's output is properly correlated with a chromosome's effectiveness as a solution, and if the GA is successful in evolving a high fitness chromosome, which experience shows is often the case, then the algorithm provides a useful method for discovering relatively good maximizers for a particular optimization problem [13].

Some stopping criterion must also be set for the process, such as a given percentage increase in average fitness values for the population, or some set number of generations. In this thesis, the stopping condition will always be a constant number of generations.

This chapter provides a brief overview of the field of GAs. A standard introductory text contains a more detailed treatment of the subject [14].

```
1  initialize currentgeneration of n random chromosomes;
2  while stopping criteria is not met do
3  |   Calculate the fitness of each chromosome in currentgeneration
   |   nextgeneration = ϕ;
4  |   for i = 1 to n do
   |   |   /* Select two parents from the current population of
   |   |      chromosomes                                          */
5  |   |   parentA = Select (currentgeneration);
6  |   |   parentB = Select (currentgeneration);
   |   |   /* Apply the crossover operator                         */
7  |   |   prob = Random (0.0,1.0);
8  |   |   if prob < p_c then
9  |   |   |   child = Crossover (parentA,parentB);
10 |   |   else
11 |   |   |   child = parentA;
12 |   |   end
   |   |   /* Apply the mutation operator                          */
13 |   |   prob = Random (0,1);
14 |   |   if prob < p_m then
15 |   |   |   child = Mutate (child);
16 |   |   end
17 |   |   add child to nextgeneration;
18 |   end
19 |   replace currentgeneration with nextgeneration;
20 end
```

Algorithm 1: The prototypical flow of a basic Genetic Algorithm GA. The algorithm begins with generating n random chromosomes. It then proceeds in a loop until the stopping criteria are met. During each iteration, it evaluates each chromosome's fitness, then proceeds through the operations of selection, crossover, and mutation.

3.2 Simple Genetic Algorithms

A Simple Genetic Algorithm (SGA), originally developed by John Holland [6], is a type of GA subject to very tight constraints on chromosomal representation and operator implementation in order to present a GA in a very basic, easily understood form. This allows for an easier analysis of the algorithm. Several theoretical properties of SGAs have been proven which are not known for more general GAs, such as the performance shown by the Fundamental Theorem of Genetic Algorithms. This, in turn, provides an intuition, though not a definite proof, of what happens in more complex GAs, and suggests a theoretical basis for understanding GAs to be an effective technique. There are no mathematical proofs that guarantee SGAs will find good solutions. SGAs are a tool of theoretical analysis, but are very rarely used in real applications, as their strict formulation presents difficulty in creating chromosomal representations of many real world problems. Also, it has been observed that other GA representations and operator implementations often produce more rapid fitness gain.

3.2.1 Simple Genetic Algorithm Definition

A chromosome is a candidate solution, mapped to a representative form which the GA operators can use. In a SGA, each chromosome is represented by a fixed length string of l bits. A population is some arbitrary, constant number, n, of chromosomes which constitute the current working set. Any arbitrary non-random fitness function may be used, allowing the SGA to be applied to any real problem, though with no guarantees of particular levels of performance. During each iteration, the three operators of selection, crossover, and mutation are applied in order to evolve the next generation of chromosomes from the current population. Crossover and mutation are illustrated in Figure 3.1. First, each chromosome in the population is evaluated by the fitness function (Algorithm 2, Line 4). The selection operator (Algorithm 2, Lines $6 - 21$) is then called $2n$ times. Selection is fitness proportional (also known as roulette wheel) selection with replacement. This means that, each parent is chosen with probability

proportional to its fitness, and a single chromosome in the current generation can be selected arbitrarily many times. There is a p_c probability (Algorithm 2, Lines $22-23$) that each pair of parent chromosomes undergoes crossover, and then a p_m probability (Algorithm 2, Lines $30-31$) that each bit in each child chromosome will be mutated, where p_m and p_c are arbitrarily defined constants. Crossover (Algorithm 2, Lines $24-25$) is single point, in which a random bit position is chosen with uniform probability, and all bits after it are exchanged between the two parents. The first of the pair is then inserted into the next generation as the child, while the other parent is discarded. Mutation (Algorithm 2, Line 32) simply flips the mutated bit. Each successive population created is known as a generation. The set of newly generated chromosomes becomes the new population. Then the next iteration begins. There must also be some arbitrary stopping criteria. For our purposes, the stopping criterion is completing some arbitrary number of generations.

3.2.2 Complexity Analysis

In order to be useful as a search algorithm, the SGA must approximate an answer in a reasonable amount of time, even when it is applied to a problem with known high time complexity. Selection requires the sum of all fitness values in order to find the ratio of each chromosome's fitness score to the total fitness of the generation, which takes $O(n)$ time. It then selects a random number modulo n, a process which is repeated n times. Thus, selection is $O(n)$. Crossover exchanges at most l bits and is repeated $\frac{n}{2}$ times. Thus, crossover is $O(nl)$. Mutation flips a bit, and is repeated at most l times for each of the n chromosomes. Thus, mutation is $O(nl)$. Since the fitness function is arbitrary, it could have any time complexity, and is repeated n times. In total, each iteration has a time complexity of $O(nl + nT_f)$ where T_f is the time taken per evaluation of the fitness function. Fitness functions are often chosen with low time complexities, and thus nl will be the dominant term. Assuming the stopping criterion is to halt after some arbitrary, constant number of steps, and T_f is less than polynomial in n and l, the entire algorithm is then polynomial in terms of l and n. It should be noted that in some cases there will be no known way to create

```
1  initialize currentgeneration of n random chromosomes;
2  while stopping criteria is not met do
3      Calculate the fitness of each chromosome in currentgeneration
       nextgeneration = φ;
4      totalFitness = sum of all Fitness values of currentgeneration
5      for each chromosome n do
6          prob = Random (0.0, totalFitness)
7          for each chromosome m in currentgeneration do
8              if prob < Fitness (currentgeneration [m]) then
9                  parentA = currentgeneration [m]
10             else
11                 prob-= Fitness (currentgeneration [m])
12             end
13         end
14         for each chromosome m in currentgeneration do
15             if prob < Fitness (currentgeneration [m]) then
16                 parentB = currentgeneration [m]
17             else
18                 prob-= Fitness (currentgeneration [m])
19             end
20         end
21         prob = Random (0.0,1.0)
22         if prob < p_c then
23             crossoverPoint = Random (1.0, l)
24             child = parentA [1, crossoverPoint ] + parentB [crossoverPoint, l]
25         else
26             child = parentA
27         end
28         for each gene j in chromosome n do
29             prob = Random (0,1)
30             if prob < p_m then
31                 child [j] = 1 - child [j]
32             end
33         end
34         add child to nextgeneration;
35     end
36     replace currentgeneration with nextgeneration;
37 end
```

Algorithm 2: The Simple Genetic Algorithm. Problems are represtened as a fitness function and a chromosome length. Given these as input, the SGA will attempt to evolve a final generation of high fitness candidate solutions.

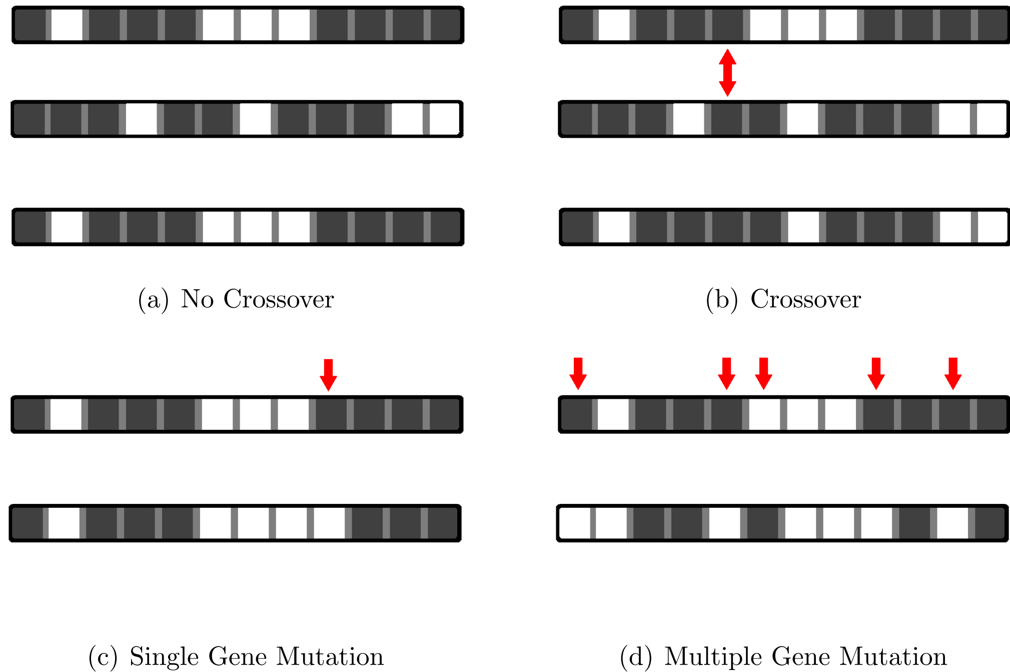

(a) No Crossover
(b) Crossover
(c) Single Gene Mutation
(d) Multiple Gene Mutation

Figure 3.1: GA crossover and mutation operators are demonstrated on example chromosomes. In 3.1(a) two chromosomes, each containing many genes set to either 0 or 1, have been selected. Crossover does not take place, and thus the first, unmodified, becomes the child. In 3.1(b) crossover occurs. The marked location is chosen as the crossover point, and the genes after that point are transplanted from the second to the first to become the child. In 3.1(c) a gene is chosen for mutation, and its value complemented. In 3.1(d) several genes of the same chromosome are chosen for mutation in a single generation.

a proper fitness function which can run in polynomial time due to theoretical time constraints on the algorithm for computing the true fitness value for that problem. These high time complexity fitness function often limit the GA's usefulness, though a GA can still be useful if there are no other applicable strategies which are faster.

Many important search and optimization problems are members of NP. However, the SGA halts in polynomial time regardless of the provided problem, as long as a polynomial or smaller time complexity fitness function is known. If it is successful in raising fitness by a substantial amount, it can be used to provide an estimate for a NP problem in polynomial time.

3.2.3 The No Free Lunch Theorem

The No Free Lunch Theorem [19] proves that there is no search algorithm, such as a GA, which is universally better than any other search algorithm over all search spaces. Consider the set of all possible solutions to a given problem, each of which has an associated utility value. A search algorithm's goal is to produce high utility in a short time. Search algorithms essentially operate by choosing a solution and checking if it has a higher utility than the current candidate. Then the only difference between two algorithms is the order in which they check solutions. Thus, if one algorithm outperforms another on a given problem, it is only because a higher utility solution happened to lie earlier within the order checked by one algorithm than the other, and this is not true for sets of solutions in general. In fact, one algorithm besting another on a given problem demands that the other algorithm will produce a better result on some other problem.

The formal statement of the No Free Lunch theorem is essentially given by the equation:

$$\sum_{f} P(d_m^y|f, m, a_1) = \sum_{f} P(d_m^y|f, m, a_2) \tag{3.1}$$

Where f is some optimization problem, d_m^y is a size m sample of points from the set of utility values y, a_1 and a_2 are any two optimization algorithms, and $P(d_m^y|f, m, a)$ is the conditional probability that a given d_m^y is obtained for the specified f, m, and a. [19]

Equation 3.1 states that the sum of performances of any two algorithms over all search spaces are equal. This means that all algorithms are equivalently good for searching and optimization when examined over all possible search spaces. If one algorithm performs better than another on some class of problems, then this discrepancy must be made up for by the existence of another class of problems for which it performs worse. This implies that there are no algorithms which are better than any other over all search spaces. In order to produce a more efficient algorithm, there must be some pattern to the assignment of utility values to solution candidates, and

the algorithm must be designed to take advantage of this pattern. Thanks to the No Free Lunch Theorem, proving that an algorithm is worthwhile requires demonstrating that it at least outperforms random search, as random search will, on average, perform just as well as any other search algorithm over all search spaces. It should be noted that most algorithms implicitly include such knowledge of the problem domain in their construction, and thus can outperform random search by violating the assumptions of the No Free Lunch theorem. For instance, algorithms designed to solve the knapsack problem will perform poorly at solving the traveling salesman problem, but in practice will never be applied to such a problem anyway.

3.2.4 The Fundamental Theorem of Simple Genetic Algorithms

The behavior of a SGA can be understood through the concept of schemata. A schema is a string of the same length of the chromosomes, l, consisting of 0s, 1s, and asterisks, which represent "don't care." A schema for chromosomes of length 4 may be $*01*$, which has four possible instances, $0010, 1010, 0011, 1011$. Over time, a SGA finds good simple schemata, which it uses as a basis for finding better schemata of higher complexity. A schema is "good" at time i if $\hat{u}(H, t)$, the average fitness of its instances, is relatively high in comparison to the average fitness for all schema, $\bar{f}(t)$, during the same generation. It can accomplish this because each chromosome of length l is an instance of 3^l schemata. Each time an example of a schemata receives a fitness score, the GA is provided with a new sample point from the set of all fitness scores of chromosomes adhering to that schema, providing a more accurate average each time it evaluates a new instance. This allows each evaluation of the fitness function on a chromosome to implicitly estimate the utility of each schemata of which that chromosome is an instance. This explains the SGA's ability to produce high utility solutions, as it begins concentrating on subspaces of the search space with increasingly higher average utility values.

It can be mathematically shown that low complexity schemata with high average utility over their instances are expected to receive exponentially more representatives within the population over time. In full, this is given by the schema theorem [6]

$$E(m(H,t+1)) \geq \frac{\hat{u}(H,t))}{\bar{f}(t))} m(H,t)(1-p_c\frac{d(H)}{l-1})[(1-p_m)^{o(H)}] \quad (3.2)$$

Where H is a given schema, t is the number of the current generation. $m(H,t)$ is the number of instances of H in generation t. $E(x)$ is the expected value of x, $\hat{u}(H;t)$ is the observed average value of schema H at time t. $f(t)$ is average fitness of the population of generation t. p_c and p_m are the probabilities of crossover and mutation, respectively. l is the length of length of a chromosome. $o(H)$ is the number of 0 or 1 bits set by the schema. $d(H)$ is the schema's defining length, the size of the region where, if the crossover point is located inside it, the schema could be destroyed in crossover.

The right hand of Expression 3.2 can be thought of as the product of three terms. The first, $\frac{\hat{u}(H;t)}{\bar{f}(t)} m(H,t)$ is the number of instances of H which are placed in the next generation by selection. The next, $(1-p_c\frac{d(H)}{l-1})$ is the probability that a given instance survives crossover, which is to say that, after crossover, the child is also an instance of schema H. Similarly, $[(1-p_m)^{o(H)}]$ is the probability that it survives mutation. Collectively, they calculate the number of instances which are selected and not destroyed by crossover and mutation, and thus give a lower bound on the number of instances of H present in the next generation. When $\hat{u}(H,t)=0$, the equation provides no useful information, only the tautology that the expected value is at least 0. This shows the weakness of GAs: if there are no instances of some schema in the population, the GA cannot estimate its fitness. If $\hat{u}(H,t) < \bar{f}(t)$, meaning that the observed fitness of H is below the average for all schema, then $E(m(H;t+1))$ will decrease exponentially as time goes on. Similarly, if $\bar{u}(H;t) > \bar{f}(t)$ then the number of instances will increase exponentially [13]. This exponential growth is important, because such exponential growth of historically better options is a very desirable property of search algorithms in which the algorithm is penalized for each poor candidate checked, as demonstrated by the two armed bandit problem.

The two armed bandit problem is a problem relating to resource allocation during a search. It posits a two armed slot machine. Each arm has a different payout

function, both of which are unknown to the searcher. The searcher is given a limited number of tokens, each of which allow for a single play of either arm, gaining a payout as determined according to the probability defined by that arm's payout function. The goal is then to achieve maximal payout. The optimal strategy is to assign exponentially more trials to the arm which has, thus far, provided a higher payout [13].

This solution suggests that the optimal solution to resource allocation between competing options in a search is to test the option which has historically proven to have higher average value exponentially more than the other. This is similar to the behavior of a SGA in regards to schema in the population, which suggests that the SGA approaches optimality for a such a search algorithm.

3.2.5 Conclusions from Schemata Theorem

Schemata theory serves as the basis for the Fundamental Theorem of SGAs, which is vital for demonstrating SGAs are a worthwhile approach to searching problems. It shows that SGAs leverage nonrandom characteristics of the objective function to find good solutions, instead of being the sort of zero knowledge algorithm covered by the No Free Lunch Theorem. However, schemata theory does not guarantee that the algorithm will converge to a global optimum, only that it is likely it will do better than random search, if high fitness schema were present in the initial population. Schemata theory also only applies to SGAs. However, SGAs are an extremely simple and rigidly defined subset of GAs, one which is rarely used in practice. Although it provides insight into how more complex GAs function, its conclusions do not apply to them, and there is no known generalization of the Fundamental Theorem of SGAs to other types of GAs. Experts have observed, based on experimental results in which GAs are capable of finding good solutions, that a similar process likely occurs in different types of GAs.

3.3 Beam Search

Beam Search [17] is an evolutionary strategy for searching. It begins by randomly picking n, the "beam width", candidates from the search space. It then checks each node adjacent to one of those in the current working set according to some heuristic function which estimates its distance to the solution. The MT Beam Search implementations defines an adjacent node as one which is a single mutation away from the current one. Selection uses a greedy strategy, the n nodes with the best heuristic scores becoming the new working set, and the whole process is repeated.

In the case of the MT system, there is no known way to reliably estimate the distance to the solution. Thus, instead of seeking minimal distance to the goal, it selects based on fitness scores. This effectively sets the goal state as a configuration with maximal fitness.

Creating an adjacent node requires only constant time to mutate a gene. For population size n, n adjacent nodes are created per generation, and each has its fitness evaluated. Thus, the time complexity is $O(n+T_f)$. This is equivalent to $O(n)$, given the restrictions on fitness function time complexity as shown in Section 3.2.2.

Beam Search was chosen due to the natural way in which it creates a new generation by selecting only a subset of the current generation, because each chosen chromosome will produce $beamWidth$ children. Other greedy algorithms could also be used in future work.

Chapter 4: Classification Algorithms

In addition to the ES described in Chapter 2, another kind of strategy is utilized by the developed MT system. This strategy, classification, deals with grouping data objects into one of several categories based on their similarity to known examples of each category. Normally, a data set is provided to the classifier, consisting of a variety of objects with many characteristics and the label of the group to which each belongs. The classifier uses this training data to create a model for the feature space. The classifier is then provided with new unlabeled data points and it returns the label of the group to which each data point belongs based on the generated model.

Classification techniques can enhance the ES by singling out low fitness gene combinations and removing them from the population. This will be achieved by correlating a parameter's setting changes to the chromosome's fitness changes, or by comparing attacked machines to machines which were not attacked. A classifier can then be trained on this data, and used to classify future settings or chromosomes as either secure or insecure.

4.1 Support Vector Machines

Many computer parameters will have a very large space of possible settings. Thus, the classifier must determine which region(s) within this space contain the insecure values, given relatively few data points. A Support Vector Machines (SVM), an algorithm which partition the space into groups, is a useful method for achieving this goal.

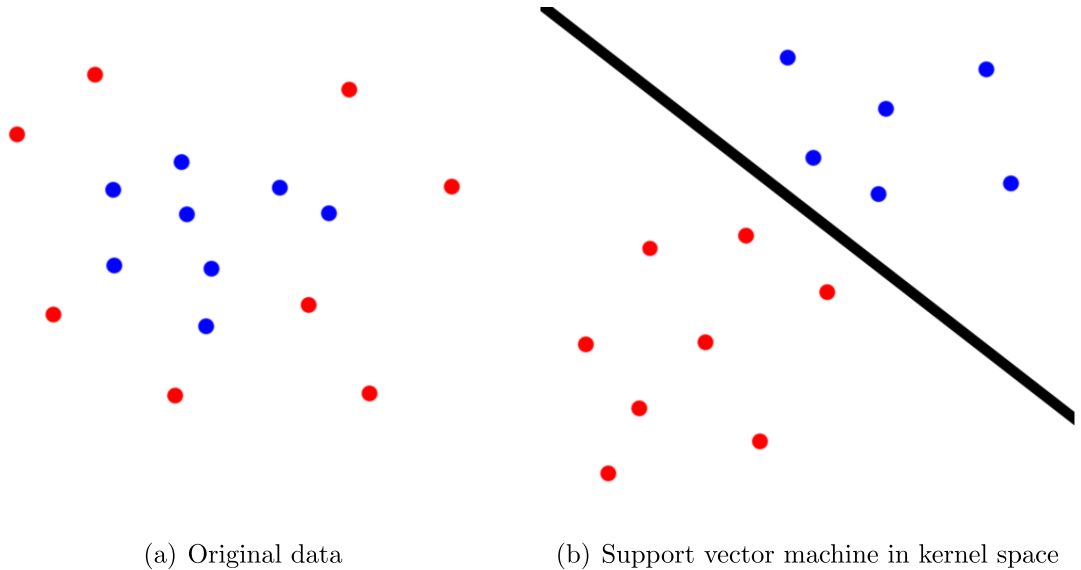
SVM is a machine learning technique used for classification. The SVM classifier is given a set of training data consisting of several points, each of which is an m-tuple of real numbers and belongs to one of two groups. The SVM then constructs a hyperplane which separates the groups with the largest possible margin, the distance from the hyperplane to the closest data points of each group. This is achieved using two support vectors, hyperplanes which contain at least one point of one group and no points of the other. The next step is solving the optimization problem of defining

these hyperplanes with maximal distance between them. Finally, the SVM is then the hyperplane half way between them, which splits the space into two regions, one for each of the two groups. The SVM's approach of choosing a line with maximal margins is intuitively a reasonable approach, because the ambiguous space between the two groups is split evenly [3].

Several additional techniques can also be used to improve the SVM's results. SVMs are linear classifiers, meaning that the line separating the two groups is defined by a linear function. SVMs cannot function when the groups cannot be linearly separated. Even when linear separation is possible, there may be a different type of line, such as a parabola, which provides smaller margins. These problems can be addressed by projecting the training data into a higher dimensional space with a function, called a kernel function. This kernel function is described by how a straight line appears when projected back from the higher dimensional space to the lower one. Thus, a polynomial kernel function produces classifying lines which appear to be polynomial, and a radial basis function produces ones which appear roughly circular. In this new space, the data points will be linearly separable, and the classifying hyperplane may be equivalent to a nonlinear equation in the original space. This process is illustrated in Figure 4.1. The SVM can also accept weighted data points, shifting the classifying hyperplane farther away from high weight points.

4.2 The Classification and Regression Tree Algorithm

SVMs can become intractable when the data has very high dimensionality, as the data becomes increasingly sparse in comparison to the size of the space. A better fit for such situations is the use of decision trees. In a decision tree each node describes a rule dictating to which child node the search should proceed, based on the current situation. The decision tree is traversed from the root according to the rule in each node, and the leaf node reached contains the label of the group the tree decided upon. Each interior node of the tree has two children and a rule about some feature of the candidate. One child corresponds to the case where the rule is true for a given data item and the other to when it is false. These rules are assertions about



(a) Original data (b) Support vector machine in kernel space

Figure 4.1: A set of data is projected into a higher dimensional space by a kernel function and a line divides the two groups. Each point is a single data element, colored according to the group to which each belongs.

a characteristic for classification, or inequality statements about characteristics for regression. For instance, a tree which classifies medical patients may have a rule such as "sex is male", while a regression tree rule could be "age $>$ 40". Leaf nodes simply contain the label of a group. When a candidate is to be classified, the process starts at the root node and proceeds to the appropriate child based on whether or not the candidate adheres to the rule. This process repeats recursively until it reaches a leaf node [8].

The Classification and Regression Tree Algorithm constructs a decision tree by calculating the entropy of each possible split of each characteristic of the data which was not already used by any ancestor node. [1] Entropy is a measure of the randomness of a set of data, defined by:

$$H(X) = -\sum_{n}^{i=1} P_{x_i} log(P_{x_i}) \tag{4.1}$$

where x_i is one of the n data points in the set, and $p(x_i)$ is x_i's probability mass function. [18] This means that entropy will be maximal when a collection contains equal numbers of instances of all groups and will be zero when the collection contains

only members of a single group. Thus, the split with the lowest entropy in each new subset is the one in which the various subsets' members are most similar to one another.

The split with the smallest entropy is selected, and the new node contains a series of rules which cover that characteristic's possible values. This approach of always selecting the maximum information gain at each iteration makes it a greedy algorithm. Then, the process is repeated recursively on the child nodes, considering only that subset of the training data which would reach that node by traversing the tree. Once all splits have the same entropy value, the node is instead made a leaf node of the group that has the most representatives within the remaining data. [1]

Chapter 5: Moving Target Framework Details

A prototype system has been developed by Brian Lucas [9] to show the feasibility and performance of the ES based MT approach. It is implemented in Python version 2 and manages several virtual machines running an Apache HTTP Server™ 2.2 on Red Hat® Enterprise Linux® 5. This prototype simulates the management of physical computers.

The Moving Target system is designed to regularly and automatically reconfigure a group of computers. These machines are reconfigured based on the instruction of configurations developed by an ES. The ES is then provided feedback from the framework regarding each machine's security record, which it uses to evolve the next generation of configurations. Over time, this will lead to more secure configurations being implemented while maintaining diversity among these configurations. The regular reconfiguration will also provide a Moving Target (MT) defense in the process, due to the changes to the computer's set up caused by the reconfiguration.

5.1 System Assumptions

The system assumes that each computer to be configured will have an identical load of programs installed, be close together in network topology, and present equally attractive targets to an attacker. This increases the likelihood that any difference between observed number of security events can be traced back to vulnerabilities exposed by misconfiguration, instead of outside factors over which the system has no control. It is further assumed that the machines can safely be reconfigured regularly in a reasonable time frame, even if such a reconfiguration requires the machine to halt work or be rebooted. An office where computers can be reconfigured at night, when they would not otherwise be in use, is an example of a situation which satisfied this criteria.

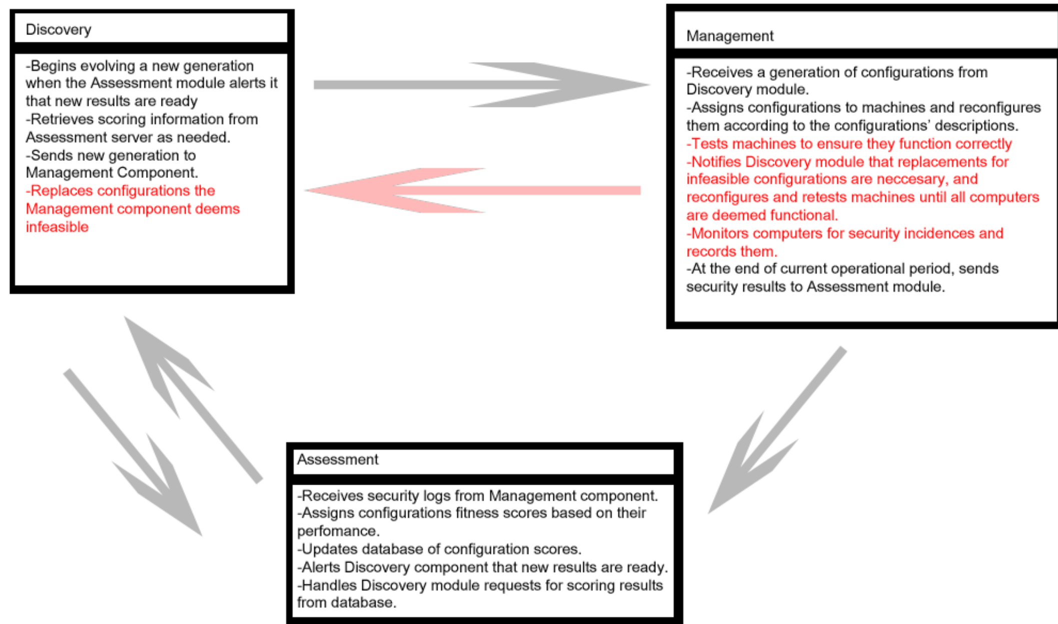


Figure 5.1: The framework consists of three separate modules along with a communication protocol to manage information sharing and commands. All listed functions are necessary for a real implementation of the system, but portions in red are not implemented in the prototype.

5.1.1 Genetic Algorithm Constraints

The previous assumptions place an unusual constraint on the ES. Most ES are intended to run for many thousands of generations in order to find a solution. However, the MTGA system can only accept a new generation of configurations from the ES after a substantial wait to allow a reasonable probability of the configured machines being subject to attack. The ES cannot create the next generation until it receives the assessment of the current generation's performance, which can only be known after that operational period is over. Thus, in order to be worthwhile, the ES must achieve notable progress within a very small number of generations.

5.2 Framework Overview

The system is divided into three communicating components, as shown in Figure 5.1. These are the discovery component, which houses the ES, the management component, which reconfigures the controlled computers, and the assessment component,

which maintains the storage of all previous data collected on configurations' performance. A generation of configurations, the set of configurations evolved by the latest iteration of the ES, passes through each in turn.

The discovery component houses the ES and is responsible for evolving the next generation of configurations. It receives a message from the assessment component that the results from the previous generation's testing are available. It then begins the EA process, requesting information from the assessment component about testing results whenever it requires them. Once it has completed an iteration, it sends the configurations to the management component. In the completed system (not the developed prototype) it may then receive a message back from the management component that some number of configurations were infeasible, in which case the discovery module will rerun the ES to generate replacements for them. Then the discovery component waits for notification from the assessment server to begin work on the next generation.

The second component is the machine manager. It takes a set of configurations from the discovery module and assigns each to to one of the managed machines, which could be physical computers in a real deployment but which are all virtual machines for testing purposes. There is a one to one correspondence between the configurations in the current generation and computers. Multiple computers will be configured identically if there is more than one instance of a particular configuration present. It then reconfigures the machines so that they adhere to the setting descriptions provided by the assigned configuration file. Although not currently implemented, the manager will then test each machine for proper functionality. A computer meant to run a web server, for instance, would fail the test if the manager could not access the web page. Computers which fail these tests would have their configurations marked infeasible, and the manager would request replacement configurations from the discovery component. Once all computers pass testing, the manager monitors the machines for security incidents and records them. Once the operational period is over, the manager sends the configurations and the security observations to the assessment component.

```
<parameter distribution="uniformOption" id="5"
preferred="-FollowSymLinks" prev=""
trueScore="AV:N/AC:L/Au:M/C:P/I:N/A:N">
-FollowSymLinks
<option>+FollowSymLinks</option>
<option>-FollowSymLinks</option>
<score>AV:N/AC:L/Au:M/C:P/I:N/A:N</score>
</parameter>
```

Figure 5.2: This XML parameter with a discrete list of options corresponds to the DISA STIG configuration guideline ID WA000-WWA052 A22.

```
<parameter distribution="uniform" id="6"
preferred="0" prev=""
feasibleRangeEnd="1" feasibleRangeStart="0"
trueScore="AV:L/AC:H/Au:M/C:N/I:N/A:N">
0
<score>AV:L/AC:H/Au:M/C:N/I:N/A:N</score>
</parameter>
```

Figure 5.3: Example XML numeric parameter with a uniform distribution in the range 0 to 1. It corresponds to the DISA STIG configuration guideline ID WA00515 A22.

```
<configuration attacks="0"
id="0" prevScore="1" score="39510" timeUp="0" trueScore="39510">
<parameter>...</parameter>
<parameter>...</parameter>
.
.
.
<parameter>...</parameter>
</configuration>
```

Figure 5.4: An XML chromosome node that represents a full computer configuration description. In addition to containing a list of parameter descriptions and their settings, it also contains data pertaining to the configuration as a whole, such as *attacks*, the number of successful attacks it suffered during the last operational period.

The final component is the assessment server. Upon receiving a set of configurations from the management unit and its associated testing data, it assigns each configuration a fitness score based on the observed security incidents. These scored configurations are stored in a database, and the discovery unit is signaled to begin creating the next generation. The assessment component then waits for either the discovery module to request the scores for a configuration or for the management unit to provide a new generation to grade.

5.3 XML Configuration Description

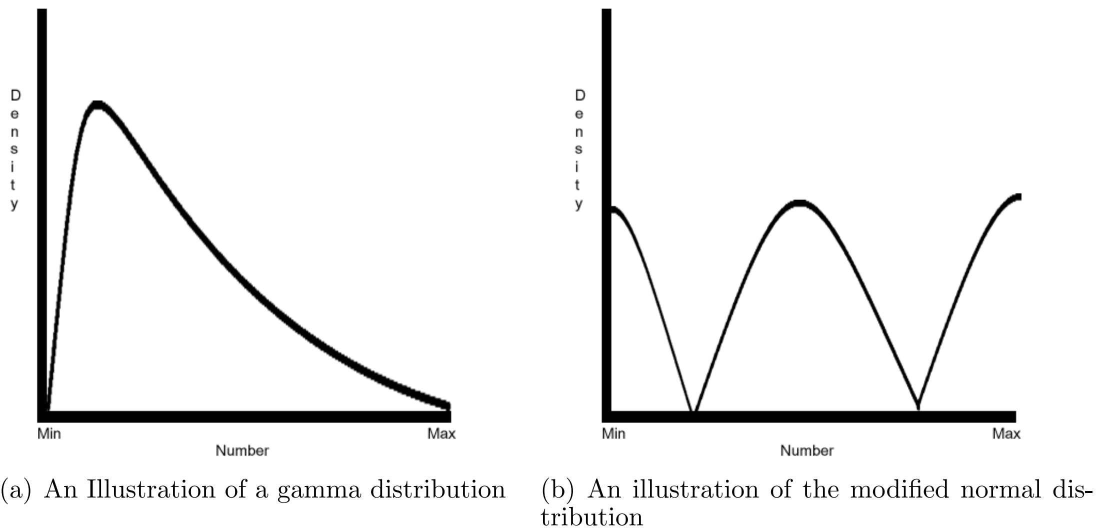
An ES requires a chromosomal representation of the candidate solutions. In the case of MT security, these candidates are computer configurations for a series of parameters the MT system is provided to control. In the framework, chromosomes are represented in XML. Each describes a computer configuration by providing a concrete setting for each parameter. Each chromosome also contains various other information about that configuration, such as a unique identification number, its score, and information about changes from its parent from the previous generation. This information is used by the MT system, instead of by the machine on which the configuration is implemented.

Two example XML parameters descriptions are shown in Figures 5.2 and 5.3. They demonstrate all the types of information which can be associated with a given parameter. The full list of parameter descriptions can be found in Appendix B.

Distribution is the parameter's type, which will described in Section 5.3.1. *id* is a unique identifier for each parameter. *Prev* is the parameter's previous setting, tracked for use in setting classification in the directed mutation system. *Preferred* is a value which does not open the parameter's associated security vulnerability, present in all parameters but only used with numeric parameters with a gamma distribution.

In addition to the previous attributes defining the parameter, the XML also contains information used in the operation of the Moving Target Genetic Algorithm (MTGA) system. *Truescore* and *score* are Common Vulnerability Scoring System (CVSS) vectors for the setting's true fitness value, as measured by the fitness function, and the fitness value reported by the simulation of the assessment server, respectively. These are maintained separately so that, when the ES is given deceptive fitness information, it is still possible to measure a configuration's true fitness.

Figure 5.4 is a configuration node, representing a full chromosome. *Attacks* is the number of observed security events in this testing period. *id* is a unique identifier for that configuration, which is identical among all configurations with the same setting for all parameters. *prevScore*, *trueScore*, and *Score* are equivalent to the attributes of the same names for parameters, except they hold the overall numerical score the



(a) An Illustration of a gamma distribution (b) An illustration of the modified normal distribution

Figure 5.5: Illustration of two sample probability functions.

configuration itself instead of CVSS vectors. *timeUp* holds the length of time a configuration has been active on a configured machine within the current generation. Each parameter node is a parameter XML representation as described earlier in this section.

5.3.1 Parameter Types

In general, a computer configuration consists of many different kinds of parameters each of a given type such as discrete options, numbers, or bit strings. In addition to each parameter's current setting, a configuration also describes the mutation rule for that parameter.

For *bit strings*, marking it as a bit string is sufficient. The *options* type requires a list of all possible values the parameter can hold, represented in XML as a series of *option* subnodes. An example *Option* parameter's XML representation is given in Figure 5.2

Numerical parameters require both a range and a distribution. An example numerical parameter's XML representation is given in Figure 5.3. The range lists the start and end points for the valid numbers which can be used for the parameter, expressed by the XML attributes *feasibleRangeEnd* and *feasibleRangeStart*. The probability distribution, defined by which of the values for numeric parameters the XML

distribution attribute contains, provides the ES information about the typical settings for the parameter. For example, a parameter which could in theory be set to any integer but is normally less than 100 could receive a gamma distribution configured to ensure most randomly chosen numbers will fall below 100. There is a default distribution for numerical parameters, and it is necessary to provide the expert knowledge to define a better fitting distribution. The currently used distributions are gamma and a modified Normal distribution [15].

The gamma distribution provides a probability density which is heavily concentrated to the lower end of the number line, as seen in Figure 5.5(a). This makes it useful for situations in which the administrator knows that standard values for the parameter are clustered at the beginning of the range.

The gamma distribution will then cause the ES to regularly mutate to values within this subrange at the beginning, while still maintaining a possibility that it will occasionally "experiment" by choosing a higher valued number. Gamma distributions are defined by a shape parameter k and a scale parameter θ. These must be provided when specifying the distribution. [15]

The other implemented distribution is a modified Normal, with a 0.5 probability of the distribution being shifted half the size of the range, with wrapping. This moves much of the distribution from the center of the range to the edges. This process is shown in Equation 5.1.

$$\begin{aligned} newValue &= (rand() + \frac{feasibleRangeEnd - feasibleRangeStart}{2}) \\ &\quad mod(feasibleRangeEnd - feasibleRangeStart) \\ &\quad + feasibleRangeStart \end{aligned} \tag{5.1}$$

This produces a likelihood function which is highly likely to select numbers at the beginning, middle, or end of the range. This is used as the default distribution, because, according to experience, values which close security vulnerabilities tend to lie in one of these areas more often than the rest of the range. This has the side expect of making those few parameters which do have safe regions far from the ends or middle unlikely to be secured. This is a side effect of its use as a default distribution in which

no information about the parameter's settings is known, and it still handles all tested parameters no worse than a uniform distribution.

Chapter 6: System Performance Measures

The problem of measuring the MTGA system's success requires several tools. A method for determining a population's diversity must be found, and a fitness function with properly measures a chromosome's resistance to attack must be defined. This fitness function must be flexible enough to accommodate simulation according to varying assumptions about the security environment in which the system will operate, such as accommodating the simulation of a limited number of attacks to provide the ES with only partial information about the true fitness score. Additionally, fitness may be insufficient to properly measure the system's success. When only parts of the chromosome are given accurate fitness evaluations in each generation, overall fitness increase will be small. In these cases, there must be a metric for measuring how well the system performed for only those parameters taken into account by the fitness function.

Techniques have been developed to measure a population's diversity and its success in using the provided information. A method for estimating a chromosome's fitness by grading consequences of its open vulnerabilities has been developed. It was used in simulations of the MTGA system.

6.1 Diversity Measurements

Diversity for a generation can be measured based on the average pairwise Hamming distance of the configurations within it. This is a genotypic description of diversity, not a phenotypic one. A genotype is the genetic representation of a chromosome, while a phenotype is how an organism's genotype is actually expressed as a set of characteristics. For example, a computer which appears to be a web server to an attacker's reconnaissance would be phenotypically different than one which appeared to be a personal computer. It is possible for two configurations to have different genotypes, by having different settings for some parameters, while having the same phenotype, by appearing identical to an adversary and being vulnerable to the same

set of exploits. Thus, Hamming distance does not measure how much variance an attacker would actually detect in the system. It also does not take into account whether spatial or temporal diversity is achieved, as it does not directly measure the number of divergent attacks the pairs of chromosomes are vulnerable to or the divergences between a single machine's configuration over multiple generations.

These characteristics make it ideal for assessing the ES's effects on setting diversity, as it measures only the chromosomal representations operated upon by the ES. However, it is computationally expensive to measure this value directly for large populations, as it involves comparing each setting of each chromosome to the same parameter's setting in each other chromosome. Thus, sampling is used to reduce the number of comparisons made. Some constant number, k, of chromosomes are randomly chosen, and the pairwise Hamming distances of the chromosomes in this randomly chosen subset of the current generation are measured. Experiments associated with this research in which the true and sampled Hamming distances with 15 sampled chromosomes were compared, have shown that this sampling-based estimate produces a close approximation of the true pairwise Hamming Distance. This technique is called the sampled average pairwise Hamming distance.

6.2 CVSS Vector Vulnerability Classification

Testing how well the system improves security is time consuming. For example, if one assumes that each operational period would require at least a day to provide a reasonable amount of time for the machines to be attacked, it would take over a month for each 40 generation run of the system. It would also be totally dependent on whether attackers choose to target the managed machines and which exploits they would attempt. For example, attackers would be unlikely to attempt a distributed denial of service (DDOS) attack against the machines, as they are not serving any content worth targeting.

These complications would prevent vulnerabilities to DDOS attacks from ever being exploited. This kind of variety in attacks can be modeled by the CIA model, in which exploits are classified based on whether they impact the Confidentiality,

Integrity, or Assurance of the targeted machine. The CIA model has been combined with descriptions of the attack vector for a given exploit in order to create the Common Vulnerability Scoring System (CVSS). CVSS vectors thus serve as a measure of the threat posed by a given exploit. Thus, actual real world testing by exposing the machines to the network is simulated by assigning CVSS vectors which reflect the security consequences for each possible parameter setting. These CVSS vectors are assigned by expert[1] scoring of parameters, and the known attacks they can create a vulnerability to.

A CVSS vector is defined by six fields, each of which can take one of three values. These fields are *Access*, *Complexity*, and *Authentication*, which measure the difficulty of the attack, and *Confidentiality*, *Integrity*, and *Assurance*, which measure the severity of the attack's result. *Access* describes where the attack can be launched from as either local, only on the machine itself; an adjacent network, only from the same LAN; or a network, from anywhere. *Complexity* describes the level of expertise needed to execute the attack as either low, medium, or high. *Authentication* describes the number of times the attacker must present the correct credentials in order to complete the attack as either none, single, or multiple. *Confidentiality* measures how much private information the attack releases to the attacker, *integrity* how much information the attacker is able to change, and *assurance* the attacker's ability to deny the machine's resources to legitimate users. All three are classified as either none, partial, or complete. [12] Secure settings are assigned the CVSS vector containing the most favorable measurements for *Access*, *Complexity*, and *Authentication*, and *none* for *Confidentiality*, *Integrity*, and *Assurance*.

A CVSS vector can be seen in this example:

AV:N/AC:L/Au:N/C:C/I:C/A:N.

Each of the six fields is filled with an appropriate value for the sample attack, an SQL injection. An SQL injection is simple, requiring only submitting SQL code to a unsecured form (AC:L). It can be done over the internet (AV:N) and requires no

[1]This expert score was provided by Dr. Errin Fulp, Scott Seal, and Bryan Prosser.

credentials (Au:N). It's effects can include divulging database tables to the attacker or overwriting or deleting database entries (C:C/I:C/A:N). These are all reflected in the various fields of the CVSS vector.

6.3 Attack Simulation For Fitness Estimation

For testing purposes, the attacks launched against a computer configured according to a particular chromosome are simulated. In this thesis, it is assumed that attackers will attempt easy exploits more than difficult ones, and more will attempt high impact attacks than low impact ones. Thus, the CVSS vector can be converted into a number which approximates how many attacks a machine will face if it leaves a particular vulnerability open. This is done by assigning each CVSS field's possible values a bad, neutral, or good categorization. Bad categorization are assigned to values which make an attack easy to perpetrate (such as None for Authentication) or attractive to an attacker (Complete for Integrity), while good categories are given to the opposite. Bad values are assigned a worth of 1 point, neutral values 10, and good values 100. Then fitness is the sum of all six CVSS vector categories for all parameters. Parameter chains, combinations of parameters which collectively open or close a vulnerability, can also be used. Each of these receive its own CVSS score, which is simply added to the total. Alternatively, this additional information can be removed, with it being assumed that the vulnerability is exploited exactly once if it exists. In this case, vulnerable settings receive a score of 0 and secure settings a value of 600.

A chromosome's fitness is then defined as the sum of the numerical scores for all of its parameters. This is the aggregate score used to by the ES as a chromosome's fitness value.

However, it is not reasonable that all possible exploits will be attempted during every operational interval. A constant number, p, of attacks per generation can be defined, and an attack profile created by randomy selecting p parameters which the attacker will attempt to exploit. It is possible that diversity parameters, ones which have no associated exploit, will be selected for the attack profile. This corresponds to fewer attempted attacks than normal occurring during that operational cycle. Only

those parameters in the attack profile will be scored correctly. All other parameters will be scored as if they were secure.

6.4 Adaptation Rate

Fitness gain per se is not an accurate measurement of the system's performance. Rather, the MTGA system's goal is to fully exploit the information gained during the testing for each operational period. The adaptation rate is a measurement of how well the ES achieves this. The attacks that were in that generation's attack profile are saved, producing a record of all attacks the system has faced. Then, this cumulative attack profile is replayed against subsequent generations, and the number of these attacks which succeeded is measured. The adaptation rate is then $1 - \frac{\text{the number of attacks which would have succeeded this generation}}{the\ number\ of\ computers * the\ number\ of\ attacks\ launched.}$. Thus, an adaptation rate of 100% means every observed attack has had its associated vulnerability closed. Meanwhile, 0% implies that all machines are vulnerable to all previously seen attacks.

Chapter 7: Evolutionary Strategies For Configuration Generation

Two different ES have been developed to implement the MTGA strategy. Both the MTGA and beam search are designed to operate within the discovery module of the MT framework outlined in Chapter 4. They are intended to produce diverse, more secure sets of configurations, given only a few generations of feedback from the assessment unit. An implementation of these strategies is presented in Appendix A.

7.1 The Moving Target Genetic Algorithm

The MTGA is a GA designed specifically to operate within the context of the MT configuration problem. It features specialized mutation operators which are capable of handling the varying types and domains found among the parameters of a configuration. It has been refined to promote fitness gain by experimentally finding a satisfactory set of GA constants and operators. The MTGA is not a Simple Genetic Algorithm (SGA), but it follows the standard procedure of a GA, as outlined in Algorithm 1 (page 9), using the selection, crossover, and mutation operators to produce the next generation from the current one.

7.1.1 Selection

Several different types of selection have been implemented and analyzed for the selection step, corresponding to Lines $5-6$ in Algorithm 1. Roulette Wheel and Tournament Selection were implemented and tested, as were other techniques for enhancing the selection step, such as selection type blending and elitism. Stochastic universal sampling was also investigated, but rejected.

Roulette Wheel Selection

Roulette wheel selection, also known as fitness proportional selection, is the simplest kind of selection. A range is made between 0 and the sum of chromosomes' fitness

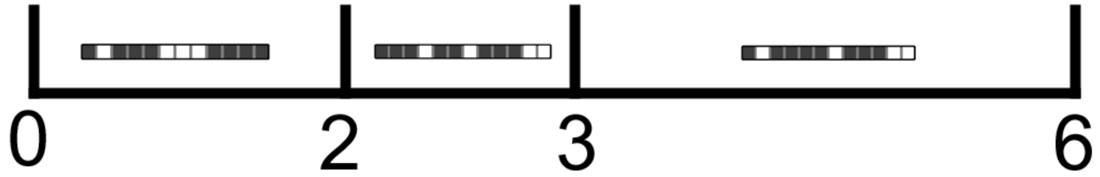


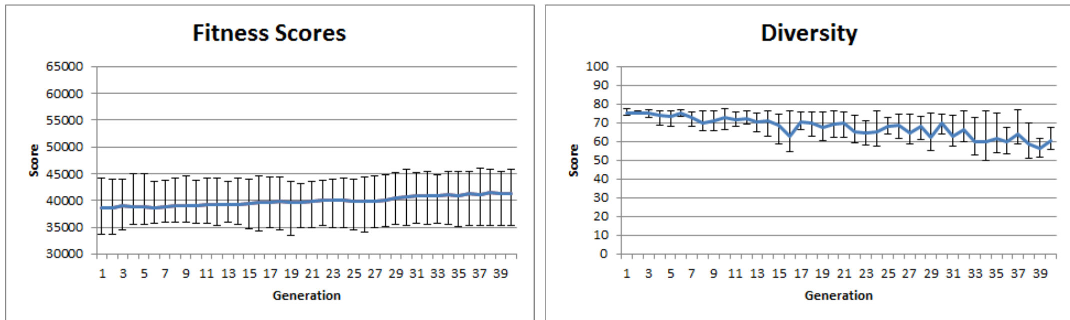
Figure 7.1: An illustration of Roulette Wheel Selection. The first chromosome has fitness 2, the second 1, and the last 3. A random number is chosen, and the chromosome in the range it falls within is selected. In this case, if the number is less than 2, the first chromosome is selected. If it is at least 2 and less than 3, the second is selected. Otherwise, the last chromosome is selected.

scores, and each chromosome is assigned an interval in the range with a size equal to its fitness. A random number is generated within the range, and whichever chromosome owns the corresponding interval is selected. This gives each chromosome a chance to be selected proportional to its fitness score. Roulette Wheel Selection is described in Lines $7-13$ of Algorithm 2 (page 8) and illustrated in Figure 7.1.

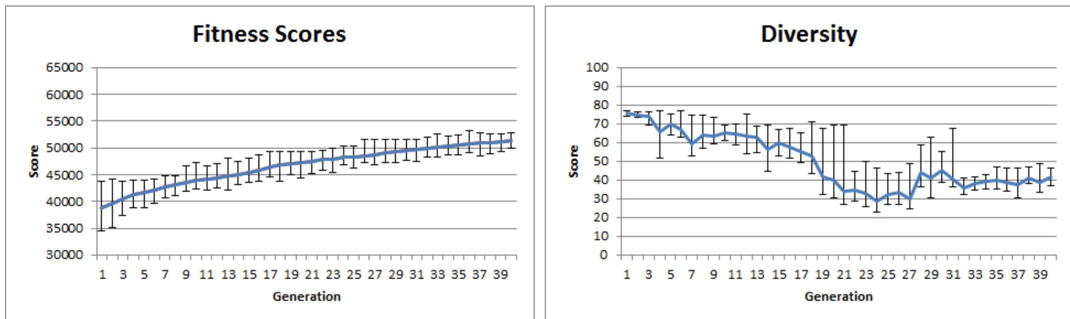
As can be seen in Figure 7.2(a), roulette wheel selection performs poorly, because it achieves almost no fitness gain over the 40 generations. It does, however, maintain high diversity.

Stochastic Universal Sampling

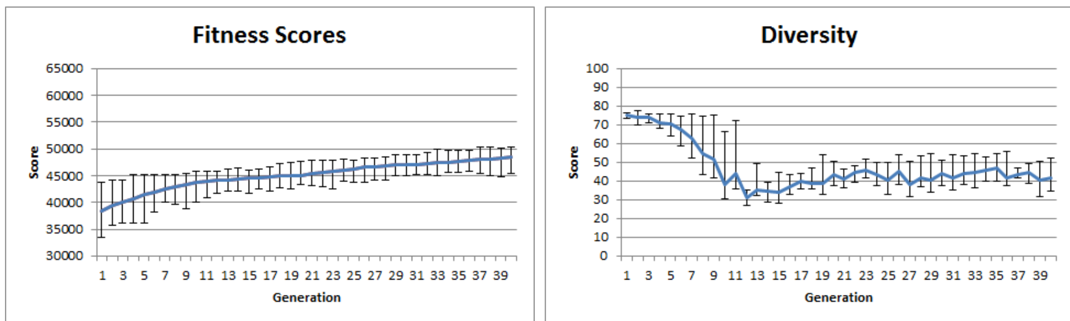
This strategy builds a range based on fitness scores as roulette wheel selection does, and selects a single random point less than the sum of all fitness scores in the generation. Then, instead of selecting more random points as in roulette wheel selection, it selects one evenly spaced point for every chromosome being selected. For example, if the total fitness of all chromosomes is 10 and the first randomly selected point out of 5 chromosomes to be selected is 7, it will then select 9, 1, 3, and 5. These points then determine which chromosome is selected using the same method by which roulette wheel selection maps random points to chromosomes. This approach was rejected as the MT problem does not exhibit the large disparities between chromosomes in the same generation that stochastic universal sampling was intended to correct for. Thus, it was never implemented.



(a) Roulette Wheel Selection fitness and diversity

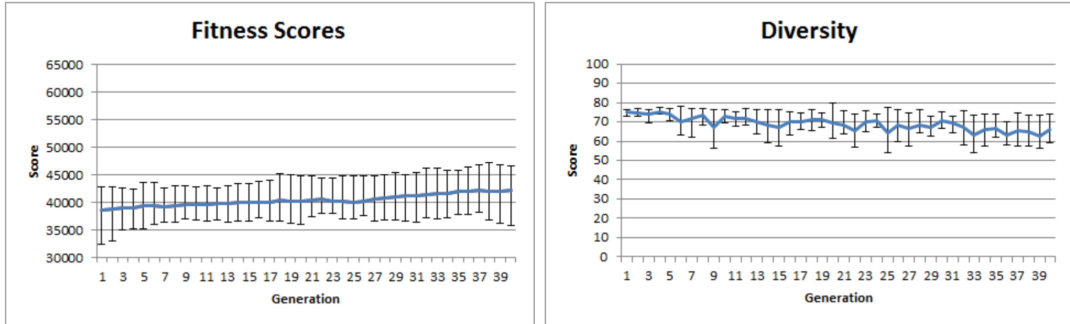


(b) Size 4 Deterministic Tournament fitness and diversity

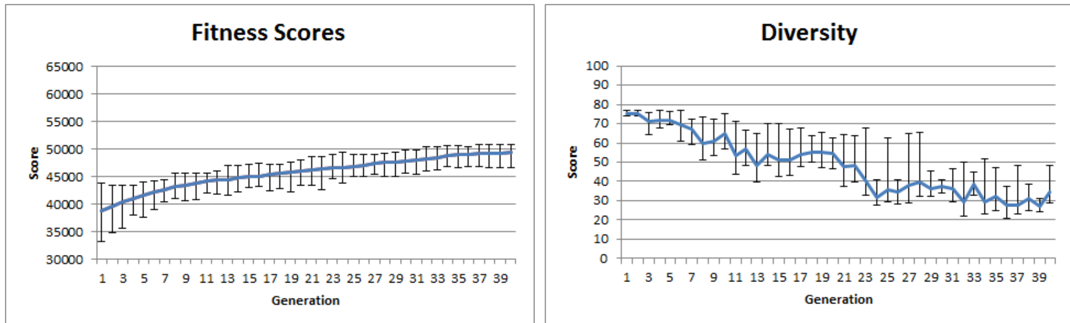


(c) Blended selection fitness and diversity

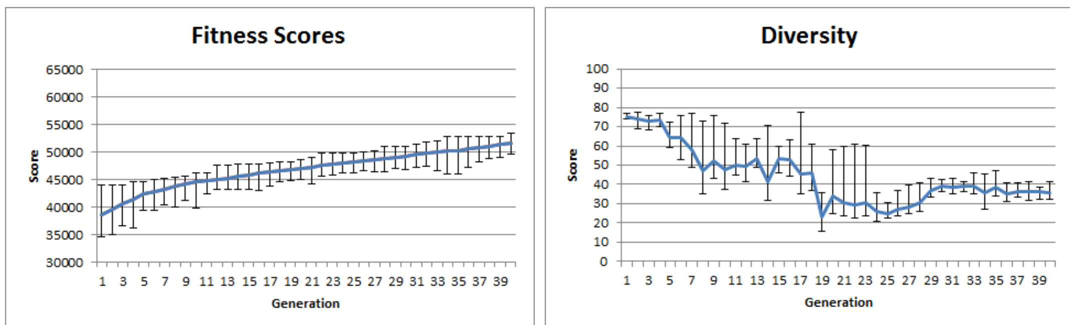
Figure 7.2: Each graph measures the average fitness score or mean sampled average Hamming Distance of each generation for a single representative run of the ES MT system. The bars indicate the highest and lowest scoring members of the population, or the samples with the highest or lowest sampled average Hamming Distance. 7.2(a) shows the results of Roulette Wheel selection, 7.2(b) shows deterministic tournament selection with two rounds, and 7.2(c) shows the .25/.75 blended selection. The other details of the runs are identical: 140 chromosomes over 40 generations. $p_c = 0.05$. $p_m = 1$. Genes modified per mutation = 2. Two point crossover type. No directed mutation. Hamming distance measured only for a subset of 15 randomly selected configurations.



(a) Nondeterministic Tournament fitness and diversity



(b) Deterministic Tournament with 0.5 Probability of Elitism fitness and diversity



(c) Deterministic Tournament with Elitism fitness and diversity

Figure 7.3: Each graph measures the average fitness score or mean sampled average Hamming Distance of each generation for a single representative run of the ES MT system. The bars indicate the highest and lowest scoring members of the population, or the samples with the highest or lowest sampled average Hamming Distance. 7.3(a) shows the results of nondeterministic Tournament Selection with two rounds, 7.3(b) shows deterministic Tournament selection with two rounds and 0.5 probability of eight elitist chromosomes chosen, and 7.3(c) shows deterministic tournament selection with two rounds and 1.0 probability of eight elitist chromosomes chosen. Other details are identical to those in Figure 7.2.

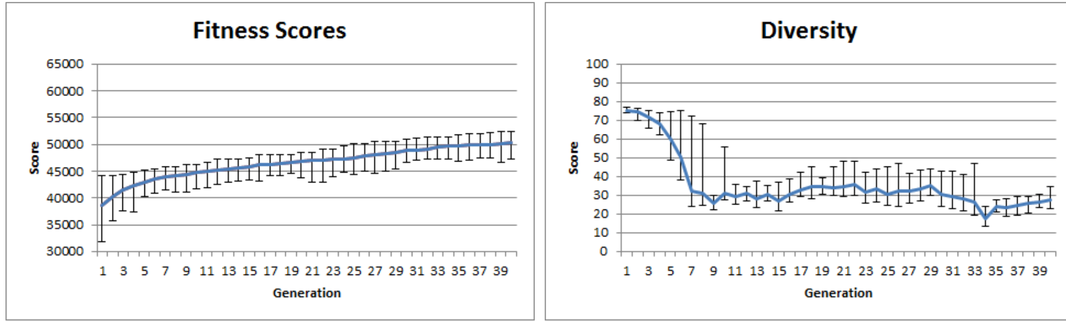


Figure 7.4: Each graph measures the average fitness score or mean sampled average Hamming Distance of each generation for a single representative run of the ES MT system. The bars indicate the highest and lowest scoring members of the population, or the samples with the highest or lowest sampled average Hamming Distance. This figure shows the results of a dynamically altered blending of size four deterministic tournament and roulette wheel selection. Other details of the run are identical to those in Figure 7.2

Tournament Selection

In tournament selection, $2k$ chromosomes, for some integer k, are chosen via some other selection method. The MTGA uses roulette wheel selection to populate the $2k$ elements of the tournament. Afterwards, the chromosomes are paired off, and one of the pair is declared loser of the round and is removed from the tournament. In a nondeterministic tournament, each chromosome has a probability to win a round equal to $\frac{chromosome's\ fitness}{the\ sum\ of\ the\ two\ chromosomes'\ fitness}$. In a deterministic tournament, the chromosome with the higher fitness automatically wins the round. The process repeats for k rounds, until only one chromosome remains, which is the one selected. Tournament selection is described in Algorithms 3 and 4 and illustrated in Figure 7.5.

As can be seen in Figure 7.2, deterministic tournament selection outperforms all other selection methods in terms of fitness gains, while maintaining moderately high diversity among the population. In Figure 7.3, nondeterministic tournament has only very small fitness gain but high diversity.

Selection Type Blending

It is not necessary to exclusively use one kind of selection. The MTGA associates a probability with a number of selection strategies. Every time the blended selection operator is called, it determines, according to the specified probability distribution, which selection operator implementation to use to select a single chromosome. This

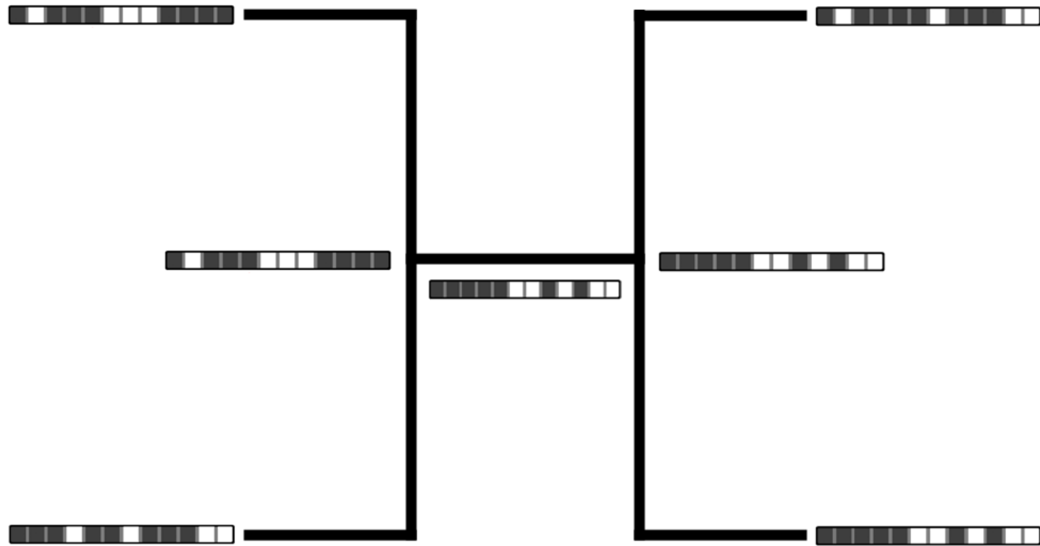


Figure 7.5: An illustration of size 4 Tournament selection. Four chromosomes are chosen via Roulette Wheel Selection. The two on the right are compared, and the bottom chromosome wins due to its higher fitness and advances to the next round. A similar process occurs on the left, except the top chromosome wins. Finally, the two remaining chromosomes are compared. The chromosome on the right wins the final round and is selected.

```
1  contestantA = RouletteWheelSelection (currentgeneration)
2  contestantB = RouletteWheelSelection (currentgeneration)
3  contestantC = RouletteWheelSelection (currentgeneration)
4  contestantD = RouletteWheelSelection (currentgeneration)
5  if contestantA.fitness > contestantB.fitness then
6  |   finalistA = contestantA
7  else
8  |   finalistA = contestantB
9  end
10 if contestantC.fitness > contestantD.fitness then
11 |   finalistB = contestantC
12 else
13 |   finalistB = contestantD
14 end
15 if finalistA.fitness > finalistB.fitness then
16 |   return finalistA
17 else
18 |   return finalistB
19 end
```

Algorithm 3: Deterministic Size Four Tournament Selection

```
1  contestantA = RouletteWheelSelection (currentgeneration)
2  contestantB = RouletteWheelSelection (currentgeneration)
3  contestantC = RouletteWheelSelection (currentgeneration)
4  contestantD = RouletteWheelSelection (currentgeneration)
5  prob = Random() mod (contestantA.fitness + contestantB.fitness)
6  if prob ≤ contestantA.fitness then
7  |  finalistA = contestantA
8  else
9  |  finalistA = contestantB
10 end
11 prob = Random() mod (contestantC.fitness + contestantD.fitness)
12 if prob ≤ contestantC.fitness then
13 |  finalistB = contestantC
14 else
15 |  finalistB = contestantD
16 end
17 prob = Random() mod (finalistA.fitness + finalistB.fitness)
18 if prob ≤ finalistA.fitness then
19 |  return finalistA
20 else
21 |  return finalistB
22 end
```

Algorithm 4: Nondeterministic Size Four Tournament Selection

process is shown in Algorithm 5, which blends roulette wheel and tournament selection.

These blended probabilities are not necessarily static and can optionally be changed dynamically during operation in response to situations which arise in the MTGA. The probability is adjusted when average diversity is reduced by $\frac{\textit{the number of parameters}}{\textit{the number of total generations}}$ within a single generation, the probability of using tournament selection is decreased by 0.05, and the probability for roulette wheel selection is raised correspondingly. This is intended to back off the more aggressive selection type, so that a higher diversity can be maintained.

As can be seen in Figure 7.2(c) a .25/.75 blend between Roulette Wheel Selection and nondeterministic Tournament Selection with two rounds leads to slightly lower fitness gain than tournament selection while also somewhat hastening the steep fitness decline observed in tournament selection. In Figure 7.4, the dynamically blended selection type did not significantly differ from the static blended selection.

Elitism and Steady State Behavior

Steady state behavior is the retention of the highest scoring chromosome(s) from the current generation into the next. This can be a desirable characteristic of an ES, because it guarantees relatively good solutions will remain within the population. Tournament selection encourages steady state behavior, but does not guarantee it. This is because tournament selection considers multiple chromosomes and only allows the one with the highest fitness to be selected for the next generation. This makes it much more likely that configurations with relatively high fitness will be selected, but it is still possible that the most fit of all will still never be chosen to enter the tournament and thus remain unselected.

Elitism enforces steady state behavior. A constant number, k, of chromosomes is chosen and, at each generation, k of the highest scoring chromosomes from the current generation are automatically placed within the next, bypassing the normal chances for crossover and mutation. The rest of the generation is populated with selection operators as described previously. It is also possible to associate elitism with a probability,

so that it only occurs in some generations. Elitism is demonstrated in Algorithm 6. As seen in Figures 7.3(b) and 7.3(c), elitism produces a slight improvement in fitness scores and a small decrease in diversity.

7.1.2 Crossover

Several different types of crossover have been implemented. When the GA performs crossover, it determines if each chromosome in the new generation is to experience modification. This is controlled by the constant p_c, the probability of crossover. If a chromosome is determined to be subject to crossover, a new chromosome is selected from the current generation to be its partner. After crossover, the partner is discarded.

```
1 prob = Random (0,n)
2 if prob < tournamentRate then
3 |  return TournamentSelection(currentgeneration)
4 end
5 return RouletteWheelSelection (currentgeneration)
```

Algorithm 5: Roulette Wheel and Deterministic Size 4 Tournament Selection Type Blending

```
1 prob = Random (0,n)
2 if prob < elitismRate then
3 |  Add the elitismCount highest scoring chromosomes of currentgeneration to
  |  nextgeneration Call (n - elitismCount) times and add the results to
  |  nextgeneration
4 else
5 |  Call Select () n times and add the results to nextgeneration
6 end
```

Algorithm 6: Elitism

Allele Permutation

Many crossover operators tend to give the resulting child both of two settings from a single parent, if those two parameters are located near to one another in the chromosomal representation. This can be exploited by placing related features near to each other. This increases the odds that trait combinations with above average utility will survive crossover. However, no such relationships between parameters can be assumed to be known beforehand for the MT problem. Thus, unrelated parameters will

be grouped together by the MTGA. This problem of settings propagating throughout the population only because they happen to be located next to a highly fit setting can be solved by taking a random permutation of parameters each time the crossover operator is called. (Since individual genes are referenced by searching for *id*s anyway, this adds only the generation of l random numbers and then one table lookup per gene crossed over.) Crossover then proceeds as if the chromosome were ordered according to this permutation instead of the real one. Since two parameters are unlikely to be close to each other over many different permutation, this prevents poor or neutral settings from being propagated throughout the population merely because they are adjacent to a good setting.

For example, normally one point crossover would ensure that the child receives both parameter 7 and parameter 8 from a single parent, unless the crossover point is exactly between 7 and 8. However, with permutation, it is roughly equally likely to give the child both from a single parent or one from each.

```
1  parentA = Select (currentgeneration)
2  parentB = Select (currentgeneration)
3  pointA = Random (0, l)
4  pointB = Random (0, l)
5  if pointA > pointB then
6  |  swap the values of pointA and pointB
7  end
8  for i, pointA ≤ i ≤ pointB do
9  |  parentA [i] = parentB [i]
10 end
11 add parentA to nextgeneration
```

Algorithm 7: Two Point Crossover

N-Point Crossover

There are several diverent types of crossover described by the number of relevant crossover points. In one point crossover, a random gene is chosen and it and all alleles after it are swapped. In two point crossover, two random alleles are chosen, and all alleles between them, inclusive, are swapped. In three point crossover, a third parent is selected via the selection operator. If the first two parents have the same

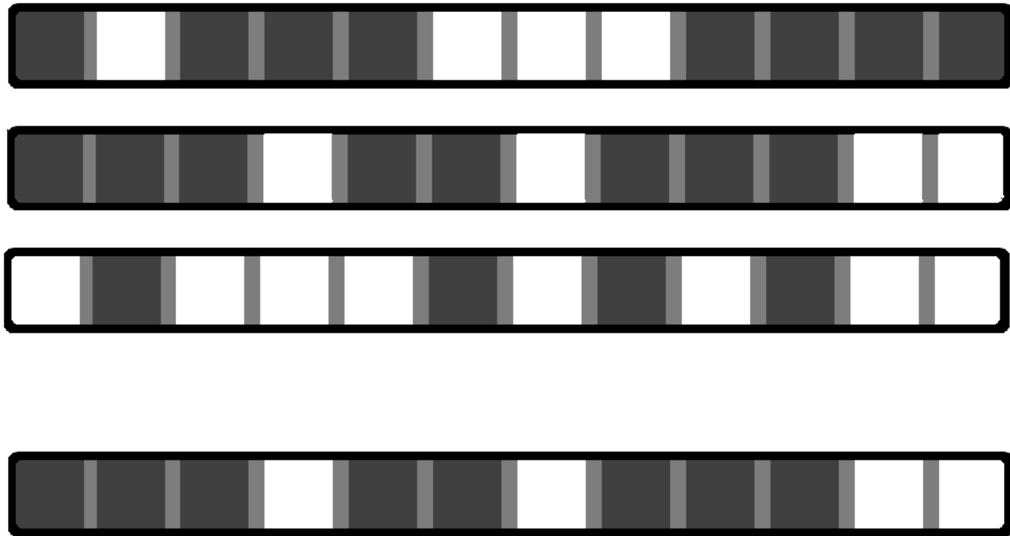


Figure 7.6: An illustration of three point crossover. When the top two parents feature the same setting for a gene, that setting is placed in the child. Otherwise, the child inherits the third parent's value.

setting for a parameter, that setting is inherited by the child. Otherwise, the child inherits the third parent's setting.

One point crossover is described in Algorithm 2 Lines 22 – 26 (page 9) and illustrated in Figure 3.1 (page 13). Two point crossover is described in Algorithm 7 and illustrated in Figure 7.7. Three point crossover is described in Algorithm 8 and illustrated in Figure 7.6. As can be seen in Figure 7.8, there is little difference in either fitness or diversity amoung the three crossover types.

Uniform Crossover

In uniform crossover, some constant number of alleles is defined. When crossover occurs, that many alleles are chosen at random without replacement, and these are swapped between the parents.

Uniform crossover is detailed in Algorithm 9 and illustrated in Figure 7.10. As seen in Figure 7.9, uniform crossover is not significantly different than any of the other implemented crossover types in fitness or diversity.

1 **for** *each gene g* **do**
2   **if** parentA *[g] !=* parentB *[g]* **then**
3   parentA [g] = parentC [g]
4   **end**
5   add parentA to nextgeneration
6 **end**

Algorithm 8: Three Point Crossover

1 points = list of pointCount distinct numbers from the range 1 to the number of genes **for** *i in* points **do**
2   parentA [i] = parentB [i]
3 **end**
4 add parentA to nextgeneration

Algorithm 9: Uniform Crossover

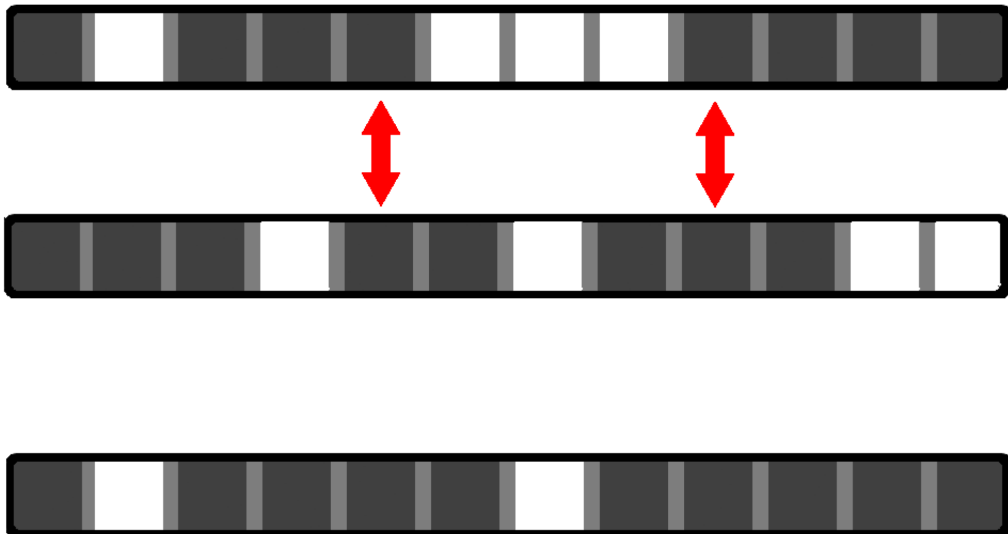
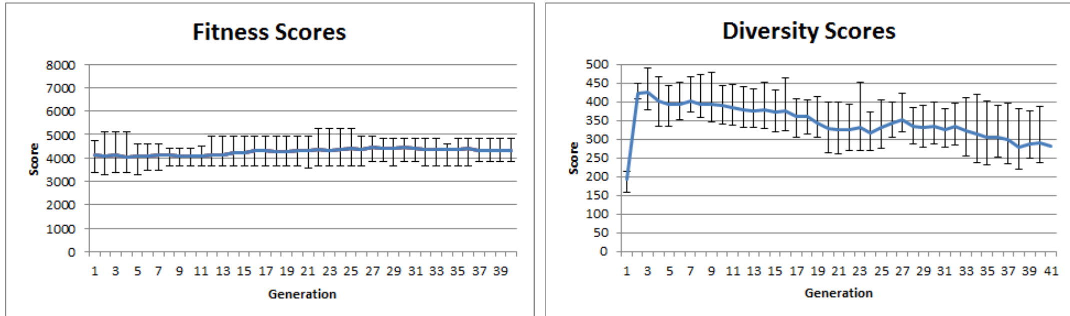
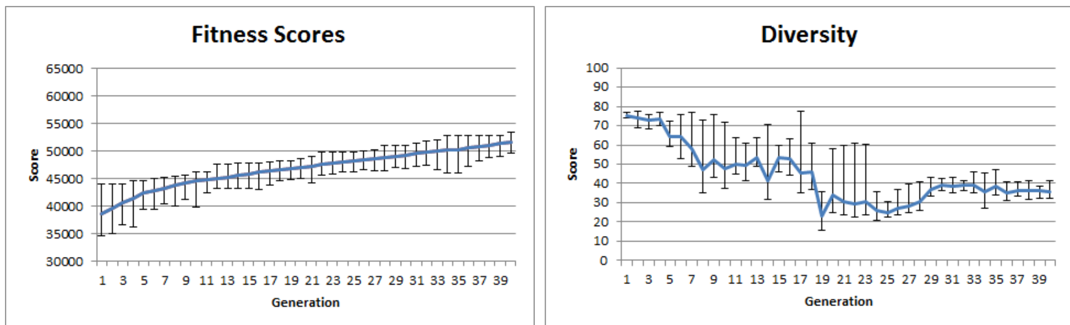


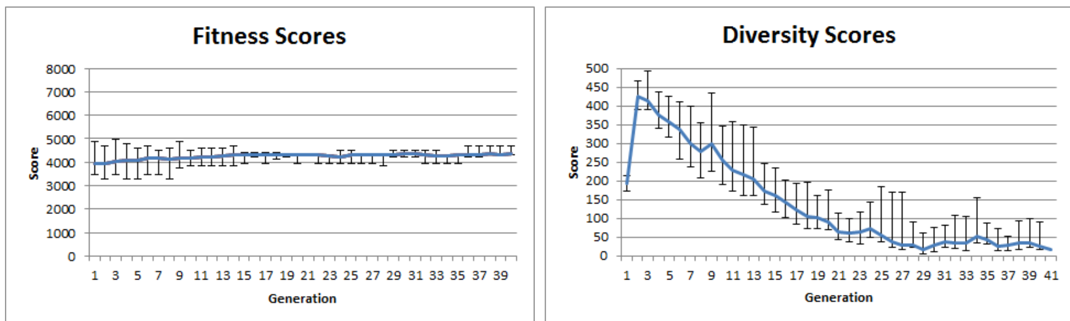
Figure 7.7: An illustration of two point crossover. Two genes are selected, and all genes between them are exchanged.



(a) 1 point crossover fitness and diversity



(b) 2 point crossover fitness and diversity



(c) 3 point crossover fitness and diversity

Figure 7.8: Each graph measures the average fitness score or mean sampled average Hamming Distance of each generation for a single representative run of the ES MT system. The bars indicate the highest and lowest scoring members of the population, or the samples with the highest or lowest sampled average Hamming Distance. 7.8(a) shows the results of 1 point crossover, 7.8(b) shows 2 point crossover, and 7.8(c) shows 3 point crossover. The other details of the runs are identical: 140 chromosomes over 40 generations. $p_c = 0.05$. $p_m = 1$. Genes modified per mutation = 2. Deterministic tournament selection with two rounds. No directed mutation. Hamming distance measured only for a subset of 15 randomly selected configurations.

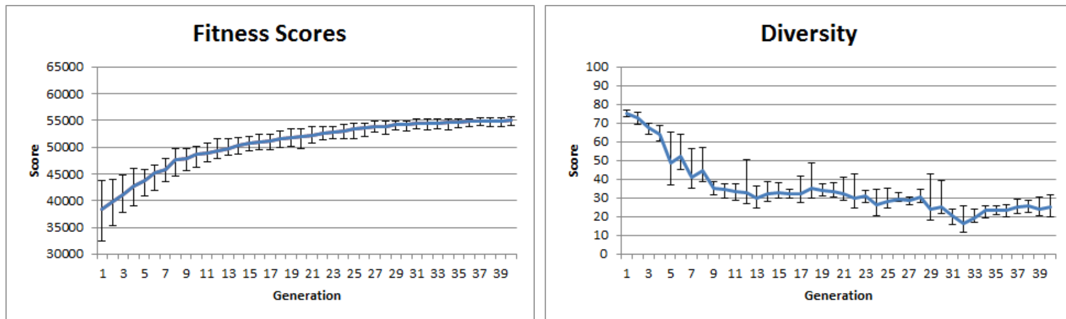


Figure 7.9: Each graph measures the average fitness score or mean sampled average Hamming Distance of each generation for a single representative run of the ES MT system. The bars indicate the highest and lowest scoring members of the population, or the samples with the highest or lowest sampled average Hamming Distance. This figure shows the results for uniform crossover. Other details of the run are identical to those for Figure 7.8.

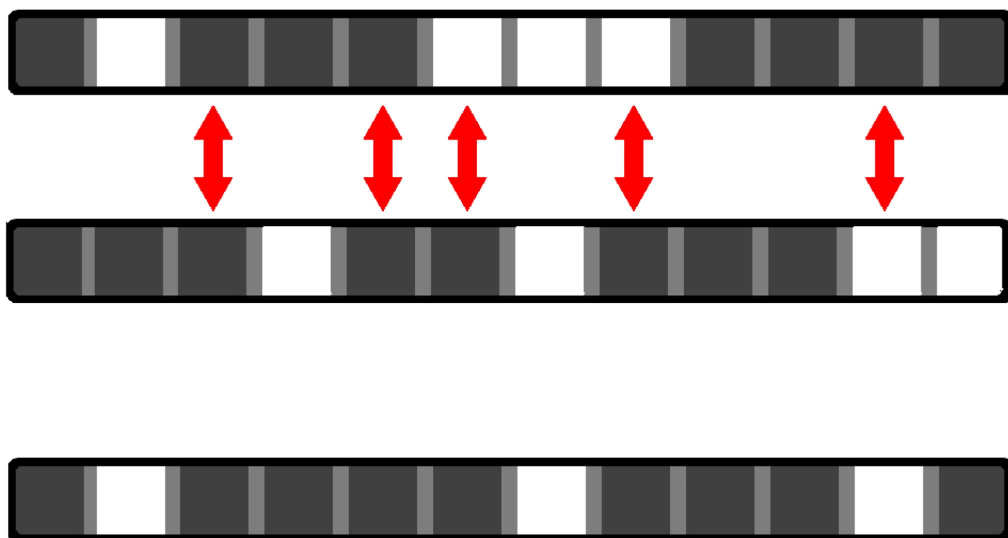


Figure 7.10: An illustration of uniform crossover. The marked locations are those which were chosen for crossover, and the parents' values in those places are swapped to produce the child.

7.1.3 Parameter Value Mutation

Each of these different parameter types has its own type specific mutation operator. Bit vectors are mutated by flipping a random bit. Option parameters have an option chosen at random from the associated list of valid settings. Numerical parameters have a number randomly chosen using the associated probability density function. By default, the distribution of this random selection for numerical parameters is the modified normal distribution described in Section 5.3.1 (28). This provides a high likelihood that very low numbers, very high numbers, or numbers close to the range's median will be selected. This is motivated by the observation that there are many parameters which have very large ranges, but a very small set of low numbers which close a security vulnerability. This distribution provides a higher chance that one of these safe numbers will be chosen, without any prior knowledge of what these numbers are. Other distributions can be provided if typical selections for that parameter are known. One such alternate distribution has been implemented: the gamma distribution.

When the algorithm performs parameter value mutation, a determination is made whether or not to modify each chromosome. There is a set constant, p_m, which determines the probability mutation will be applied to a given chromosome. Another constant controls the number of settings which are to be changed per mutation. If a chromosome is selected for mutation, a parameter is selected randomly, and its associated mutation operator is called. The MTGA then checks to see if the mutation actually produced a change, or if the operator chose the same value the parameter originally held. This process is repeated until the correct number of changes have occurred, ignoring mutations for which the gene value is the same as before the mutation operator was applied.

7.2 The Moving Target Beam Search

A beam search has also been developed for comparison with the GA. The beam search begins with a population of randomly generated configurations. A constant beam width is arbitrarily defined, and the $\frac{generationSize}{beamWidth}$ chromosomes with the highest

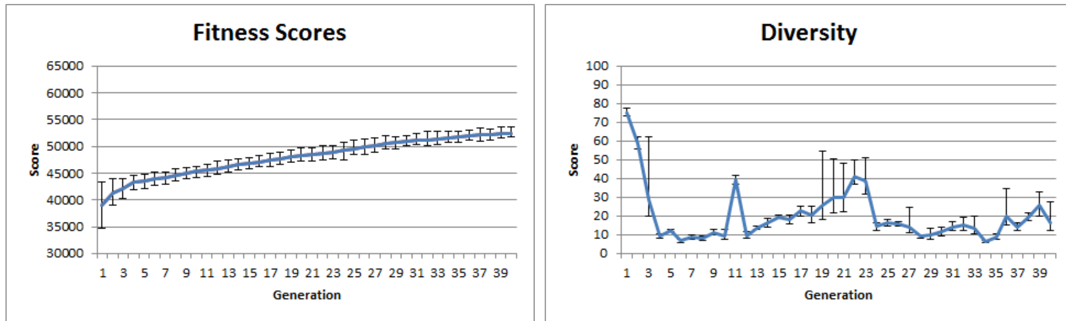


Figure 7.11: Each graph measures the average fitness score or mean sampled average Hamming Distance of each generation for a single representative run of the ES MT system. The bars indicate the highest and lowest scoring members of the population, or the samples with the highest or lowest sampled average Hamming Distance. This shows the results of the MT Beam Search, with no directed mutation and Hamming distance measured only for a subset of 15 randomly selected configurations.

fitness scores are chosen. A number of copies of this set equal to the beam width are inserted into the next generation. Then, each chromosome undergoes mutation. The process repeats after this new generation is scored.

As can be seen in Figure 7.11, Beam Search did as well as the GA in evolving high fitness chromosomes, but with somewhat lower diversity.

Chapter 8: Machine Learning Techniques for Domain Restriction

The MT configuration discovery problem differs from instances of the class of optimization problems for which GAs are commonly used in several ways. The greatest hurdle to solving the MT problem is not the difficulty of optimizing the fitness function, as although it is theoretically difficult, experience has shown that it can be closely approximated by a fairly simple function. The configuration fitness function will have no non-global local maxima unless there exists some vulnerability which can only be secured by opening another vulnerability. The time required to create a generation is normally important to an ES, because time spent per generation reduces the number of total generations the algorithm can run in a given amount of time. However, in the MT configuration problem this is unimportant in comparison to the amount of fitness gain per generation. In the MTGA poor configurations are actually implemented, leaving machines with unsecured vulnerabilities. This is in contrast to a normal GA, as the low fitness chromosomes will normally only slow the search process. Information about what the GA has learned and why it created the chromosomes in each generation would be very useful in understanding security threats.

These observations can be addressed by combining machine learning techniques with the ES. These techniques speed the GA's fitness gain by forcibly removing poor chromosomes from the population and preventing the reimplementation of settings which the GA previously learned to be insecure and discarded. The rules the GA learn through these techniques are simple enough that they can be easily exported and understood by humans.

Two such machine learning schemes have been developed for the MTGA. One, the temporal classifier, classifies settings based on correspondences between observed changes in fitness and changes to parameter settings. The other, the spatial classifier instead compares different configurations within the same generation to understand

why some were vulnerable to certain attacks while others were immune.

An implementation of these strategies is presented in Appendix A.

8.1 The Temporal Classifier

In order to classify parameter settings based on their observed impact, settings' effect on chromosome fitness must be estimated. During MTGA crossover, the parent which contributed the most alleles is defined as the child's ancestor. When the current generation is returned with fitness scores, the changes between each parameter for each chromosome are examined. The classifier collects information about each setting to create a historical tally, a numerical estimate of the setting's fitness. If the child chromosome adopted a setting of Y for a given parameter, where its ancestor had a setting of X, the signed difference between the child's fitness and the ancestor's fitness is added to the historical tally for X, and subtracted from the historical tally for Y. This accurately reflects the observed change between generations, as X is rewarded and Y penalized if switching from X to Y resulted in a fitness improvement. The opposite holds true if switching from X to Y resulted in a fitness decrease. Thus, each setting's historical tally is an estimate of its effect on fitness. This process is illustrated in Algorithm 10.

```
1 for each chromosome c in currentgeneration do
2     for each parameter p do
3         history [c.p] = c.fitness - c.ancestor.fitness history [c.ancestor.p] =
          c.ancestor.fitness - c.fitness
4     end
5 end
```

Algorithm 10: The Calculation Of Settings' Historical Fitness Impact Tallies.

8.1.1 Parameter Domain Mutation

Parameter domain mutation, as the name suggests, is a mutation operator which permanently restricts the domain of a configuration parameter. Its goal is to remove settings from consideration after a great deal of evidence suggesting the setting has low fitness has been collected. Machine learning techniques are used with the historical

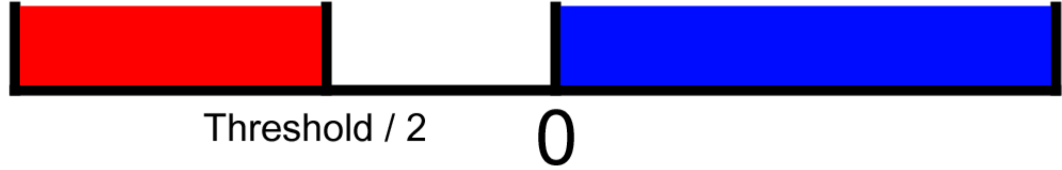


Figure 8.1: A number line of possible historical tally scores. The data with tallies in the red region is classed as insecure. The blue region is classed as possibly secure. Data with values in the unmarked region are removed from the training data.

tally information to determine an appropriate way to restrict the parameter's domain. A constant negative threshold is chosen, based on experimentation. When a setting's historical tally drops below this threshold, the classifier deems it insecure and the parameter domain mutation operator is called to remove it from the domain.

For parameters with discrete lists of options, that insecure option is simply removed from the list, so that the mutation operator can never again select it. Each chromosome in the current generation is checked for that setting. When found, the mutation operator is applied in order to remove it. This is illustrated in Algorithm 11.

For both bit vector and numerical parameters a SVM is required. Since these types of parameters feature settings which have measurable distances between one another, it is necessary to not only remove the insecure value but an entire insecure region around it. First, each setting with a positive historical tally is placed into the *possibly secure* group. Settings with negative historical tallies less than half the threshold for are placed in the *insecure* group. The rest of the settings are discarded from the training data, although they will eventually be classified into one of the two

```
1 for each option list parameter p do
2     for each possible setting s of p do
3         if history [s] < threshold then
4             remove s from all current chromosomes via mutation and prevent
              mutation from selecting s history [c.ancestor.p] = c.ancestor.fitness -
              c.fitness
5         end
6     end
7 end
```

Algorithm 11: Parameter Domain Mutation For Option List Parameters.

groups by the SVM. This is done to avoid peppering the possibly secure region with very slightly negative settings which cannot reasonably classified as insecure. Each point, representing a single setting, is then weighted according to the absolute value of its historical tally. This method of grouping tally values can be seen in Figure 8.1.

For a bit vector of length n, the setting is projected into n-dimensional space, with each bit becoming the value of the associated number in the n-tuple. The SVM for the bit vector parameter is then trained on the resulting data with a radial basis kernel, with this particular kernel being chosen arbitrarily. This kernel function seemed effective. The process for classifying a bit vector parameter is illustrated in Algorithm 12.

For numeric parameters, the settings are placed on the x-axis of a two dimensional plane and the SVM is trained on the data with a polynomial kernel of degree two. This allows the classifier to contain all of one group inside the parabola and all of the other outside it, splitting the axis into a possibly secure region and an insecure one or intersect the axis twice within the region containing the data, creating a secure region surrounded by two insecure regions or vice versa. This is illustrated in Algorithm 13.

Whenever the bit vector or numeric parameter's value mutation operator is called, it classifies the new value it chose with the SVM. If the new value is in the *insecure* region, a new setting is regenerated until a possibly secure one is chosen. All chromosomes in the current generation are similarly checked for parameter values in the *insecure* region.

After the bit vector or numeric parameter SVM is created, it is used, unmodified, in subsequent generations until a new setting falls below the insecure threshold. When this occurs, the associated SVM is retrained. In the case of a numerical parameter, a novelty detector is created and trained on the *insecure* region. Novelty detectors are much like SVMs, except they are trained on only a single group and classify whether or not a given data point belongs or does not belong to the group. [11]

If the new *insecure* values are determined not to be a part of the previous insecure region, the degree of the polynomial kernel is incremented by two for the retraining. This modifies the classifying line so that it can intersect the axis two additional times

```
1  for each bit vector parameter p do
2      for each possible setting s of p do
3          if history [s] < threshold then
4              trainSVM = True
5          end
6      end
7      if trainSVM then
8          trainingData = ϕ
9          for each possilbe setting s of p do
10             dataPoint = (s[1], s[2], s[3]...s[s.length])
11             dataPoint.Weight = | s.Fitness |
12             if history [s] ≤ threshold/2 then
13                 dataPoint.label = insecure
14                 add dataPoint to trainingData
15             end
16             if history [s] ≤ threshold/2 then
17                 dataPoint.label = possibly-secure
18                 add dataPoint to trainingData
19             end
20         end
21         train SVM on trainingData remove all settings from currentgeneration
           which the SVM classifies as insecure via mutation
22     end
23 end
```

Algorithm 12: Parameter Domain Mutation For Bit Vector Parameters.

```
1  for each bit vector parameter p do
2      for each possible setting s of p do
3          if history [s] < threshold then
4              trainSVM = True
5          end
6      end
7      if trainSVM then
8          trainingData = φ
9          for each possilbe setting s of p do
10             dataPoint = (s, 0)
11             dataPoint.Weight = | s.Fitness |
12             if history [s] ≤ threshold/2 then
13                 dataPoint.label = insecure
14                 add dataPoint to trainingData
15             end
16             if history [s] ≤ threshold/2 then
17                 dataPoint.label = possibly-secure
18                 add dataPoint to trainingData
19             end
20         end
21         train SVM on trainingData remove all settings from currentgeneration
           which the SVM classifies as insecure via mutation
22     end
23 end
```

Algorithm 13: Parameter Domain Mutation For Numerical Parameters.

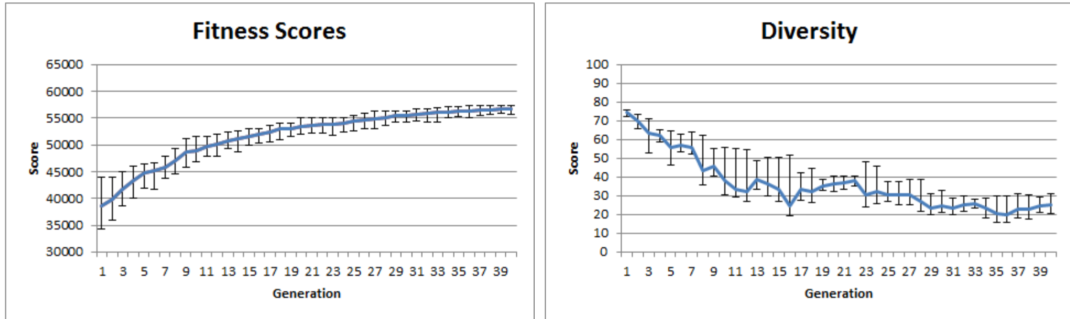
to create a new region.

The temporal classifier relies on strong correlation between changing to or from an insecure setting with fitness changes. However, in real world testing this would only be the case if the managed machines were subjected to the same exploit consistently over many operational periods. This is an unreasonable assumption, and thus a strategy must be developed to extract information within a single generation, where it can be assumed that all machines faced the same exploits.

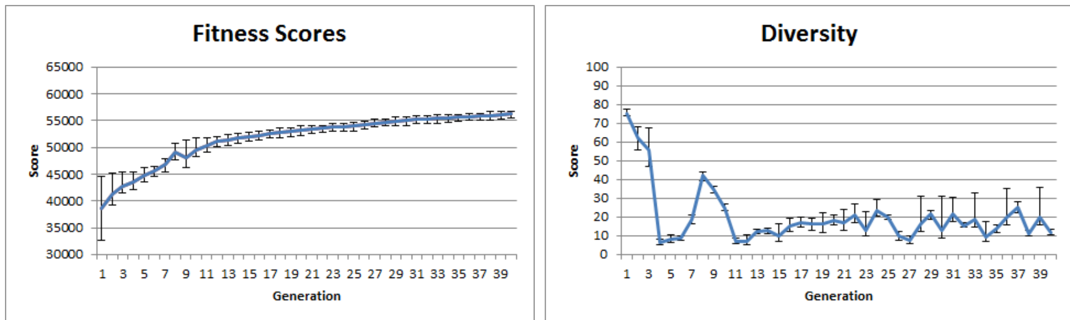
As seen in Figure 8.2 as compared to Figure 7.3(c), the classifier was able to increase the amount of fitness gain from that observed for the unsupervised ES. The final average fitness is so high that only a few parameters can be misconfigured. This final average fitness is likely as good as an ES can be expected to perform. There are some parameters with such a small range of secure settings that it is unlikely any secure settings will appear in the initial population. For example, one parameter has exactly one secure value out of $16,384$ possible settings. The ES has almost no chance to encounter the secure value, either from the initial population or through mutation. Thus, it is unlikely any ES could ever properly configure that parameter. The classifier also decreased diversity moderately, by 5.6% for the GA and 60.8% for beam search.

8.2 The Spatial Classifier

In order to classify the chromosomes, they are grouped as possibly secure or insecure. When some computers were not successfully attacked, those computers' associated configurations are classified possibly secure, while the others are classified insecure. When all computers were successfully attacked, those which faced the minimum number of attacks are deemed possibly secure, and the rest are grouped as insecure. The hope is that some number of attacks effected the entire population, and that all additional attacks failed against some subset of the machines, allowing the classifier to determine the differences that made those computers immune. If, instead, multiple attacks each effected subsets which collectively covered the entire generation, the classifier can only learn to classify based on those configurations vulnerable to both.



(a) GA with temporal classifier



(b) Beam Search with temporal classifier

Figure 8.2: Each graph measures the average fitness score or mean sampled average Hamming Distance of each generation for a single representative run of the ES MT system. The bars indicate the highest and lowest scoring members of the population, or the samples with the highest or lowest sampled average Hamming Distance. 8.2(a) shows the results of the MTGA, using 140 chromosomes over 40 generations. $p_c = 0.05$. $p_m = 1$. Genes modified per mutation = 2. Deterministic size 4 tournament selection. Two point crossover type. Parameter Domain Mutation. Hamming distance measured only for a subset of 15 randomly selected configurations. 8.2(b) shows the Beam Search with temporal classifier.

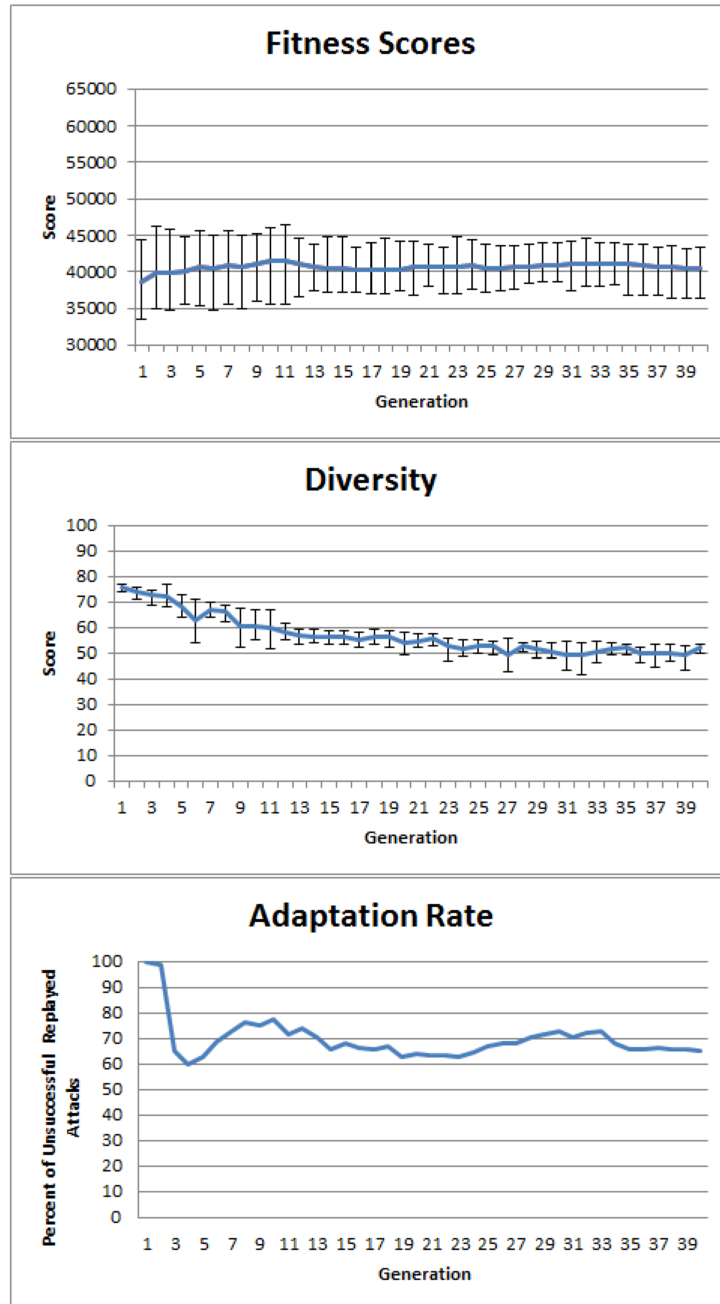


Figure 8.3: Each graph measures the average fitness score or mean sampled average Hamming Distance of each generation for a single representative run of the ES MT system. The bars indicate the highest and lowest scoring members of the population, or the samples with the highest or lowest sampled average Hamming Distance. Genetic algorithm with 140 chromosomes over 40 generations. $p_c = 0.05$. $p_m = 1$. Genes modiified per mutation = 2. Two point crossover type. Tournament selection. No directed mutation. Hamming distance measured only for a subset of 15 randomly selected configurations.

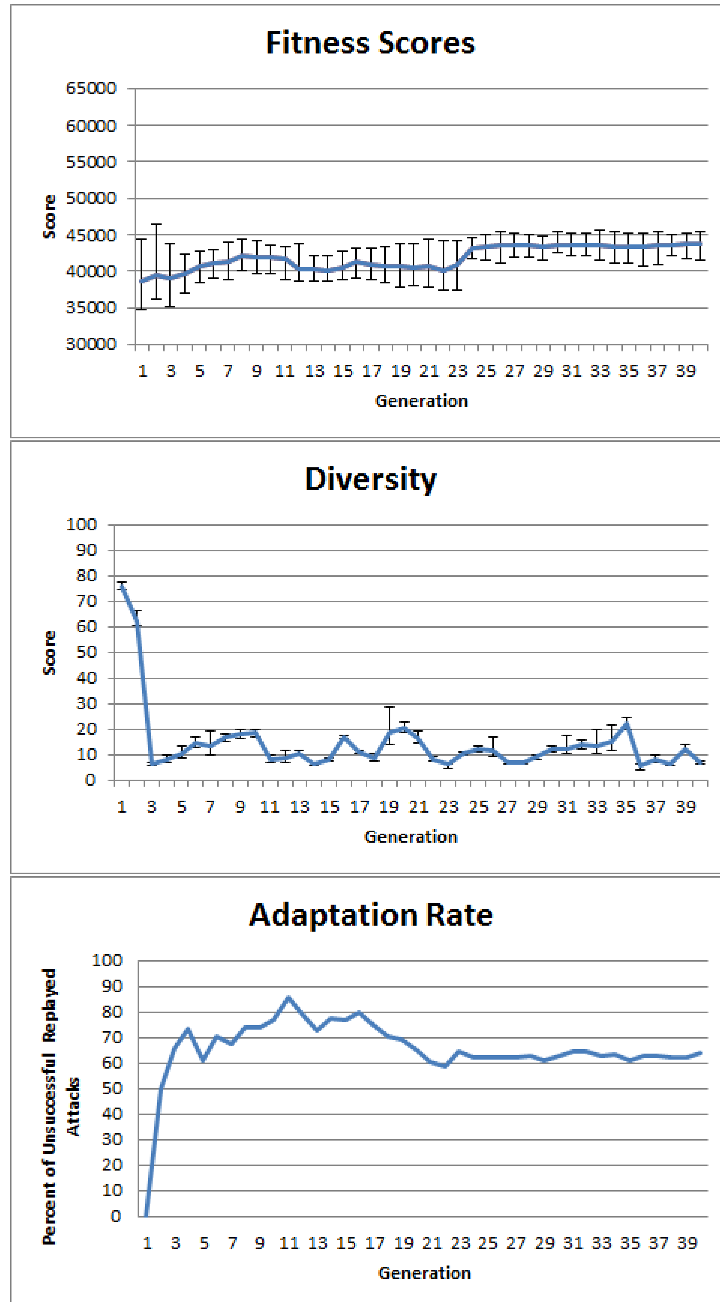


Figure 8.4: Each graph measures the average fitness score or mean sampled average Hamming Distance of each generation for a single representative run of the ES MT system. The bars indicate the highest and lowest scoring members of the population, or the samples with the highest or lowest sampled average Hamming Distance. Beam search with 140 chromosomes over 40 generations.

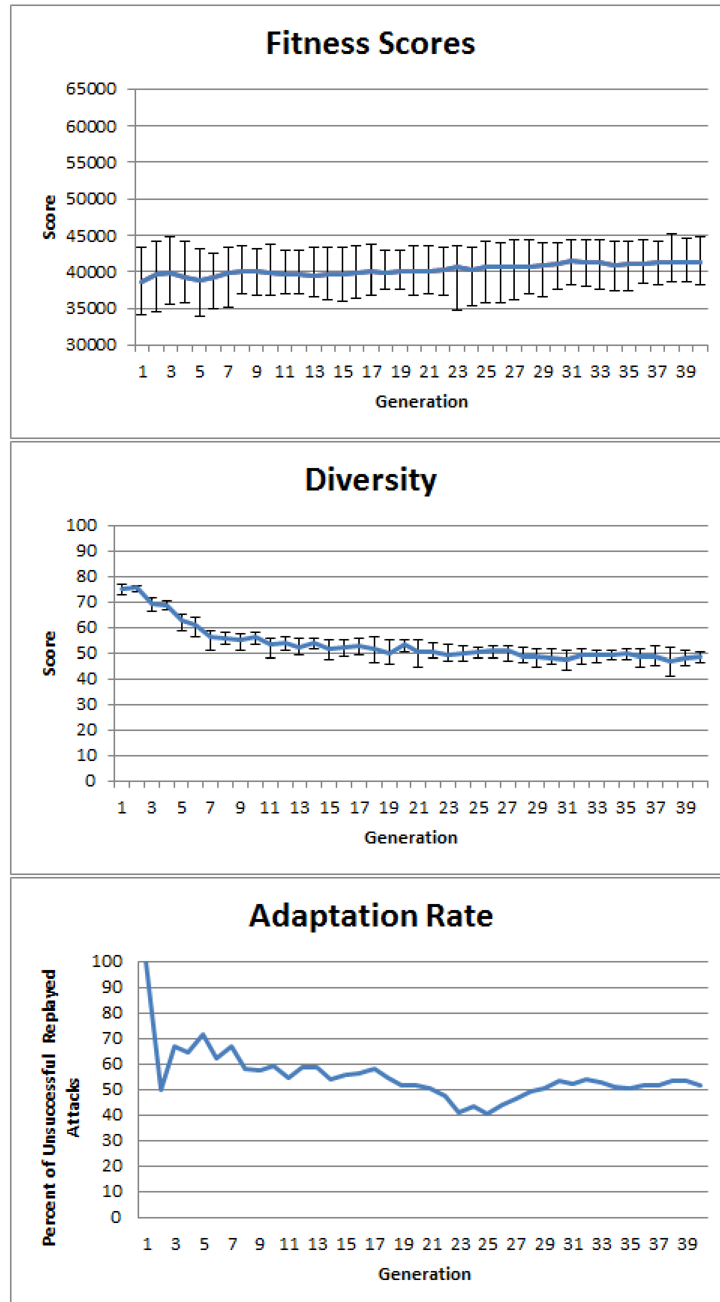


Figure 8.5: Each graph measures the average fitness score or mean sampled average Hamming Distance of each generation for a single representative run of the ES MT system. The bars indicate the highest and lowest scoring members of the population, or the samples with the highest or lowest sampled average Hamming Distance. Genetic algorithm with spatial classifier and 140 chromosomes over 40 generations. $p_c = 0.05$. $p_m = 1$. Genes modified per mutation = 2. Two point crossover type. Tournament selection. No directed mutation. Hamming distance measured only for a subset of 15 randomly selected configurations.

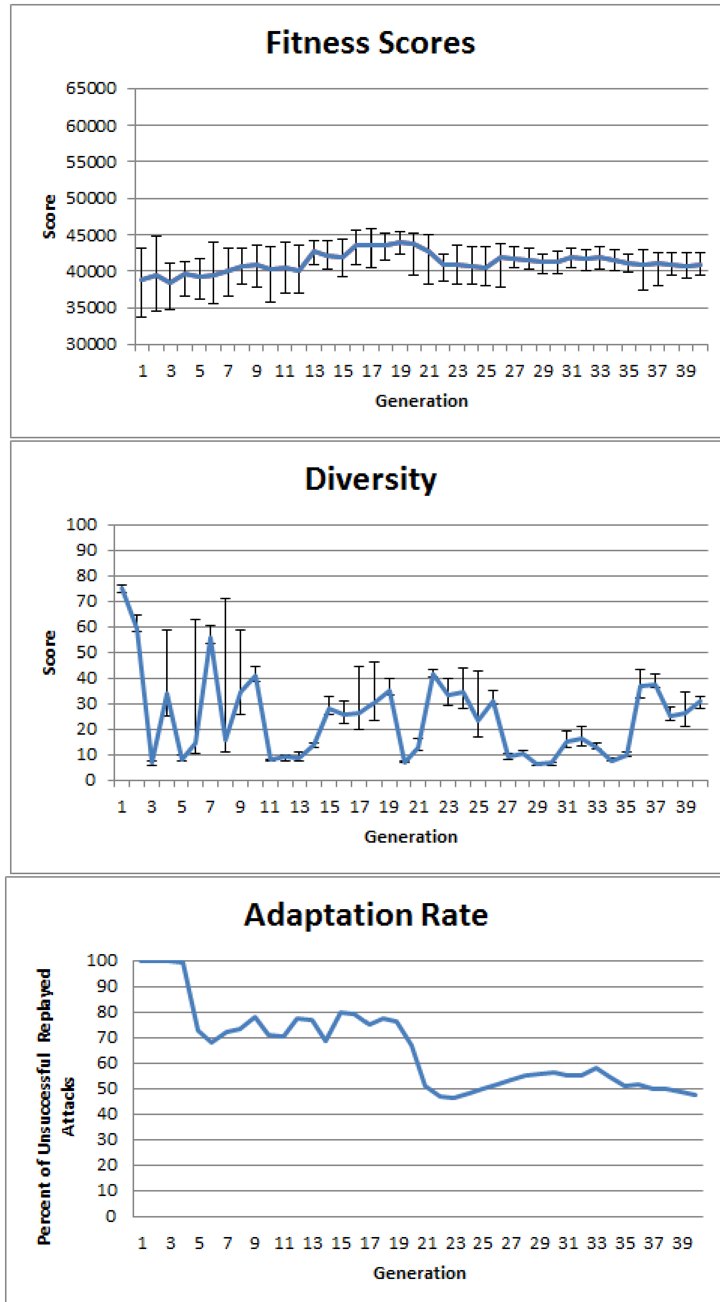


Figure 8.6: Each graph measures the average fitness score or mean sampled average Hamming Distance of each generation for a single representative run of the ES MT system. The bars indicate the highest and lowest scoring members of the population, or the samples with the highest or lowest sampled average Hamming Distance. Beam search with spatial classifier 140 chromosomes over 40 generations.

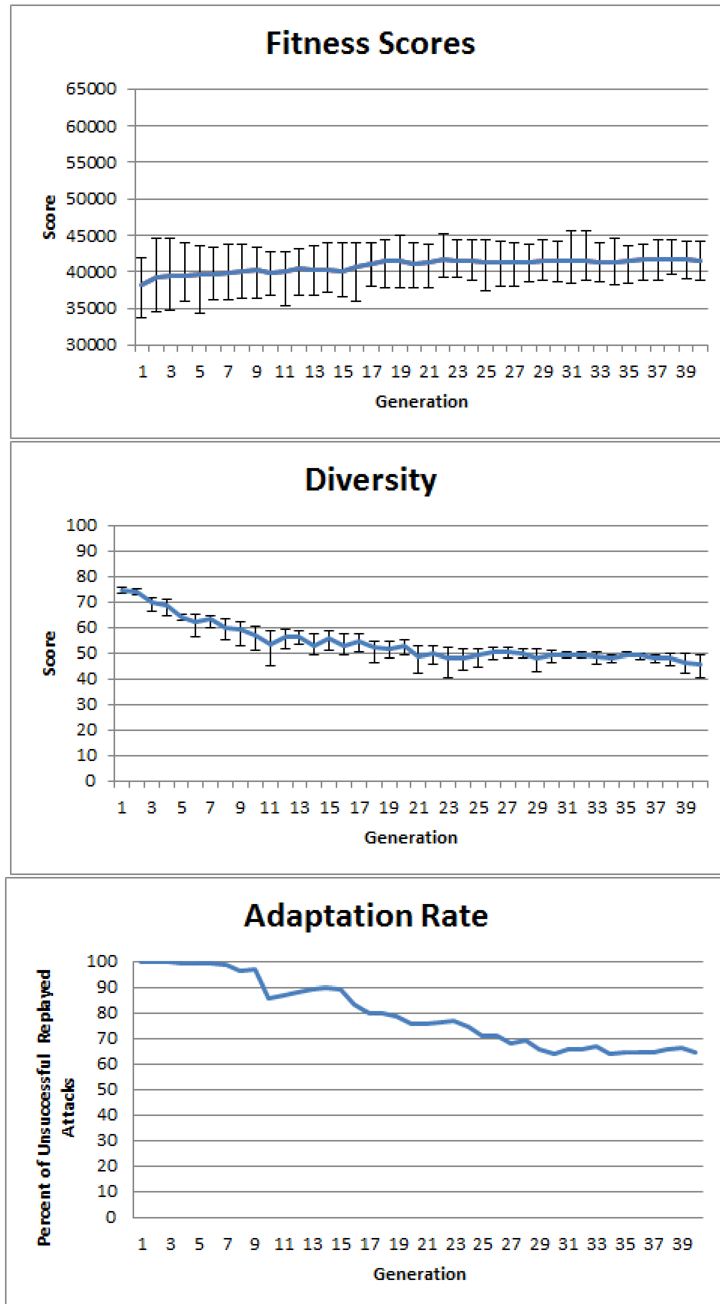


Figure 8.7: Each graph measures the average fitness score or mean sampled average Hamming Distance of each generation for a single representative run of the ES MT system. The bars indicate the highest and lowest scoring members of the population, or the samples with the highest or lowest sampled average Hamming Distance. Genetic algorithm with spatial classifier using one tree per generation and 140 chromosomes over 40 generations. $p_c = 0.05$. $p_m = 1$. Genes modified per mutation $= 2$. Two point crossover type. Tournament selection. No directed mutation. Hamming distance measured only for a subset of 15 randomly selected configurations.

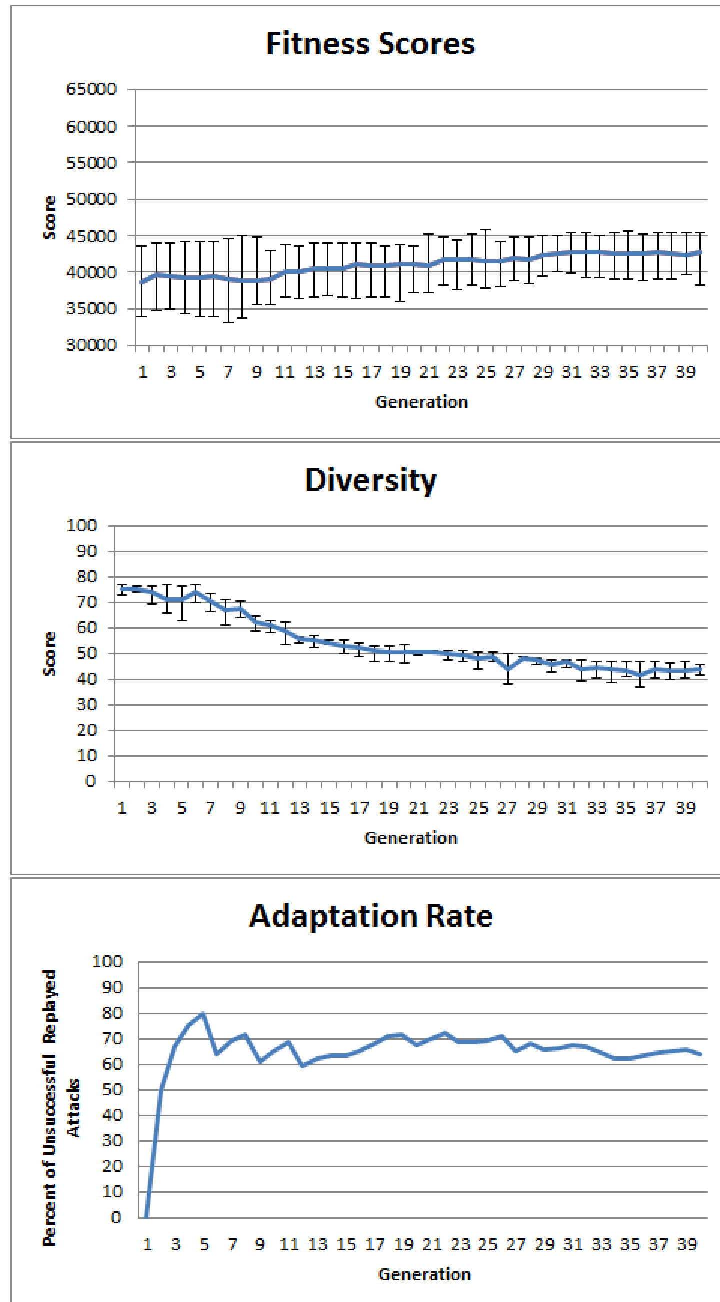


Figure 8.8: Each graph measures the average fitness score or mean sampled average Hamming Distance of each generation for a single representative run of the ES MT system. The bars indicate the highest and lowest scoring members of the population, or the samples with the highest or lowest sampled average Hamming Distance. Beam search with spatial classifier using one tree per generation 140 chromosomes over 40 generations.

```
1  trainingData = ϕ for each chromosome c from any generation do
2      dataPoint = ϕ
3      for each parameter p in c do
4          if p is an option list parameter then
5              for each possible setting s for p do
6                  if c.p == s then
7                      insert 0 into dataPoint
8                  else
9                      insert 1 into dataPoint
10                 end
11             end
12         end
13         if p is a bit vector parameter then
14             for each possible setting s for p do
15                 for each bit b in p do
16                     insert b into dataPoint
17                 end
18             end
19         end
20         if p is a numeric parameter then
21             insert p into dataPoint
22         end
23     end
24     if in any generation c was succesfully attacked more times than another
       macine in the same generation then
25         dataPoint.label = insecure
26     else
27         dataPoint.label = possibly-secure
28     end
29     add dataPoint to trainingData
30 end
31 train CART on trainingData
```

Algorithm 14: The Spatial Classifier

8.2.1 Configuration Classification

In order construct a decision tree, first a configuration is converted into a tuple in a higher dimensional space. Numerical settings remain a single number in the tuple. Bit vectors of length n are split into n different numbers, each of which gets one of the bits. Option parameters are replaced by dummy variables. A parameter with a list of i possible options will be become i numbers, exactly one of which will be equal to 1 while all others are equal to 0. A Classification and Regression Trees (CART) decision tree is then constructed based on all configurations ever scored. If, in any generation, a configuration was in the *insecure* group, it is classified *insecure*. Otherwise, it is classified as *possibly secure*. This process is illustrated in Algorithm 14.

At the end of the ES's iteration, it classifies each configuration using the decision tree. If it is deemed *insecure*, mutation is applied to a single random parameter and the chromosome is reclassified. This repeats until it is classified as *possibly secure*. This process can go on for many iterations until a possibly secure configuration is discovered, significantly increasing the time to create a new generation. However, the time taken still remains small in comparison to testing period.

It is also possible to build a separate tree for each generation. Then, configurations are checked against every tree individually, and must pass them all. This prevents insecure configurations from a generation in which they were not targeted from clustering around insecure configurations which were attacked, causing a too strict classification model.

As Figures 8.3 through 8.8, which provide the results for a single attack per generation environment using both the GA and Beam Search on their own, with the spatial classifier, and with the spatial classifier using one CART tree per generation, show, there was little difference among the ES with or without either technique. All increased fitness only slightly, by about half a percent, with adaptation rates of around 0.6. However, the spatial classifier increased the diversity of the beam search by 83.3.

Chapter 9: Discussions

Many existing strategies have been used to address the MT configuration problem. Additionally, new techniques combining Evolutionary Strategies (ES) and machine learning techniques have been developed and applied to the MT system. The resulting implementations have been tested to demonstrate their effectiveness at solving the problem.

9.1 Evolutionary Strategies for Moving Target Configuration

Two ES, a Genetic Algorithm (GA) and Beam Search, have been developed to evolve configurations for greater security while maintaining diversity. These two ES's results in increasing security and maintaining diversity can be seen in Chapter 7. For example, this fitness gain with acceptable diversity loss can be seen in Figure 7.2(b) (page 38), in which the first generation had an average fitness of $38,528$ and the final generation had an average fitness of $51,605$. This is an increase of 33%. Diversity remained at approximately 38 average pairwise Hamming distance. Thus, two randomly selected configurations will differ in about 38 parameter settings and be otherwise identical. Many other parameterizations of the GA and the implementation of Beam Search had very similar numbers in both fitness gain and diversity over the 40 generations.

These ES's fitness and diversity measurements can be compared to Figure 9.1, which shows the results of random search. In random search, every parameter for every configuration is set randomly in each generation. Random search fails to produce any significant increase in fitness. Chromosomes produced by both the ES (GA and Beam Search) by the final generation are all more fit than those produced randomly.

The ES's observed increase in fitness score correlates to more secure computer configurations. Since the maximum score a parameter can receive is 600 and the minimum is 6, each fitness score increase of 594 means that at least one security vulnerability has been partially or fully closed. For example, the representative Beam

Search run in Figure 7.11 (page 51 had average fitness $38,732.08$ in generation 1 and final average fitness $52,055.19$, instances of *Prod*, the single secure value, and roughly equal numbers of the other five, insecure values. By the last generation, there were 132 instances of *Prod*. Thus, 104 computers were secured against that exploit over 40 generations.

9.2 Supervising Evolutionary Strategies with Machine Learning

This thesis has introduced the idea of using the machine learning techniques of Support Vector Machines (SVM) and Classification and Regression Trees (CART) to explicitly classify individual genes as having low fitness and removing them from the population. These machine learning techniques have been shown to be successful in increasing fitness gain when used to supervise an ES, as by the increased rate of fitness gain in Figure 8.2 (page 59). This increase in fitness suggests that the technique can be effective in general. However, machine learning supervision can only be used for fitness functions which can be very closely approximated by concave functions, ones in which a change that produces higher fitness almost always moves the chromosome towards the global maximizer. Otherwise, the domain restriction will likely cause the population to settle on a local maximum. Since problems with such concave approximation functions are likely simple enough to use hill climbers instead of ES, it may have limited use. The strategy's secondary function of providing rules describing the population the ES has selected may be useful even if the technique ultimately hampers the ES's performance.

9.3 Machine Learning for Moving Target Configuration

The two developed ES prototypes have been modified to include classification techniques based on SVMs and CART trees. This combination of strategies resulted in the rate of fitness gain increasing as compared to an ES alone, as well as higher total average fitness in the final population, as can be seen in Chapter 8. For example, as

seen in Figure 8.2(b), the beam search with directed mutation was able to achieve a final average fitness of 56,793. Since a totally secure configuration has a fitness of 61,200, this means that only few possible exploits remain unsecured. Again, this can be demonstrated concretely by examining an individual parameter. Parameter 2 had four of its five insecure values removed from the domain, leaving only the secure value and a single insecure one. Similar results in fitness increase and the removal of vulnerable settings from the population were observed when the classification scheme was applied to the GA.

The addition of the spatial classifier to the ES did not increase the average amount of fitness gained or better defend the population against attacks it had faced. However, the ES on their own did slightly better than random. This improvement over random search can be seen in Figures 8.3 through 8.8 (pages 60 through 65 as compared to Figure 9.1. Random achieved an adaptation rate of 50%, only 10 percentage points lower than the ES. Thus, the ES would secure the network against 20% of incoming repeated attacks which random configuration would have allowed.

9.4 Research Results

Ultimately, this thesis has resulted in a system which succeeded in the goal of finding more secure configurations. Much of this work went into finding proper parameters for the GA. Currently, the best performing parameterization of the GA uses $p_c = 0.05$, $p_m = 1$, 2 genes modified per mutation, deterministic tournament selection, using elitism, and using parameter domain mutation. None of the tested crossover types significantly impacted fitness or diversity, and so all are equivalent. Other parameterizations worked nearly as well, increasing security by significant amounts. Only roulette wheel selection and nondeterministic tournament selection were noticeably worse, causing large drops in the amount of fitness gain. The beam search performed roughly as well as the GA, making it a viable alternative as the system's ES.

In the more realistic case, there was little differentiation among the techniques. All tested methods produced similar results, and there should be no real preference for one over another on the basis of performance. They performed barely better than

random.

The time to run the program increases with the use of the temporal classifier, but not by a large amount. The greater rate of fitness gain more than makes up for the additional time taken, as it remains trivially small in comparison to the assumed amount of time per operational cycle. The spatial classifier, however, heavily increased the amount of time taken. The time spent per generation increased over time, as the tree classified more and more configurations insecure, necessitating more mutation operations to reach a possibly secure configuration. This increase in time was not worthwhile, and suggests that the spatial classifier should not be used unless it can be modified to significantly increase fitness.

Several novel ideas were also developed in this thesis. Most importantly, supervising an ES with machine learning techniques was a new application of artificial intelligence techniques. Relying on classifiers to explicitly examine and remove low fitness settings is a unique idea which takes advantage of properties of the MTGA environment, such as the relatively simple fitness function and large cost associated with low fitness chromosomes appearing in the population. It has proven very successful, consistently increasing the performance of both ES when supplied full fitness information. When only given information about a limited number of attacks, it ceases to function. This is because it no longer accurately receives data for both sides of the change from one setting to another. Usually, only one of the two generations will feature an attack against that parameter, obscuring the fitness effects of the setting change.

The idea of classifying configurations instead of settings was much less successful. Classification of insecure configurations was too tight around the known vulnerable configurations previously seen. Thus, even after mutating out of the insecure region, the configuration would still be on the edge of the region and could often retain the vulnerability. If the spatial classifier is ever to be successful, it will require modifications that allow it to more accurately determine the qualities which distinguished the attacked configurations from those which remained secure.

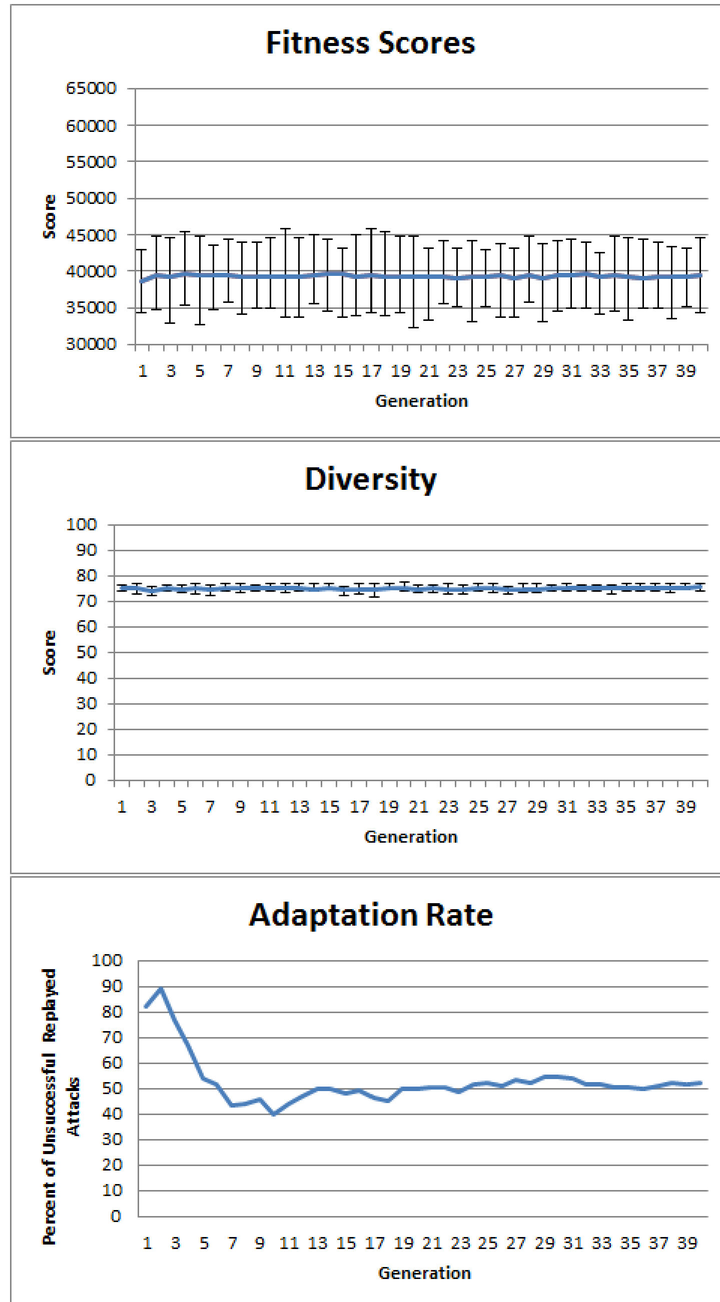


Figure 9.1: Each graph measures the average fitness score or mean sampled average Hamming Distance of each generation for a single representative run of the ES MT system. The bars indicate the highest and lowest scoring members of the population, or the samples with the highest or lowest sampled average Hamming Distance. Random Search 140 chromosomes over 40 generations.

Chapter 10: Future Work

A variety of unexplored areas of research remain which show promise in improving the system. Furthermore, many theoretical properties of the approach warrant additional research.

10.1 Automated Feasibility Testing

The problem of reconfiguring computers requires that the newly configured machine must be able to continue its intended work. This process requires an enumeration of all functionalities the machine is required to support. This list must then be converted into a number of tests which can be applied to the computer. The ES must also be modified such that it can create configurations in smaller batches than a full generation, in order to fill requests for replacements when the management component finds an infeasible configuration.

10.2 Automated Attack Detection

The system also relies on the ability of the management component to detect security incidents. The fitness function is defined by this testing feedback, and, without it, the system is incapable of assigning scores to differentiate good configurations from bad ones. A method by which the number of incidents can be measured must notice all attacks, but be computationally non-intensive enough that regular checks for attacks will not hamper the machine's operation.

10.3 Confounding Factors in System Assumptions

The system as described relies on a number of very restrictive assumptions about the nature of the machines it is configuring. All must have the exact same software load and present equally enticing targets for attackers. Furthermore, it is assumed that all attacks can be traced directly back to a vulnerability opened through misconfigura-

tion. This is not, in general, true, as many security incidences cannot be associated with a combination of settings within the configuration, either because they are allowed by misconfiguration of a parameter the system has not been given to manage or because they are related to some other uncontrollable variable, such as user behavior. The ES must be resilient enough to receive such inaccurate data without allowing it to greatly influence the algorithm's decision of what constitutes a fit chromosome.

An realistic modeling of an organization using such a system would, of necessity, require the system to manage different types of machines. These machines may, for example, be running different operating systems, different versions of some software, or even have entirely different functions, such as web servers and personal computers used in offices. This requires research into how best to handle configuration files when some machines will have different sets of parameters which require settings. Further research could then explore how information learned in one environment could be transferred to another. For example, the ES may learn that a particular setting in the Linux systems it manages is insecure, then be faced with the task of finding the corresponding parameter(s) in Microsoft Windows and determining which settings for it would open the same vulnerability. The management of different kinds of machines could also violate the assumption that an attacker is likely to attempt the same exploit against every machine. The adversary may, for example, target computers containing a database of social security numbers while ignoring other computers which house a database of grades. Again, the ES must be able to cope with this trend in the testing data without allowing it to become a decisive factor in how the system attempts to repair the vulnerability.

10.4 Diversity Measurements and Enforcement

The current strategy of sampled pairwise Hamming distance is insufficient in several ways. It can be said to be a genotypic measure, one which considers only differences between the chromosomal representations of configurations. Research may be able to produce a phenotypic measurement system, one which measures diversity based on the differences between machines as would be observed by an attacker. This latter

type of diversity is especially important to the goal of creating an MT defense, as configuration differences which do not significantly impact those attributes observed by an adversary and used in the construction of an exploit will not provide any gain in security. Even differences between chromosomal representations are not fully taken into account. For example, pairwise Hamming distance would consider the population A1, A1, B2, and B2 and A1, A2, B1, and B2 to be equally diverse, despite the fact that the second population clearly has a greater diversity in terms of setting combinations and number of repeated configurations.

Once a proper measurement of diversity is created, it may be utilized not only as a diagnostic tool, but to improve the results of the ES. More diverse populations have instances of a greater number of setting combinations, allowing the algorithm to explore more of the search space in each generation. For classification, the presence of a more diverse population during an attack allows for better information about exactly which parameter settings were relevant to the vulnerability. This in turn may reduce the number of false positives, settings which are inaccurately classified as insecure. Therefore, action may be taken during the course of the ES to artificially force the production of more diverse populations.

Some research has already been carried out with respect to causing GAs to discover multiple local optima. This area of research is called niching [10] and offers several strategies for encouraging a GA to maintain a diverse populaiton. Techniques such as Crowding, Fitness Sharing, Heterozygote Advantage, and Restricted Competetion are designed to promote spatial diversity within a single run of a GA appear best suited to the MTGA system, but further research may allow modifications to other methods to make them applicable to this domain.

10.5 Improvement of Classification Accuracy and ES Fitness Increase

There is much room for improvement in the system's performance at increasing fitness. Different kinds of ESs can be developed and tested in order to determine a better fit to the MT configuration problem, and these approaches can then be fine tuned through

experimentation for even greater performance. The classification techniques used can also be improved for greater accuracy. They may, for example, compare the attacked generation with previous ones, searching for correlations between compromised machines in the current generation and machines which had similar configurations in a previous one, so that it can determine when the reconnaissance took place and what features the attacker was looking for during it.

The system will also have to be scaled up to handle much larger data sets, such as thousands of machines with millions of parameters each, as these are the kinds of networks such a system would be expected to configure for a real organization. Increasing the scale of the system would drastically alter the performance of the GA. At least the very least, it would require further research into reconfiguring the ES. It may be that the approach would no longer be effective at such scales. In this case, it may need to be restricted to work on only a few, important applications or machines.

10.6 Specialization to Types of Defended Networks

The determination of which techniques are most effective at evolving more fit chromosomes is heavily dependent on the security environment in which the system operates. As seen in the testing for this thesis, the ES performs very well when every parameter is always attacked, while being greatly aided by the classifier. When the number of attacks is small, however, the spatial classifier is far more important to defending against attacks, while the ES serves mainly to provide a diverse population on which the classifier can draw data. More research on the number and distribution of successful cyber-attacks faced by an organization is required to better understand how to defend against them.

This also means that improvements to the system need to be tailored to the environment in which it is to be used. Take for example, the idea of weighting attacks which compromise Confidentiality, Integrity, and Assurance differently. In order to be worthwhile, this requires an environment in which the defender greatly values one of these attributes over another but also faces so many successful hacking attempts per operational cycle that weighing those attacks differently would have a noticeable

effect on fitness. It would also require that significant numbers of attackers would regularly attempt to compromise an aspect of the system which the owner deems to have such low value that it is worth sacrificing its security. Until such an environment is found, implementing this idea is of limited value.

The best approach to improving the system may be the parallel development of several different ESs, each designed to operate within a single security environment defined by the length of an operational cycle, the number and types of machines it manages, and the number and types of attacks it is likely to face.

Bibliography

[1] Leo Breiman. *Classification and regression trees.* CRC press, 1993.

[2] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

[3] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[4] Michael Crouse and Errin W Fulp. A moving target environment for computer configurations using genetic algorithms. In *Configuration Analytics and Automation (SAFECONFIG), 2011 4th Symposium on*, pages 1–7. IEEE, 2011.

[5] Michael B Crouse, Errin W Fulp, and Daniel Canas. Improving the diversity defense of genetic algorithm-based moving target approaches, 2012.

[6] John H Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[7] Dorene Kewley, Russ Fink, John Lowry, and Mike Dean. Dynamic approaches to thwart adversary intelligence gathering. In *Proc. of the DARPA Information Survivability Conference & Exposition II (DISCEX '01)*, volume 1, pages 176–185, 2001.

[8] Roger J Lewis. An introduction to classification and regression tree (cart) analysis. In *Annual Meeting of the Society for Academic Emergency Medicine in San Francisco, California*, pages 1–14, 2000.

[9] Brian F. Lucas. An automated system for evolving secure systems. Technical report, Department of Computer Science, Wake Forest University, May 2013.

[10] Samir W Mahfoud. *Niching methods for genetic algorithms.* PhD thesis, University of Illinois, 1995.

[11] Markos Markou and Sameer Singh. Novelty detection: a reviewpart 1: statistical approaches. *Signal processing*, 83(12):2481–2497, 2003.

[12] Peter Mell, Karen Scarfone, and Sasha Romanosky. CVSS v2 complete documentation. http://www.first.org/cvss/cvss-guide, March 2007.

[13] Zbigniew Michalewicz. *Genetic algorithms+ data structures= evolution programs*. Springer, 1996.

[14] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[15] NIST/SEMATECH e-handbook of statistical methods. http://www.itl.nist.gov/div898/handbook/index, February 2014.

[16] Guillaume Prigent, Florian Vichot, and Fabrice Harrouet. Ipmorph: fingerprinting spoofing unification. *Journal in Computer Virology*, 6:329–342, 2010.

[17] Raj Reddy. Foundations and grand challenges of artificial intelligence: AAAI presidential address. *AI Magazine*, 9(4):9, 1988.

[18] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[19] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

Appendix A: Evolutionary Framework Code

The code for the discovery module of the MT framework prototype follows. It is written in Python version 2.

A.1 Settings and Global Variables

```
import xml.etree.ElementTree as ElementTree
import Queue
import random
from sklearn import svm
from copy import deepcopy
from multiprocessing import Process
from sklearn import tree
from sklearn.feature_extraction import DictVectorizer

#Variable used in random runs
randTemp = False


#Logging option
logging = True

#log file
log = "log.txt"

#log for excel charts
logExcel = "logExcel.txt"

#Generate set of random configurations
newSet = True

#Percent chance of tournament selection instead of
    roullete wheel
tournamentRate = 1#0.75

#The type of crossover used
# 3point = Three point crossover
# template = two point crossover with coevolving
    template of acceptable crossover points
# template1 = one point crossover with coevolving
    template of acceptable crossover points
# uniform = arbitrary distribution of single points for
    crossover
```

```
crossoverType = "uniform"

#Number of points used in uniform crossover
numCrossoverPoints = 70

maxInt = 2147483647

mutationRate = 1#0.05
crossoverRate = 0.05#0.05#0.3

#Number of configurations per generation
populationSize = 140

#Chance of elitism
elitismRate = 0

#Number of elitist configs picked
elitismNum = 8

#ID to be assigned to next new configuration
nextConfigID = 0


#Previous generation
xmlFile = ElementTree.parse("configuration.xml")

#Pool of all configurations
poolFile = ElementTree.parse('pool.xml')

# XML for scoring chains
chainTree = ElementTree.parse('chain.xml')
chainRubric = chainTree.getroot()

#Next generation
generationTree = ElementTree.parse("generation.xml")

#Previous generation elementtree
configuration = xmlFile.getroot()

#Determines if percent chance of tournament selection
    decreases after large drops in diversity
dynamicSelectionType = False

#Historical score change tallies for each parameter
history =
    [{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},
{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},
{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},
```

```
{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},
{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},
{},{},{},{},{},{}, {}]

#Lists of banned options
banList = [[],[],[],[],[],[],[],[],[],
[],[],[],[],[],[],[],[],[],[],[],[],[],
[],[],[],[],[],[],[],[],[],[],[],[],[],
[],[],[],[],[],[],[],[],[],[],[],[],[],
[],[],[],[],[],[],[],[],[],[],[],[],[],
[],[],[],[],[],[],[],[],[],[],[],[],[],
[],[],[],[],[],[],[],[],[],[],[],[],[],
[],[],[],[],[],[],[],[],[],[],[],[],[],
[],[],[]]

#List of classifiers for each parameter
classifiers = [0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0]

#Degrees of svm kernal polynomials for each parameter
degrees = [2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2]

#Highest absolute values of all parameters
extrema = [0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0]

#When true, uses beam search instead of GA
beamSearch = True;
if beamSearch:
mutationRate = 1
crossoverRate = 0
```

```
#Number of copies of each chromosome put on beam in
   beam search
beamSize = 4

#Numbner of parameters changed per mutation
numMutations = 2

#Number of parameters to target in each generation
attacksPerGen = 1#102

#Turn on temporal classifier
classify = False;

#Turn on spatial classifier classifier
spatialClassify = True;

#Build a new tree each generation for spatial classifier
multiTree = True;

#List of configurations for spatial classifier
configData = []

#list of configurations for each generation for spatial
   classifier
multiConfigData = []
for next in range(0, 41):
multiConfigData.append([])

#List of configuration's status as possibly secure (1)
   or insecure (-1)
configGroup = []

#List of each generation's configurations' status as
   possibly secure or insecure
multiConfigGroup = []
for next in range(0, 41):
multiConfigGroup.append([])

#Minimum number of times any computer was attacked this
   generation
minAttackNumber = -1

#The spatial classifier's decision tree
classifier = tree.DecisionTreeClassifier()

#The list of decision trees for the spatial classifier
multiClassifiers = []
```

```
for next in range(0, 41):
multiClassifiers.append([])

#technical stuff for decision tree
keys = range(257)
dictV = DictVectorizer()
configDict = []

#If decision tree has been made for spatial classifier
classifierReady = False

#If a decision tree exists for a given generation for
    spatial classifier
multiClassifierReady = []
for next in range(0, 41):
multiClassifierReady.append(False)

#Number of unique attacks seen over all generations
attacksSeen = 0

#List of parameters attacked in previous generations
prevAttackProfile = []
```

A.2 Functions

```
#Find parameter with id = index in configuration element
def parameterID(element, index):
    """Find parameter, with ID index, of configuration
       element"""
    for child in element:
        if child .attrib["id"] == str(index):
            return child

#Intiailize values of new configuration file
def initialize(target):
    """Set initial values of new configuraiton file"""
    #Calculate number of parameters, intiailize
       timestamp, increment generation
    ParameterNum = len(target[0].findall('parameter'))
    target.set('ParameterNum',str(ParameterNum))
    target.set('timestamp','0')
    target.set('generation',
       str(int(target.attrib['generation']) + 1))
```

```
    #Set attacks and time used for each configuration
       in generation
    for child in target:
        child.set('attacks', '0')
        child.set('timeUp', '0')

        #Create initial crossover template
        if newSet:
            temp = ""
            for i in range(ParameterNum + 1):
                temp += '1'
            child.set('crossoverTemplate', temp)

        #Assign id
        found = False
        for child2 in pool:
            #If all parameter values identical, match
            mismatch = False
            for parameter in
               range(1,int(configuration.get('ParameterNum'))
              + 2):
                if parameter ==
                   int(configuration.get('ParameterNum'))
                  + 1:
                    if child.get('crossoverTemplate')
                       !=
                       child2.get('crossoverTemplate'):
                        mismatch = True
                        break
                elif parameterID(child, parameter).text
                   != parameterID(child2,
                   parameter).text:
                    mismatch = True
                    break
            if not mismatch:
                found = True
                child.set("id", child2.get('id'))
                break
            #Otherwise, record new id
        if not found:
            global nextConfigID
            child.set('id', str(nextConfigID))
            nextConfigID = nextConfigID + 1
            pool.append(child)


# apply logical AND or OR for chain evaluation
def applyBoolOperator(var1, var2, expr):
```

```
if expr == 'and':
return var1 and var2;
elif expr == 'or':
return var1 or var2;
else:
print 'applyBoolOperator: error, "and" and "or" not
    requested in chain rubric ';
return False;


# determine if there is a match for the chain parameter
    in the configuration
def compareChainValue(chainRule, configuration):
match = False;
for parameter in configuration:
# if parameter in configuration and matches chain
    setting
if parameter.attrib['id'] ==
    chainRule.attrib['paramid']:
# determine the type of value
# print('compareChainValue: chain', chainRule.tag,
    chainRule.attrib);
if parameter.attrib['distribution'] == 'range':
if float(parameter.text.strip()) <=
    float(chainRule.attrib['paramvalue']):
match = True;
elif parameter.attrib['distribution'] == 'permission':
match = True;
permission = parameter.text.strip();
# print('permission', permission, 'permission length',
    len(permission), 'chain perm',
    str(chainRule.attrib['paramvalue']))
for i in range(0, len(permission)):
if str(chainRule.attrib['paramvalue'])[i] != "*":
if str(chainRule.attrib['paramvalue'])[i] !=
    permission[i]:
match = False;
else:
if parameter.text.strip() ==
    chainRule.attrib['paramvalue']:
match = True;
return match;


# evaulate the Boolean expression used for chains
def evalExpr(expr, configuration):
logicalOp = expr.attrib['type'];
result = True;
```

```
for child in expr:
if child.tag == 'expr':
result = applyBoolOperator(result, evalExpr(child,
   configuration), logicalOp);
elif child.tag == 'parameter':
# determine if setting is in configuration
result = applyBoolOperator(result,
   compareChainValue(child, configuration), logicalOp);
return result;


# score the configuration based on the the chain XML
def scoreChains(configuration):
#print 'attempt to score chains '
# print(configuration.tag, configuration.attrib);
chainScore = 0
# consider all chains in the chaining rubric
        for chain in chainRubric:
                trueScore = chain.attrib['truescore'];
                falseScore = chain.attrib['falsescore'];
                # print(chain.tag, chain.attrib);
                # set the default score
                result = True;
                for expr in chain:
                        logicalOp = expr.attrib['type'];
                        result =
                           applyBoolOperator(result,
                           evalExpr(expr,
                           configuration), logicalOp);
                if result:
score = trueScore;
                else:
score = falseScore;
#print('scoreChain: chain was a ' + str(result) +',
   adding ' + score + ' to configuration score ')
chainScore = chainScore +
   convertCVSStoNumber(falseScore);
return chainScore


def convertCVSStoNumber(cvsStr):
#Get variables from the cvsStr string
# Access Vector (AV)
cvs1 = cvsStr[3]
# Access Complexity (AC)
cvs2 = cvsStr[8]
# Access autentication (Au)
cvs3 = cvsStr[13]
```

```
# Confidentiality (C)
cvs4 = cvsStr[17]
# Integrity (I)
cvs5 = cvsStr[21]
# Availability (A)
cvs6 = cvsStr[25]

# the total score for this CVSS string
score = 0

#Assign scores, bad = 1, medium = 10, good = 100
if cvs1 == 'N': score = score + 1
elif cvs1 == 'A': score = score + 10
else: score = score + 100

if cvs2 == 'L': score = score + 1
elif cvs2 == 'M': score = score + 10
else: score = score + 100

if cvs3 == 'N': score = score + 1
elif cvs3 == 'S': score = score + 10
else: score = score + 100

if cvs4 == 'C': score = score + 1
elif cvs4 == 'P': score = score + 10
else: score = score + 100

if cvs5 == 'C': score = score + 1
elif cvs5 == 'P': score = score + 10
else: score = score + 100

if cvs6 == 'C': score = score + 1
elif cvs6 == 'P': score = score + 10
else: score = score + 100

return score

#Calculate score for configuration element, return
    number of attacks experienced by element
def score(element, totalAttacks, numNotAttacked,
    numAttacked):
      """For new configurations, set all scores to 1.
          Otherwise, sum scores of all parameters"""
     #first generation
      if not randTemp:
element.set('prevScore', '1')
          element.set('score', '1')
return 0
```

```
    else:
#Update previous score
global minAttackNumber
element.set('prevScore', element.attrib['score'])
        score = 0
trueScore = 0
attacks = 0
        #Score each parameter get true score, number of
            attacks, and score on 0 or 600 binary scale
        for child in element:
#Add parameter's real value to trueScore
trueScore += convertCVSStoNumber(child
    .attrib['trueScore'])
#For nonperfect scores, record one attack
if convertCVSStoNumber(child.find('score').text) < 600:
    attacks+= 1
#If attacked, add nothing to reported score, else add
    600
else: score += 600
#uncomment to provide 6-594 cvss based scoring, comment
    to provice 0 or 600 attack detection based scoring
#score = trueScore;

# let's consider any chains and adjust the score...
#score = score + scoreChains(element)
element.attrib['trueScore'] = str(trueScore)
element.set('score', str(score))
element.attrib['attacks'] = str(attacks)
#If this configuration had fewer attacks than an other
    this far in a generation, update minAttackNumber
if attacks < minAttackNumber or minAttackNumber == -1:
    minAttackNumber = attacks
return attacks

#Select candidates for new generation
def select(pool):
    """Select a single candidate from pool"""
    totalScore = 0

    #Sum scores
    for child in pool:
        totalScore += int(child .attrib['score'])

    #Select with likelihood porpotional to score
    target = random.randrange(totalScore)
    for child in pool:
        if target < int(child .attrib['score']):
            return child
```

```
        else:
            target -= int(child .attrib['score'])

#Set up size four tournament, return winner
def tournamentSelection():
#select four configuraitons with roulette wheel
entry1 = select(configuration)
entry2 = select(configuration)
entry3 = select(configuration)
entry4 = select(configuration)

#decide first two rounds
finalist1 = tournamentRound(entry1, entry2)
finalist2 = tournamentRound(entry1, entry2)

#return winner
return tournamentRound(finalist1, finalist2)

#Decides tournament round
def tournamentRound(entry1, entry2):
#Determanisticly return higher valued configuration
if int(entry1.attrib['score']) >
   int(entry2.attrib['score']): return entry1
#Select with probability porpotional to score
#if random.random() < float(entry1.attrib['score']) /
   float(entry2.attrib['score']): return entry1
else: return entry2

#Determine which kind of selection to use
def selectType():
temp = random.random()
if temp < tournamentRate: return tournamentSelection()
else: return select(configuration)

#Select top scoring configurations from previous
   generation
def elitistSelection():
data = []
for config in configuration:
key = config.attrib['score']
data.append((key, config))
data.sort(reverse=True)
for config in range(elitismNum):
generation.append(data[config][1])


#Determine if mutation should occur
def mutationChance():
```

```
    """Return true with probability = mutationRate"""
    if newSet: return True
    else: return random.random() < mutationRate

#Uniform distribution mutation
def uniformMutator(start, finish, parameterID):
    """Return random intiget in specified range"""
    #Use uniiform instead of modified normal
    #temp = random.randrange(start, finish + 1)
    #Choose random number according to normal
        distribution
    temp = random.gauss((finish + start) / 2, 10);
    #%50 chance to shift number by half size of the
        range
    if (bool(random.getrandbits(1))): temp = (temp +
        finish / 2) % finish;
    #Regenerate values outside region's bounds
    while temp < start or temp > finish:
temp = random.gauss((finish + start) / 2, 10);
if (bool(random.getrandbits(1))): temp = (temp + finish
    / 2) % finish;
    #If a classifier has been made, regenerate until
        secure value in bounds is found
    if classifiers[parameterID] != 0:
while classifiers[parameterID].predict([temp /
    extrema[parameterID], 0]) == [0] or temp < start or
    temp > finish:
#if classifiers[parameterID].predict([temp /
    extrema[parameterID], 0]) == [0]: print ("hit")
#temp = random.randrange(start, finish + 1)
temp = random.gauss((finish + start) / 2, 10);
if (bool(random.getrandbits(1))): temp = (temp + finish
    / 2) % finish;
    return str(int(temp))

#Gamma distribution mutation
def gammaMutator(preferred, start, finish, parameterID):
    """Return random number according ot gamma
        distribution"""
    #No easy way to programmatically create gamma
        distribution, program one for each range in by
        hand
    temp = -1
    if preferred == 15:
        while True:
temp = random.gammavariate(7.4, 1.8)
if (temp >= start) and (temp <= finish):
#If classifier has been made, exclude vulnerable values
```

```
if newSet or classifiers[parameterID] == 0: break
if classifiers[parameterID].predict([temp /
   extrema[parameterID], 0]) == [1]: break
    return str(round(temp));

#Bitflip mutation
def bitflipMutator(targetString, parameterID):
    """Change one bit in binary string"""
    target = list(targetString)
    length = len(target)

    #Random binary string for new set
    if newSet:
        for i in range(length):
            target[i] = str(random.randrange(2))
        return "".join(target)
    #Select random bit, and flip it
    while(True):
temp = random.randrange(length)
     if target[temp] == '0':
         target[temp] = '1'
         candidate = "".join(target)
else:
         target[temp] = '0'
         candidate = "".join(target)
if classifiers[parameterID] == 0: return candidate
#If classifier has been made, exclude vulnerable values
if gen > 1 and classifiers[parameterID].predict(target)
   == [1]: return candidate

#Uniform option mutator
def uniformOptionMutator(target):
    """Return string of one option at random"""
    #build list of options form xml, return one at
       random
    options = target.findall('option')
    numOptions = len(options)
    temp = random.randrange(numOptions)
    return options[temp].text

#Determine if crossover should occur
def crossoverChance():
    """Return true with probability = crossoverRate"""
    if newSet: return False
    else: return random.random() < crossoverRate

#Determine if elitism should occur
def elitismChance():
```

```
    """Return true with probaility = elitismRate"""
    if newSet: return False
    else: return random.random() < elitismRate

#Return a list of points for corssover between
   configurations child and partner
def crossover(child, partner, generation):
    """Return list of points for crossover to occur"""
    #Return points which differ from third
       configuration selectd from generation
    if crossoverType == "3point":
partner2 = select(generation)
points = []
for parameter in
   range(int(generation.get('ParameterNum'))):
if parameterID(child,parameter + 1).text !=
   parameterID(partner2,parameter + 1).text:
   points.append(parameter + 1)
return points
    #Select a valid point from each template, return
       range between them
    if crossoverType == "template":
        template1 = child .attrib['crossoverTemplate']
        template2 = partner.attrib['crossoverTemplate']
        point1 = random.randrange(len(template1) - 1)
#Uncomment to turn crossover templates on
#i = 0
        #while template1[point1] != '1':
# print i
# i = i + 1
        # point1 = random.randrange(len(template1))
        point2 = random.randrange(len(template2) - 1)
#i = 100000
        #while template2[point1] != '1':
# print i
# i = i + 1
        # point2 = random.randrange(len(template2))
        if point1 < point2:
            return range(point1 + 1, point2 + 2)
        else:
            return range(point2 + 1, point1 + 2)
    #Select a valid point from template, return range
       between it and end of chromosome
    if crossoverType == "template1":
        template1 = child .attrib['crossoverTemplate']
        template2 = partner.attrib['crossoverTemplate']
        point1 = random.randrange(len(template1) - 1)
#Uncomment to turn crossover tmeplates on
```

```
        #while template1[point1] != '1':
        # point1 = random.randrange(len(template1))
point2 = len(template2) - 1
        if point1 < point2:
            return range(point1 + 1, point2 + 2)
        else:
            return range(point2 + 1, point1 + 2)
    #Select random points parameter list
    elif crossoverType == "uniform":
        points = []
        ParameterNum =
           int(generation.get('ParameterNum'))
        for i in range(numCrossoverPoints):
            point = random.randrange(ParameterNum) + 1
            while points.count(point) != 0:
                point = random.randrange(ParameterNum)
                   + 1
            points.append(point)
        return points

#Decide which permutation of parameters will be used as
   the ordering for this pair of configurations for
   this crossover
def crossoverPermutation():
    """Returns list with permutation of parameters"""
    #Permutations start with zero for proper indexing,
       though the zero is never selected
    temp = range(1,102)
    random.shuffle(temp)
    return [0] + temp

#Calculate hamming distance to members of sample of
   other configurations in generation
def divergence(configurations):
    divergences = []

    #samples = random.sample(range(0, 34), 15)
    samples = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
       12, 13, 14]
    temp = 0
    for child in configurations:
#If configuration in sample
if temp in samples:
distance = 0.0
temp2 = 0
#For each other configuration in sampe
         for child2 in configurations:
if temp2 in samples:
```

```
#For each parameter, increase distance if there is a
    difference
for parameter in range(1,102):
        if parameter ==
            int(configuration.get('ParameterNum')) + 1:
        if child.get('crossoverTemplate') !=
            child2.get('crossoverTemplate'):
        distance = distance + 1
        elif parameterID(child, parameter).text !=
            parameterID(child2, parameter).text:
        distance = distance + 1
temp2 = temp2 + 1
        divergences.append(distance / 14)
temp = temp + 1
    return divergences

#Create a classifier for numerical parameter with id
    =index
def classify(index, tally) :
data = []
group = []
weight = []
minimum = 0
#update extrema
for point in tally:
if tally[point] < minimum: minimum = tally[point]
if abs(float(point)) > extrema[index]: extrema[index] =
    abs(float(point))
#Populate classifier training data set, <min/2 score
    inseucre, >0 socre possibly secure
for point in tally:
data.append([float(point) / extrema[index], 0])
if tally[point] < minimum / 2: group.append(0)
else: group.append(1)
if tally[point] < 0 and tally[point] >= minimum/2:
    weight.append(0)
else: weight.append(abs(tally[point]))
classifiers[index] =
    svm.SVC(kernel='poly',degree=degrees[index])
classifiers[index].fit(data, group,
    sample_weight=weight)

#Create a classifier for bit vector parameter with
    id=index
def classifyBitflip(index, tally):
data = []
group = []
weight = []
```

```
minimum = 0
#update extrema
for point in tally:
if tally[point] < minimum: minimum = tally[point]
if abs(float(point)) > extrema[index]: extrema[index] =
   abs(float(point))
#populate classifier with training data set
for point in tally:
data.append(list(point))
if tally[point] < minimum / 2: group.append(0)
else: group.append(1)
if tally[point] < 0 and tally[point] >= minimum/2:
   weight.append(0)
else: weight.append(abs(tally[point]))
classifiers[index] =
   svm.SVC(kernel='linear',degree=degrees[index])
classifiers[index].fit(data, group,
   sample_weight=weight)

#mutate the current generation
def mutate():
#apply mutation numMutation times
for dummy in range(numMutations):
if (logging and not newSet): open(log,
   'a').write("\nMutation\n")
#For each configuration in the current generation
for config in generation:
#FOr initial generations, mutate every chromosome
if newSet:
for child in config:
if mutationChance():
if child .attrib['distribution'] == 'bitflip':
child.text = bitflipMutator(child.text, 0)
elif child .attrib['distribution'] == 'gamma':
child.text = gammaMutator(int(child
   .attrib['preferred']), int(child
   .attrib['feasibleRangeStart']), int(child
   .attrib['feasibleRangeEnd']), 0)
         elif child .attrib['distribution'] ==
            'uniform':
         if child .attrib['feasibleRangeEnd'] ==
            'MaxInt':
         child.text = child.text =
            uniformMutator(int(child
            .attrib['feasibleRangeStart']), maxInt, 0)
         else:
         child.text = uniformMutator(int(child
            .attrib['feasibleRangeStart']),
```

```
            int(int(child
               .attrib['feasibleRangeEnd'])), 0)
         elif child .attrib['distribution'] ==
            'uniformOption':
         child.text = uniformOptionMutator(child)
else:
         print("Bad distribution option:" + child
            .attrib['distribution'])
#Check if mutaiton occurs
elif mutationChance():

#choose random parameter
temp =
   random.randrange(int(configuration.attrib['ParameterNum'])
   + 1);
#Log configuration id, mutated parameter
if (logging and not newSet): open(log,
   'a').write(config.get("id") + " " + str(temp) + " ")
if temp == 0:
         config.set('crossoverTemplate',
            bitflipMutator(config.attrib['crossoverTemplate'],
            0))
if (logging and not newSet): open(log,
   'a').write(config.get('crossoverTemplate') + "\n")
         else:
changed = False
#until a new valid value has been chosen, keep
   reapplying mutation
while(changed == False):
temp =
   random.randrange(int(configuration.attrib['ParameterNum'])
   + 1);
while (temp == 0): temp =
   random.randrange(int(configuration.attrib['ParameterNum'])
   + 1);
child = parameterID(config, temp)
#save parameter child's current value
prev = child.text
#apply appriopriate mutation function
         if child .attrib['distribution'] == 'bitflip':
         child.text = bitflipMutator(child.text, temp)
         if (logging and not newSet): open(log,
            'a').write(child.text + "\n")
elif child .attrib['distribution'] == 'gamma':
         child.text = gammaMutator(int(child
            .attrib['preferred']), int(child
            .attrib['feasibleRangeStart']), int(child
            .attrib['feasibleRangeEnd']), temp)
```

```
        if (logging and not newSet): open(log,
            'a').write(child.text + "\n")
        elif child .attrib['distribution'] ==
            'uniform':
        if child .attrib['feasibleRangeEnd'] ==
            'MaxInt':
        child.text = child.text =
            uniformMutator(int(child
            .attrib['feasibleRangeStart']), maxInt,
            temp)
        if (logging and not newSet): open(log,
            'a').write(child.text + "\n")
        else:
        child.text = uniformMutator(int(child
            .attrib['feasibleRangeStart']),
            int(int(child
            .attrib['feasibleRangeEnd'])), temp)
        if (logging and not newSet): open(log,
            'a').write(child.text + "\n")
        elif child .attrib['distribution'] ==
            'uniformOption':
        child.text = uniformOptionMutator(child)
        if (logging and not newSet): open(log,
            'a').write(child.text + "\n")
        else:
        print("Bad distribution option:" + child
            .attrib['distribution'])
#Check if parameter value actually changed
if child.text != prev:
child .attrib["prev"] = prev
#For multitree classfication, check chromosome against
    each existing rule
if multiTree:
changed = True
for rule in range(0,41):
if multiClassifierReady[rule]:
if multiClassifiers[rule].predict(dictV.
transform(dict(zip(keys,
    configToVector(config)))).toarray()) != [1]:
changed = False
else:
if classifierReady:
if classifier.predict(dictV.transform(dict(zip(keys,
    configToVector(config)))).toarray()) == [1]: changed
    = True
#Log new value
if (logging and not newSet): open(log,
    'a').write(child.text + "\n")
```

```
#Convert xml configuration to dictionary
def configToVector(target):
configVector = []
temp = 0
for parameter in target:
if parameter.attrib['distribution'] == 'bitflip':
for bit in parameter.text:
configVector.append(bit)

elif parameter.attrib['distribution'] ==
    'uniformOption':
configVector.append(parameter.text)

else:
configVector.append(float(parameter.text))


return configVector
```

```
#!/usr/bin/python

import socket,time
from XMLConfig import XMLConfig
#from Logger import Logger #worry about this in a second

DEBUG = 0

READY = 0
ACK = 0
XML = 0
MSG_SIZE = 16
#line below creates empty XMLConfig object
#initConfig = XMLConfig('')
initConfig = XMLConfig('generation.xml')
pickledConfig = ""
count = 1

def communicate(generation):
"""INSERT GA HOOKS/CODE HERE"""
"""FOR THIS PHASE OF TESTING, YOU CAN CHANGE THE
    CONNECTION TO THE AS"""
"""s.connect(('localhost',9998))"""
```

```
global READY
global ACK
global XML

#initialize the XMLConfig object here
initConfig = XMLConfig("generation.xml")

s = socket.socket()
time.sleep(0.01)
#comment line below to bypass VM connection, replace
   port 9999 with 9998
s.connect(('localhost',9999))


pickledConfig = initConfig.picklers()

#check to see if the VM Farm Manager is ready
while READY == 0:
#generate 16 byte first message
str_to_send = str(len('RDY?')) + ':'
pad_len = MSG_SIZE - len(str_to_send)
for i in range (0, pad_len):
str_to_send += '%'

s.sendall(str_to_send)
#block and wait to recv reply
s.sendall('RDY?')
buffer = s.recv(MSG_SIZE)
#print buffer
length_str,ignored,buffer = buffer.partition(':')
buffer = s.recv(int(length_str))
while True:
                        if len(buffer) >=
                           int(length_str):
                                break
                        length = int(length_str)
                        len_buff = int(len(buffer))
                        buffer +=
                           s.recv(length-len_buff)


if buffer == 'RDY':
READY = 1
elif buffer == 'NRDY':
print 'VM␣Farm␣is␣not␣yet␣Ready'
else:
print 'Malformed␣Communication␣at␣RDY,␣Exiting'
exit(0)
```

```
while ACK == 0:
    str_to_send = str(len(pickledConfig)) + ':'
pad_len = MSG_SIZE - len(str_to_send)
for i in range (0, pad_len):
str_to_send += '%'
s.sendall(str_to_send)
s.sendall(pickledConfig)

buffer = s.recv(MSG_SIZE)
#print buffer
length_str, ignored, buffer = buffer.partition(':')
buffer = s.recv(int(length_str))
while True:
                              if len(buffer) >=
                                 int(length_str):
                                       break
                              length = int(length_str)
                              len_buff = int(len(buffer))
                              buffer +=
                                 s.recv(length-len_buff)

#print buffer
if buffer == 'ACK':
ACK = 1
elif buffer == 'NACK':
print 'VM Farm Manager is not yet ready'
else:
print "Malformed communication protocol, Exiting"
exit(0)

ACK = 0
READY = 0

serv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
          serv.setsockopt(socket.SOL_SOCKET,
             socket.SO_REUSEADDR,1)
          serv.bind(('localhost',9997))
          serv.listen(5)

          (c,address) = serv.accept()

          while READY == 0:
                    buffer = c.recv(MSG_SIZE)
                    length_str, ignored,buffer =
                       buffer.partition(':')
                    buffer = c.recv(int(length_str))
while True:
```

```
                    if len(buffer) >=
                       int(length_str):
                            break
                    length = int(length_str)
                    len_buff = int(len(buffer))
                    buffer +=
                       c.recv(length-len_buff)
            if buffer == 'RDY?':
                   READY = 1
                    str_to_send = str(len('RDY')) +
                       ':'
                    pad_len = MSG_SIZE -
                       len(str_to_send)
                    for i in range (0, pad_len):
                            str_to_send += '%'
                    c.sendall(str_to_send)
                    c.sendall('RDY')
            else:
                    print "Malformed Communication
                       Protocol, Exiting"
                    exit(0)


while ACK == 0:
#Determine if GA is ready goes here
#Receive message from AS, this is the generation that
   has
#just been scored
buffer = c.recv(MSG_SIZE)
            length_str, ignored, buffer =
               buffer.partition(':')
            buffer = c.recv(int(length_str))
while True:
                    if len(buffer) >=
                       int(length_str):
                            break
                    length = int(length_str)
                    len_buff = int(len(buffer))
                    buffer +=
                       c.recv(length-len_buff)

            #if OK, then ACK
            str_to_send = str(len('ACK')) + ':'
            pad_len = MSG_SIZE - len(str_to_send)
            for i in range(0, pad_len):
                    str_to_send += '%'
            c.sendall(str_to_send)
            c.sendall('ACK')
```

```
                    ACK = 1

READY = 0
ACK = 0

while READY == 0:
                    str_to_send = str(len('RDY?')) + ':'
pad_len = MSG_SIZE - len(str_to_send)
for i in range (0, pad_len):
str_to_send += '%'

#str_to_send = str(len('RDY?')) + ':' + 'RDY?'
c.sendall(str_to_send)
#block and wait to recv reply
c.sendall('RDY?')
buffer = c.recv(MSG_SIZE)
length_str,ignored,buffer = buffer.partition(':')
buffer = c.recv(int(length_str))
while True:
                              if len(buffer) >=
                                 int(length_str):
                                     break
                              length = int(length_str)
                              len_buff = int(len(buffer))
                              buffer +=
                                 c.recv(length-len_buff)

if buffer == 'RDY':
READY = 1
elif buffer == 'NRDY':
print 'AS is not yet Ready'
else:
print 'Malformed Communication at RDY, Exiting'
exit(0)

while XML == 0:
#need to arbitrarily decide where GENID is defined
"""THE GENERATION ID SHOULD BE DEFINED AT LATEST HERE"""
GENID = generation.get('generation')
print(GENID)
str_to_send = str(len(GENID)) + ':'
pad_len = MSG_SIZE - len(str_to_send)
for i in range (0, pad_len):
str_to_send += '%'
c.sendall(str_to_send)
c.sendall(GENID)
```

```
buffer = c.recv(MSG_SIZE) #expecting to see pickled
    config
length_str, ignored, buffer = buffer.partition(':')
buffer = c.recv(int(length_str))
while True:
                                if len(buffer) >=
                                   int(length_str):
                                        break
                                length = int(length_str)
                                len_buff = int(len(buffer))
                                buffer +=
                                   c.recv(length-len_buff)
"""CONFIGURATION DUMPED HERE"""
initConfig.unpickles(buffer)
str_to_send = str(len('ACK')) + ':'
pad_len = MSG_SIZE - len(str_to_send)
for i in range (0, pad_len):
str_to_send += '%'
c.sendall(str_to_send)
c.sendall('ACK')
XML = 1


#Modified from original code, set received config as
    xmlFile
xmlFile._setroot(initConfig.root)
          xmlFile.write("configuration.xml")
initConfig.dumpConfigToFile()
print "COMPLETE_SUCESS"
READY = 0
ACK = 0
XML = 0
```

A.3 Evolutionary Strategy Implementation

```
#Generation number
gen = 0

#Loop over generations
while True:

minAttackNumber = -1

#Previous generation
configuration = xmlFile.getroot()
```

```
#History of all configurations ever seen
pool = poolFile.getroot()

#Compute total attacks
totalAttacks = 0
numNotAttacked = 0
numAttacked = 0
for child in configuration:
totalAttacks = totalAttacks + int(child
    .attrib["attacks"])
if int(child .attrib["attacks"]) == 0: numNotAttacked =
    numNotAttacked + 1
else: numAttacked = numAttacked + 1

print('start')

#Score incoming configurations
for child in configuration:
totalAttacks += score(child, totalAttacks,
    numNotAttacked, numAttacked)

#Save xml representation of current generation. If
    generation over 40, exit
tempGenNum = configuration.get('generation')
if int(configuration.get('generation')) > 40:
configuration.set('generation', '0')
xmlFile.write('generation' + tempGenNum + '.xml')
exit(0)
xmlFile.write('generation' + tempGenNum + '.xml')


#Whether or not to attempt to ban values in this
    generation
ban = False

print('classify')
#update historical score chnge tallies
for child in configuration:
#If a configuration has a non-defualt previous socre,
    can begin computing
if child .attrib['prevScore'] != '1':
ban = True
#For each parameter add signed difference between it
    and previous configuration's socre to value's
    history and previous value's history
for parameter in child:
if parameter.attrib['prev'] != "":
indexID = int(parameter.attrib['id'])
```

```
if not parameter.attrib['prev'] in history[indexID]:
    history[indexID][parameter.attrib['prev']] = 0
if not parameter.text in history[indexID]:
    history[indexID][parameter.text] = 0
change = int(child .attrib['score']) - int(child
    .attrib['prevScore'])
history[indexID][parameter.attrib['prev']] =
    history[indexID][parameter.attrib['prev']] - change
history[indexID][parameter.text] =
    history[indexID][parameter.text] + change

#Determine and remove insecure values
#if not classify: ban = False
ban = False
#Index of parameter being considered
index = -1
if ban:
for tally in history:
index = index + 1
for setting in tally:
#If a setting is below the threshold, begin banning
    procedure
if tally[setting] < -4800:
#For gamma/uniform distributions
if (parameterID(configuration[0],
    index).attrib['distribution'] == 'gamma' or
    parameterID(configuration[0],
    index).attrib['distribution'] == 'uniform') and
    tally[setting] < -4800
#Check if value already banned earlier in this
    generation
if classifiers[index] == 0:
classify(index, tally)
banList[index].append(setting)
elif not setting in banList[index]:
#if this value is not caught by classifier built from
    earlier configuraitons in this generation, retrain
if classifiers[index].predict([float(setting) /
    extrema[index],0]) == [0]:
    banList[index].append(setting)
else:
classify(index, tally)
banList[index].append(setting)
#For bitflip distributions, check if value already
    banned, check if caught by latest classifier, if not
    then retrain
elif parameterID(configuration[0],
    index).attrib['distribution'] == 'bitflip':
```

```
if classifiers[index] == 0:
classifyBitflip(index, tally)
banList.append(setting)
elif not setting in banList[index]:
if classifiers[index].predict(list(setting)) == [0]:
    banList[index].append(setting)
else:
classifyBitflip(index, tally)
banList[index].append(setting)


#Remove banned values from current generation
for child in configuration:
#For option parameters
if parameterID(child, index).attrib['distribution'] ==
    'uniformOption' and tally[setting] < -4800:
#Remove option node for all configurations
for option in parameterID(child,
    index).findall('option'):
options = parameterID(child, index).findall('option')
numOptions = len(options)
if option.text == setting and not numOptions == 1:
parameterID(child, index).remove(option)
#If setting present, mutate it
if parameterID(child, index).text == setting:
    parameterID(child, index).text =
    uniformOptionMutator(parameterID(child,index))
#For other parameters, check value with classifier and
    mutat if found insecure
if parameterID(child, index).attrib['distribution'] ==
    'bitflip':
if classifiers[index].predict(list(parameterID(child,
    index).text)) == [0]: parameterID(child, index).text
    = bitflipMutator(parameterID(child, index).text,
    index)
if (parameterID(child, index).attrib['distribution'] ==
    'gamma' or parameterID(child,
    index).attrib['distribution'] == 'uniform') and
    tally[setting] < -2400:
if classifiers[index].predict([float(parameterID(child,
    index).text) / extrema[index],0]) == [0]:
if parameterID(child, index).attrib['distribution'] ==
    'gamma': parameterID(child, index).text =
    gammaMutator(int(parameterID(child,
    index).attrib['preferred']), int(parameterID(child,
    index).attrib['feasibleRangeStart']),
    int(parameterID(child,
    index).attrib['feasibleRangeEnd']), index)
```

```
elif parameterID(child, index).attrib['distribution']
    == 'uniform':
if parameterID(child, index).attrib['feasibleRangeEnd']
    == 'MaxInt':
parameterID(child, index).text =
    uniformMutator(int(parameterID(child,
    index).attrib['feasibleRangeStart']), maxInt, index)
else:
parameterID(child, index).text =
    uniformMutator(int(parameterID(child,
    index).attrib['feasibleRangeStart']),
    int(int(parameterID(child,
    index).attrib['feasibleRangeEnd'])), index)

#Classify configurations on non-initial generations
if spatialClassify and gen > 0:
for child in configuration:
configVector = configToVector(child)
#Configuration group insecure if the configuration
    experienced the non-minimum number of attacks,
    otherwise it's possibly secure
if int(child .attrib['attacks']) > minAttackNumber:
    securityGroup = -1
else: securityGroup = 1
#Build training data from unique configuraitons
if multiTree:
if not configVector in multiConfigData[gen]:
multiConfigData[gen].append(configVector)
multiConfigGroup[gen].append(securityGroup)
else:
if not configVector in configData:
configData.append(configVector)
configGroup.append(securityGroup)
#If congfiguration was insecurei n previous generation,
    reset it to insecure group
elif configGroup[configData.index(configVector)] == 1
    and securityGroup == -1:
    configGroup[configData.index(configVector)] = -1

configDict = []
if multiTree:
for config in multiConfigData[gen]:

configDict.append(dict(zip(keys, config)))

else:
for config in configData:
```

```
configDict.append(dict(zip(keys, config)))

#if there are any insecure configurations,
   update/create tree
if multiTree:
if -1 in multiConfigGroup[gen]:
multiClassifiers[gen] =
   classifier.fit(dictV.fit_transform(configDict).toarray(),
   multiConfigGroup[gen])
multiClassifierReady[gen] = True

else:
if -1 in configGroup:
classifier =
   classifier.fit(dictV.fit_transform(configDict).toarray(),
   configGroup)
classifierReady = True







randTemp = True

#Next generation
generation = generationTree.getroot()


if newSet: diversity = 0

print('log')
#Log generation scores/diversity
if (logging):

#Average and variance of scores
scores = []
total = 0.0
for child in configuration:
scores.append(int(child.get('trueScore')))
for number in scores:
open(log, 'a').write(str(number) + " ")
open(logExcel, 'a').write(str(number) + ",")
total = total + number
total = total / len(scores)
#calclulate score increase
```

```
if generation.get('generation') == '1': scoreChange =
    total
if generation.get('generation') == '40':
scoreChange = total - scoreChange
print scoreChange
variance = 0.0
for number in scores:
variance = variance + (number - total)**2
variance = variance / len(scores)
open(log, 'a').write("\navg " + str(total) + " var " +
    str(variance))
open(logExcel, 'a').write(str(total) + "," +
    str(variance) +',')

#Average and variance of diversity measures
divergences = divergence(configuration)
total = 0.0
for number in divergences:
open(logExcel, 'a').write(str(number) + ",")
total = total + number
total = total / len(divergences)
variance = 0.0
for number in divergences:
variance = variance + (number - total)**2
variance = variance / len(divergences)
open(log, 'a').write("\nDiversity\navg " + str(total) +
    " var " + str(variance))
open(logExcel, 'a').write(str(total) + "," +
    str(variance) +',')
if logging: open(log, 'a').write("\n\n\n\nGeneration "
    + str(int(generation.get("generation")) + 1) + '\n')

#If diversity fell enough in the last generation,
    decrease torunament rate
prevDiversity = diversity
diversity = total
if dynamicSelectionType and prevDiversity - diversity >
    140 / 40: tournamentRate = tournamentRate - 0.05




print('select')
#Index of configurations for next generation
temp = 0
#Next generation
```

```
generation = generationTree.getroot()
#Create empty generation
for child in generation.findall('configuration'):
generation.remove(child)
#For beamsearch, put in beam width copiues of top
    configurations
if(beamSearch and not newSet):
q = Queue.PriorityQueue();
for child in configuration:
q.put((int(child .attrib['score']) * -1 , child))
while temp < populationSize / beamSize:
repeatingConfig = q.get(True)
temp = temp + 1
for i in range(beamSize):
generation.append(deepcopy(repeatingConfig[1]))


#For GA, apply selection
else:
while temp < populationSize:
generation.append(deepcopy(selectType()))
temp = temp + 1

#Has elitism occurred
elitismChosen = False

#If elitism is to occur, remove elitism number
    configuraitons form generation and decrease
    generation size accordingly
if elitismChance():
elitismChosen = True
for child in range(elitismNum):
generation.remove(generation[child])
populationSize = populationSize - elitismNum


#Reset all parameter prev values
for child in generation:
for parameter in child:
parameter.attrib['prev'] = "";


print('crossover')

#Crossover
if (logging and not newSet): open(log,
    'a').write("Crossover\n")
for child in generation:
```

```
#Check that crossover template is proper length
if len(child .attrib['crossoverTemplate']) !=
   int(configuration.attrib['ParameterNum']) + 1:
if not(newSet): print('Malformed crossover template.')
#If crossover occurs
if crossoverChance()
#Select other parent:
partner = select(configuration)
#Log parent ids
if (logging and not newSet): open(log,
   'a').write(child.get('id') + " " + partner.get('id'))
#Get points to swap
crossoverPoints = crossover(child, partner, generation)
#Ancestor is parent with more points contributed in
   crossover
if len(crossoverPoints) > 51: ancestor = "partner"
else: ancestor = "child"
#Permute parameters
permutation = crossoverPermutation()
#Swap all parameter values between selected points,
   inclusive
for parameter in range(1,102):
crossoverPoint = permutation[parameter]
if parameter in crossoverPoints:
#Crossover template is notionally just past the end of
   the list of parameters
         if parameter ==
            int(configuration.attrib['ParameterNum']) +
            1:
if (logging and not newSet): open(log, 'a').write(" " +
   str(parameter))
temp = child .attrib['crossoverTemplate']
child.set('crossoverTemplate',
   partner.attrib['crossoverTemplate'])
partner.set('crossoverTemplate', temp)
else:
#Log crossed over points
if (logging and not newSet): open(log, 'a').write(" " +
   str(permutation[parameter]))
#Set parameter's prev values
   if ancestor == "child":
if parameterID(child,crossoverPoint).text !=
   parameterID(partner,crossoverPoint).text:
   parameterID(child, crossoverPoint).attrib['prev'] =
   parameterID(child, crossoverPoint).text
else: parameterID(child, crossoverPoint).attrib["prev"]
   = ""
```

```
parameterID(child,crossoverPoint).text =
   parameterID(partner,crossoverPoint).text

else:
if parameterID(child,crossoverPoint).text !=
   parameterID(partner,crossoverPoint).text:
   parameterID(child, crossoverPoint).attrib['prev'] =
   parameterID(partner, crossoverPoint).text
else: parameterID(child, crossoverPoint).attrib["prev"]
   = ""


if (logging and not newSet): open(log, 'a').write('\n')


print('mutation')
mutate();

#If elitism happened, add elitist chromosomes and
   restore population size
if elitismChosen:
populationSize = populationSize + elitismNum;
elitistSelection();

initialize(generation)


poolFile.write('pool.xml')



#Uncomment to send current generation to management
   component and wait for assessment server feedback
#communicate(generation)
        newSet = True


#Score configurations in this process to remove other
   modules
#Scoring rubric
rubricFile = ElementTree.parse('rubric.xml')
rubric = rubricFile.getroot()
last = 0



print('score')
#Determine which parameters will be attacked
```

```
attackProfile = random.sample(range(1, 103),
   attacksPerGen)
replayedAttacks = 0
for config in generation:

for parameter in config:
#Calculate parameter 8's score eplicityly, because of
   large number of combinations needed to input into
   rubric explicityly
if parameter.attrib['id'] == '8':
scoreVectorString = "AV:L/AC:H/Au:M/C:N/I:N/A:N"
flaw = False
scoreVector = list(scoreVectorString)
if parameter.text[5] == '1':
flaw = True
scoreVector[17] = 'P'
scoreVector[21] = 'P'
scoreVector[25] = 'P'
if parameter.text[0] == '1':
flaw = True
if scoreVector[21] == 'P':
scoreVector[21] = 'C'
else: scoreVector[21] = 'P'
if parameter.text[1] == '1':
flaw = True
if scoreVector[17] == 'P':
scoreVector[17] = 'C'
else: scoreVector[17] = 'P'
if scoreVector[25] == 'P':
scoreVector[25] = 'C'
else: scoreVector[25] = 'P'
if parameter.text[2] == '1':
flaw = True
if scoreVector[25] == 'P':
scoreVector[25] = 'C'
else: scoreVector[25] = 'P'
if parameter.text[3] == '1':
flaw = True
if scoreVector[21] == 'P':
scoreVector[21] = 'C'
else: scoreVector[21] = 'P'
if parameter.text[4] == '1':
flaw = True
if scoreVector[17] == 'P':
scoreVector[17] = 'C'
else: scoreVector[17] = 'P'
if parameter.text[6] == '1':
flaw = True
```

```
if scoreVector[25] == 'P':
scoreVector[25] = 'C'
else: scoreVector[25] = 'P'
if parameter.text[7] == '1':
flaw = True
if scoreVector[17] == 'P':
scoreVector[17] = 'C'
else: scoreVector[17] = 'P'
if parameter.text[8] == '1':
flaw = True
if scoreVector[25] == 'P':
scoreVector[25] = 'C'
else: scoreVector[25] = 'P'
if parameter.text[9] == '1':
flaw = True
if scoreVector[25] == 'P':
scoreVector[25] = 'C'
else: scoreVector[25] = 'P'
if parameter.text[10] == '1':
flaw = True
if scoreVector[17] == 'P':
scoreVector[17] = 'C'
else: scoreVector[17] = 'P'
if parameter.text[11] == '1':
flaw = True
if scoreVector[21] == 'P':
scoreVector[21] = 'C'
else: scoreVector[21] = 'P'
elif flaw:
scoreVector[3] = 'N'
scoreVector[8] = 'L'
scoreVector[13] = 'N'
parameter.find('score').text = "".join(scoreVector)


else:
#Find rule's fior this parameter
for rule in rubric:
if parameter.attrib['id'] == rule.attrib['paramid']:
#Apply score of lowest valued rule the parameter is
   less than or equal to, or the highest valued rule if
   it is higher than all of them
if rule.attrib['type'] == 'range':
if rule.attrib['id'] == '1':
parameter.find('score').text = rule.attrib['score']
last = int(rule.attrib['paramvalue'])
elif float(parameter.text) <=
   float(rule.attrib['paramvalue']) and
```

```
    float(parameter.text) > last:
parameter.find('score').text = rule.attrib['score']
last = int(rule.attrib['paramvalue'])
elif float(parameter.text) >
    float(rule.attrib['paramvalue']):
parameter.find('score').text = rule.attrib['score']
last = int(rule.attrib['paramvalue'])
else:
last = int(rule.attrib['paramvalue'])
#Match value to first applicable schema in a rule
elif rule.attrib['type'] == 'permission':
match = True;
for i in range(0, len(parameter.text)):
if str(rule.attrib['paramvalue'])[i] != "*":
if str(rule.attrib['paramvalue'])[i] !=
    parameter.text[i]:
match = False;
if match: parameter.find('score').text =
    rule.attrib['score']
#Apply score of for this option
elif parameter.text == rule.attrib['paramvalue']:
parameter.find('score').text = rule.attrib['score']
parameter.attrib['trueScore'] =
    parameter.find('score').text
#Check if previously seen attack would apply to this
    configuraiton
if gen > 0 and int(parameter.attrib['id']) in
    prevAttackProfile and parameter.find('score').text
    != "AV:L/AC:H/Au:M/C:N/I:N/A:N": replayedAttacks += 1
#If this parameter is not to be attacked, assign false
    maximum score
if not int(parameter.attrib['id']) in attackProfile:
parameter.find('score').text =
    "AV:L/AC:H/Au:M/C:N/I:N/A:N"

print(replayedAttacks)
print(attacksSeen)
#Log adaptation rate
if attacksSeen > 0: adaptPercent = str(100 -
    (float(replayedAttacks)/float(140 * attacksSeen) *
    100))
else: adaptPercent = "100"
open(log, 'a').write("Adaptation Ratio\n" +
    adaptPercent + '\n')
open(logExcel, 'a').write( adaptPercent + ',\n')

#Add current attacks to profile of all previosu attakcs
for attack in attackProfile:
```

```
if not attack in prevAttackProfile:
prevAttackProfile.append(attack)
attacksSeen += 1




print gen
gen += 1

#Set new generation to previous generation
xmlFile._setroot(deepcopy(generationTree.getroot()))
newSet= False
#generationTree.write('configuration.xml')
```

Appendix B: Parameter Decriptions

The 102 parameters used in testing are descriped here. They are identfied by the DISA STIG number of their associated security vulnerability. The first line lists the type of the parameter. "bit flip" is a bit vector, "gamma" is a number with a gamma distribution, "uniform" is a number with a modified normal distribution, and "uniformOption" is a list of options. For numbers, the range of possible values is also provided here. For parameters with a gamma distribution, pref gives a number describing a point before which most of the probability distribution lies.

The rest of the lines describe the parameter's possible values and their associated scores. Bit vectors are described by rules giving each bit as either a 0, 1, or a * for "don't care". Going from the top of the list to the bottom, the first rule the setting matches defines the score for that parameter. Number variables have a list of numbers. Going down the list, if the setting is less than or equal to given number, it is assigned that score. If it is reaches the last line, the last line's score is applied regardless of the setting's value.

```
<!WG270 A22>
<parameter distribution="bitflip"
score ='AV:N/AC:L/AU:N/C:P/I:P/A:P'
paramvalue='******110' />
score ='AV:N/AC:L/AU:N/C:P/I:P/A:P'
paramvalue='******111' />
score ='AV:N/AC:L/Au:N/C:P/I:N/A:N'
paramvalue='******100' />
score ='AV:N/AC:L/Au:N/C:P/I:N/A:N'
paramvalue='******101' />
score ='AV:N/AC:L/Au:N/C:P/I:P/A:N'
paramvalue='******010' />
score ='AV:N/AC:L/Au:N/C:P/I:P/A:N'
paramvalue='******011' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N'
paramvalue='******000' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N'
paramvalue='******001' />

<!WG520 A22>
<parameter distribution="uniformOption"
score ='AV:N/AC:L/Au:N/C:P/I:N/A:N' paramvalue='Major' />
score ='AV:N/AC:L/Au:N/C:P/I:N/A:N' paramvalue='Minor' />
```

```
score ='AV:N/AC:L/Au:N/C:P/I:N/A:N' paramvalue='Min' />
score ='AV:N/AC:L/Au:N/C:P/I:N/A:N' paramvalue='OS' />
score ='AV:N/AC:L/Au:N/C:P/I:N/A:N' paramvalue='FULL' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='Prod' />

<!WA000-WWA024 A22>
<parameter distribution="gamma" feasibleRangeEnd="300"
feasibleRangeStart="0" preferred="15"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='5' />
score ='AV:N/AC:L/Au:N/C:N/I:N/A:P' paramvalue='15' />
score ='AV:N/AC:L/Au:N/C:N/I:N/A:C' paramvalue='300' />

<!WA000-WWA022 A22 >
<parameter distribution="uniformOption"
score = 'AV:N/AC:L/Au:N/C:N/I:N/A:P' paramvalue='Off' />
score = 'AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='On' />

<!WA000-WWA052 A22 >
<parameter distribution="uniformOption"
score='AV:L/AC:H/Au:M/C:N/I:N/A:N'
paramvalue='+FollowSymLinks' />
score='AV:N/AC:L/Au:M/C:P/I:N/A:N' paramvalue='-
FollowSymLinks' />

<!WA00515 A22>
<parameter distribution="uniform" feasibleRangeEnd="1"
feasibleRangeStart="0"
score='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='0' />
score='AV:N/AC:L/Au:N/C:P/I:N/A:N' paramvalue='1' />

<!WA00547 A22>
<parameter distribution="uniformOption"
score='AV:N/AC:L/Au:N/C:P/I:P/A:P' paramvalue='All' />
score='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='None' />
score='AV:N/AC:L/Au:N/C:P/I:P/A:P'
paramvalue='AuthConfig' />
score='AV:N/AC:L/Au:N/C:P/I:P/A:P' paramvalue='FileInfo'
/>
score='AV:N/AC:L/Au:N/C:P/I:P/A:P' paramvalue='Index' />
score='AV:N/AC:L/Au:N/C:P/I:P/A:P' paramvalue='Limit' />

<!WA00565 A22>
<parameter distribution="bitflip"
score ='AV:N/AC:L/AU:N/C:P/I:P/A:P'
paramvalue='******110' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N'
paramvalue='******111' />
score ='AV:N/AC:L/Au:N/C:P/I:P/A:P'
paramvalue='******100' />
score ='AV:N/AC:L/Au:N/C:P/I:P/A:P'
paramvalue='******101' />
```

```
score ='AV:N/AC:L/Au:N/C:P/I:P/A:P'
paramvalue='******010' />
score ='AV:N/AC:L/Au:N/C:P/I:P/A:P'
paramvalue='******011' />
score ='AV:N/AC:L/Au:N/C:P/I:P/A:P'
paramvalue='******000' />
score ='AV:N/AC:L/Au:N/C:P/I:P/A:P'
paramvalue='******001' />

<!WA000-WWA058 A22>
<parameter distribution="uniformOption"
score='AV:N/AC:L/Au:N/C:P/I:N/A:N' paramvalue='+Indexes'
/>
score='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='Indexes' />

<!WA000-WWA060 A22 >
<parameter distribution="uniform"
feasibleRangeEnd="21474836" feasibleRangeStart="0"
score='AV:N/AC:L/Au:N/C:P/I:N/A:C' paramvalue='0' />
score='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='1' />

<!WA000-WWA062 A22 >
<parameter distribution="uniform"
feasibleRangeEnd="32767" feasibleRangeStart="0"
score='AV:N/AC:L/Au:N/C:P/I:N/A:C' paramvalue='0' />
score='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='1' />

<!WA000-WWA064 A22 >
<parameter distribution="uniform"
feasibleRangeEnd="MaxInt" feasibleRangeStart="0"
score='AV:N/AC:L/Au:N/C:P/I:N/A:C' paramvalue='8191' />
score='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='8192' />

<!WA000-WWA066 A22>
<parameter distribution="uniform"
feasibleRangeEnd="MaxInt" feasibleRangeStart="0"
score='AV:N/AC:L/Au:N/C:P/I:N/A:C' paramvalue='8191' />
score='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='8192' />
<parameter distribution="uniformOption"
score='AV:L/AC:M/Au:N/C:P/I:P/A:P' paramvalue='None' />
score='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='Includes'
/>

<!WA000-WWA054 A22>
score='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='-
IncludesNoExec' />
score='AV:L/AC:H/Au:M/C:N/I:N/A:N'
paramvalue='+IncludesNoExec' />

<!WG370 A22 >
<parameter distribution="uniformOption" id="15"
```

```
score='AV:N/AC:M/Au:N/C:P/I:P/A:P' paramvalue='Enabled' />
score='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='Disabled'
/>

<!WG080 A22 >
<parameter distribution="uniformOption" id="16"
score='AV:L/AC:M/Au:S/C:N/I:P/A:P' paramvalue='Yes' />
score='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='No' />

<!WG200 A22 >
<parameter distribution="uniformOption" id="17"
score='AV:L/AC:L/Au:S/C:C/I:N/A:N' paramvalue='Local' />
score='AV:N/AC:L/Au:N/C:C/I:N/A:N' paramvalue='Anyone' />
score='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='Admin' />

<!WG130 A22 >
<parameter distribution="uniformOption" id="18"
score='AV:L/AC:L/Au:S/C:C/I:C/A:C' paramvalue='Available'
/>
score='AV:L/AC:H/Au:M/C:N/I:N/A:N'
paramvalue='Unavailable' />

<!WG300 A22 >
<parameter distribution="bitflip"
score ='AV:N/AC:L/AU:N/C:P/I:P/A:P'
paramvalue='******11*' />
score ='AV:N/AC:L/AU:N/C:P/I:N/A:N'
paramvalue='******10*' />
score ='AV:N/AC:L/AU:N/C:P/I:P/A:N'
paramvalue='******01*' />
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='******00*' />

<!WG330 A22 >
<parameter distribution="uniformOption"
score='AV:N/AC:L/Au:N/C:P/I:N/A:C' paramvalue='In' />
score='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='Out' />
score='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='None' />

<!WA000WWA020 A22 >
<parameter distribution="uniform"
feasibleRangeEnd="10000" feasibleRangeStart="0"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='300' />
score ='AV:N/AC:L/Au:N/C:N/I:N/A:P' paramvalue='1000' />
score ='AV:N/AC:L/Au:N/C:N/I:N/A:C' paramvalue='1001' />

<!NEEDS RANGE WA000WWA026 A22 >
<parameter distribution="uniform" feasibleRangeEnd="30"
feasibleRangeStart="1"
score ='AV:N/AC:L/Au:N/C:N/I:N/A:P' paramvalue='5' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='6' />
```

```
<!NEEDS RANGE WA000WWA028 A22 >
<parameter distribution="uniform" feasibleRangeEnd="40"
feasibleRangeStart="0"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='5' />
score ='AV:N/AC:L/Au:N/C:N/I:N/A:P' paramvalue='20' />
score ='AV:N/AC:L/Au:N/C:N/I:N/A:C' paramvalue='21' />

<!NEEDS RANGE WA000WWA030 A22 >
<parameter distribution="uniform" feasibleRangeEnd="60"
feasibleRangeStart="1"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='5' />
score ='AV:N/AC:L/Au:N/C:N/I:N/A:P' paramvalue='30' />
score ='AV:N/AC:L/Au:N/C:N/I:N/A:C' paramvalue='31' />

<! WA000WWA032 A22 >
<parameter distribution="uniform" feasibleRangeEnd="2000"
feasibleRangeStart="1"
score ='AV:N/AC:L/Au:N/C:N/I:N/A:P' paramvalue='250' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='400' />
score ='AV:N/AC:L/Au:N/C:N/I:N/A:P' paramvalue='1000' />
score ='AV:N/AC:L/Au:N/C:N/I:N/A:C' paramvalue='1001' />

<!WA000WWA052 A22 >
<parameter distribution="bitflip"
score ='AV:N/AC:M/AU:N/C:P/I:N/A:N'
paramvalue='******10*' />
score ='AV:N/AC:H/AU:N/C:C/I:C/A:N'
paramvalue='*******1*' />
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='******00*' />

<!WA000WWA056 A22>
<parameter distribution="uniformOption"
score='AV:L/AC:L/Au:N/C:C/I:N/A:N'
paramvalue='+Multiviews' />
score='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='-
Multiviews' />

<!WA000WWA00505 A22 >
<parameter distribution="bitflip"
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N' paramvalue='000' />
score ='AV:N/AC:l/AU:S/C:C/I:C/A:N' paramvalue='1**' />
score ='AV:N/AC:l/AU:S/C:C/I:C/A:N' paramvalue='*1*' />
score ='AV:N/AC:l/AU:S/C:C/I:C/A:N' paramvalue='**1' />

<!WA000WWA00510 A22 >
<parameter distribution="bitflip"
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N' paramvalue='00' />
score ='AV:N/AC:M/AU:S/C:P/I:P/A:P' paramvalue='1*' />
score ='AV:N/AC:M/AU:S/C:P/I:P/A:P' paramvalue='*1' />
```

```
<!WA00525 A22 >
<parameter distribution="uniformOption"
score='AV:N/AC:M/Au:S/C:P/I:P/A:P' paramvalue='Yes' />
score='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='No' />

<!WA00530 A22 >
<parameter distribution="bitflip"
score ='AV:N/AC:M/AU:N/C:P/I:N/A:N'
paramvalue='******10*' />
score ='AV:N/AC:M/AU:N/C:N/I:P/A:P'
paramvalue='*******1*' />
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='******00*' />

<! WA00535 A22 >
<parameter distribution="bitflip"
score ='AV:N/AC:M/AU:N/C:P/I:N/A:N'
paramvalue='******10*' />
score ='AV:N/AC:M/AU:N/C:N/I:P/A:P'
paramvalue='*******1*' />
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='******00*' />

<!WA00540 A22 >
<parameter distribution="bitflip"
score ='AV:N/AC:L/AU:M/C:C/I:N/A:N' paramvalue='1*' />
score ='AV:N/AC:L/AU:M/C:C/I:N/A:N' paramvalue='01' />
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N' paramvalue='00' />

<! WA00545 A22 >
<parameter distribution="bitflip"
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='*00000000' />
score ='AV:N/AC:M/AU:N/C:N/I:P/A:P'
paramvalue='*1*******' />
score ='AV:N/AC:M/AU:N/C:N/I:P/A:P'
paramvalue='**1******' />
score ='AV:N/AC:M/AU:N/C:N/I:P/A:P'
paramvalue='***1*****' />
score ='AV:N/AC:M/AU:N/C:N/I:P/A:P'
paramvalue='****1****' />
score ='AV:N/AC:M/AU:N/C:N/I:P/A:P'
paramvalue='*****1***' />
score ='AV:N/AC:M/AU:N/C:N/I:P/A:P'
paramvalue='******1**' />
score ='AV:N/AC:M/AU:N/C:N/I:P/A:P'
paramvalue='*******1*' />
score ='AV:N/AC:M/AU:N/C:N/I:P/A:P'
paramvalue='********1' />
```

```
<! GEN000000LNX00400 >
<parameter distribution="bitflip"
score ='AV:L/AC:L/AU:N/C:P/I:P/A:P'
paramvalue='******11*' />
score ='AV:L/AC:L/AU:N/C:P/I:N/A:N'
paramvalue='******10*' />
score ='AV:L/AC:L/AU:N/C:N/I:P/A:P'
paramvalue='******01*' />
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='******00*' />

<! GEN001366>
<parameter distribution="bitflip"
score ='AV:L/AC:L/AU:N/C:P/I:P/A:P'
paramvalue='******11*' />
score ='AV:L/AC:L/AU:N/C:P/I:N/A:N'
paramvalue='******10*' />
score ='AV:L/AC:L/AU:N/C:N/I:P/A:P'
paramvalue='******01*' />
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='******00*' />

<! GEN001371 >
<parameter distribution="bitflip"
score ='AV:L/AC:L/AU:N/C:P/I:P/A:P'
paramvalue='******11*' />
score ='AV:L/AC:L/AU:N/C:P/I:N/A:N'
paramvalue='******10*' />
score ='AV:L/AC:L/AU:N/C:N/I:P/A:P'
paramvalue='******01*' />
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='******00*' />

<! GEN001378 >
<parameter distribution="bitflip"
score ='AV:L/AC:L/AU:N/C:P/I:P/A:P'
paramvalue='******11*' />
score ='AV:L/AC:L/AU:N/C:P/I:N/A:N'
paramvalue='******10*' />
score ='AV:L/AC:L/AU:N/C:N/I:P/A:P'
paramvalue='******01*' />
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='******00*' />

<! GEN001400 >
<parameter distribution="bitflip"
score ='AV:L/AC:L/AU:N/C:P/I:P/A:P'
paramvalue='******11*' />
score ='AV:L/AC:L/AU:N/C:C/I:N/A:N'
paramvalue='******10*' />
score ='AV:L/AC:L/AU:N/C:C/I:C/A:N'
```

```
paramvalue='******01*' />
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='******00*' />

<! GEN001660 >
<parameter distribution="bitflip"
score ='AV:L/AC:L/AU:N/C:P/I:P/A:P'
paramvalue='******11*' />
score ='AV:L/AC:L/AU:N/C:P/I:N/A:N'
paramvalue='******10*' />
score ='AV:L/AC:L/AU:N/C:P/I:P/A:N'
paramvalue='******01*' />
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='******00*' />

<! GEN000000LNX001431 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='root' />
score ='AV:L/AC:L/Au:S/C:C/I:C/A:C' paramvalue='user' />

<! GEN000000LNX001432 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='root' />
score ='AV:L/AC:L/Au:S/C:C/I:C/A:C' paramvalue='group' />

<! GEN000000LNX001433 >
<parameter distribution="bitflip"
score ='AV:L/AC:H/AU:M/C:C/I:C/A:C'
paramvalue='11*****0*' />
score ='AV:L/AC:L/AU:S/C:C/I:C/A:C'
paramvalue='*******1*' />
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='00*****0*' />

<! GEN000000LNX001434 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='none' />
score ='AV:L/AC:L/Au:S/C:C/I:C/A:C' paramvalue='extended
acl' />

<! GEN000000LNX001476 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='none' />
score ='AV:L/AC:L/Au:S/C:P/I:P/A:P' paramvalue='group
passwords' />

<! GEN000000LNX00380 >
<parameter distribution="bitflip"
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N' paramvalue='000' />
score ='AV:L/AC:L/AU:S/C:P/I:P/A:P' paramvalue='1**' />
score ='AV:L/AC:L/AU:S/C:P/I:P/A:P' paramvalue='*1*' />
```

```
score ='AV:L/AC:L/AU:S/C:P/I:P/A:P' paramvalue='**1' />

<! GEN000000LNX00400 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='root' />
score ='AV:L/AC:L/Au:S/C:P/I:P/A:P' paramvalue='user' />

<! GEN000000LNX00420 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N'
paramvalue='privileged group' />
score ='AV:L/AC:L/Au:S/C:P/I:P/A:P' paramvalue='non-
privileged group' />

<! GEN000000LNX00440 >
<parameter distribution="bitflip"
score ='AV:L/AC:L/AU:S/C:P/I:P/A:P'
paramvalue='****1**0*' />
score ='AV:L/AC:L/AU:S/C:P/I:P/A:P'
paramvalue='*******1*' />
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='****0**0*' />

<! GEN000000LNX00450 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='none' />
score ='AV:L/AC:L/Au:S/C:P/I:P/A:P' paramvalue='extended
acl' />

<! GEN000000LNX00480 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='root' />
score ='AV:L/AC:L/Au:S/C:P/I:P/A:P' paramvalue='user' />

<! GEN000000LNX00500 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='root' />
score ='AV:L/AC:L/Au:S/C:P/I:P/A:P' paramvalue='group' />

<! GEN000000LNX00520 >
<parameter distribution="bitflip"
score ='AV:L/AC:L/AU:S/C:P/I:P/A:P'
paramvalue='***11*00*' />
score ='AV:L/AC:L/AU:S/C:P/I:P/A:P'
paramvalue='******11*' />
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='***00*00*' />

<! GEN000000LNX00530 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='none' />
```

```
score ='AV:L/AC:L/Au:S/C:P/I:P/A:P' paramvalue='extended
acl' />

<!GEN000000LNX00560 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='insecure'
/>
score ='AV:L/AC:L/Au:S/C:C/I:C/A:C' paramvalue='secure' />

<!GEN000000LNX00600 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='no
default' />
score ='AV:L/AC:L/Au:S/C:C/I:C/A:C' paramvalue='first
user' />

<!GEN000000LNX00620 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='root' />
score ='AV:L/AC:L/Au:S/C:P/I:P/A:P' paramvalue='group' />

<!GEN000000LNX00640 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='root' />
score ='AV:L/AC:L/Au:S/C:C/I:P/A:P' paramvalue='group' />

<! GEN000000LNX00660 >
<parameter distribution="bitflip"
score ='AV:L/AC:L/AU:S/C:P/I:P/A:P'
paramvalue='***11*00*' />
score ='AV:L/AC:L/AU:S/C:P/I:P/A:P'
paramvalue='******11*' />
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='***00*00*' />

<!GEN000000LNX00720 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='root' />
score ='AV:L/AC:L/Au:S/C:P/I:P/A:P' paramvalue='group' />

<!GEN000020 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='root' />
score ='AV:L/AC:L/Au:S/C:C/I:C/A:C' paramvalue='group' />

<!GEN000240 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='sync' />
score ='AV:L/AC:L/Au:S/C:N/I:P/A:P' paramvalue='no sync'
/>
```

```
<! GEN000241 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='sync' />
score ='AV:L/AC:L/Au:S/C:N/I:P/A:P' paramvalue='no sync'
/>

<! GEN000242 >
<parameter distribution="uniform"
feasibleRangeEnd="2" feasibleRangeStart="0"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='1' />
score ='AV:L/AC:L/Au:S/C:N/I:P/A:P' paramvalue='2' />

<! GEN000250 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='root' />
score ='AV:L/AC:L/Au:S/C:N/I:N/A:P' paramvalue='user' />

<! GEN000251 >
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='root' />
score ='AV:L/AC:L/Au:S/C:N/I:N/A:P' paramvalue='user' />

<! GEN000252 >
<parameter distribution="bitflip"
score ='AV:L/AC:L/AU:S/C:N/I:N/A:P'
paramvalue='****1**0*' />
score ='AV:L/AC:L/AU:S/C:N/I:N/A:P'
paramvalue='*******1*' />
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='****0**0*' />

<! GEN000253>
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='none' />
score ='AV:L/AC:L/Au:S/C:N/I:N/A:P' paramvalue='extended
acl' />

<! GEN000280>
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='not
permitted' />
score ='AV:L/AC:L/Au:S/C:P/I:P/A:P'
paramvalue='permitted' />

<!GEN000340 Account 1>
<parameter distribution="uniform"
feasibleRangeEnd="10000" feasibleRangeStart="0"
score ='AV:L/AC:L/Au:S/C:N/I:P/A:P' paramvalue='999' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='1000' />

<!GEN000340 Account 2>
```

```
<parameter distribution="uniform"
feasibleRangeEnd="10000" feasibleRangeStart="0"
score ='AV:L/AC:L/Au:S/C:N/I:P/A:P' paramvalue='999' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='1000' />

<!GEN000340 Account 3>
<parameter distribution="uniform"
feasibleRangeEnd="10000" feasibleRangeStart="0"
score ='AV:L/AC:L/Au:S/C:N/I:P/A:P' paramvalue='999' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='1000' />

<!GEN000340 Account 4>
<parameter distribution="uniform"
feasibleRangeEnd="10000" feasibleRangeStart="0"
score ='AV:L/AC:L/Au:S/C:N/I:P/A:P' paramvalue='999' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='1000' />

<!GEN000340 Account 5>
<parameter distribution="uniform"
feasibleRangeEnd="10000" feasibleRangeStart="0"
score ='AV:L/AC:L/Au:S/C:N/I:P/A:P' paramvalue='999' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='1000' />

<!GEN000360 Account 1>
<parameter distribution="uniform"
feasibleRangeEnd="10000" feasibleRangeStart="0"
score ='AV:L/AC:L/Au:S/C:N/I:P/A:P' paramvalue='999' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='1000' />

<!GEN000360 Account 2>
<parameter distribution="uniform"
feasibleRangeEnd="10000" feasibleRangeStart="0"
score ='AV:L/AC:L/Au:S/C:N/I:P/A:P' paramvalue='999' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='1000' />

<!GEN000360 Account 3>
<parameter distribution="uniform"
feasibleRangeEnd="10000" feasibleRangeStart="0"
score ='AV:L/AC:L/Au:S/C:N/I:P/A:P' paramvalue='999' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='1000' />

<!GEN000360 Account 4>
<parameter distribution="uniform"
feasibleRangeEnd="10000" feasibleRangeStart="0"
score ='AV:L/AC:L/Au:S/C:N/I:P/A:P' paramvalue='999' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='1000' />

<!GEN000360 Account 5>
<parameter distribution="uniform"
feasibleRangeEnd="10000" feasibleRangeStart="0"
score ='AV:L/AC:L/Au:S/C:N/I:P/A:P' paramvalue='999' />
```

```
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='1000' />

<! GEN000400>
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='nonDoD' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='DoD' />

<! GEN000402>
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='nonDoD' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='DoD' />

<! GEN000410>
<parameter distribution="uniform"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='nonDoD' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='DoD' />

<! GEN000450>
<parameter distribution="uniform"
feasibleRangeEnd="MaxInt" feasibleRangeStart="0"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='9' />
score ='AV:L/AC:L/Au:S/C:P/I:P/A:P' paramvalue='10' />

<! GEN000452>
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='not
displayed' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N'
paramvalue='displayed' />

<! GEN000460>
<parameter distribution="uniform"
feasibleRangeEnd="MaxInt" feasibleRangeStart="0"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='9' />
score ='AV:L/AC:L/Au:S/C:P/I:P/A:P' paramvalue='10' />

<! GEN000480>
<parameter distribution="uniform"
feasibleRangeEnd="MaxInt" feasibleRangeStart="0"
score ='AV:N/AC:L/Au:S/C:C/I:C/A:C' paramvalue='2' />
score ='AV:N/AC:L/Au:S/C:P/I:P/A:P' paramvalue='3' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='4' />

<! NEEDS RANGE GEN000500>
<parameter distribution="uniform"
feasibleRangeEnd="MaxInt" feasibleRangeStart="0"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='15' />
score ='AV:L/AC:L/Au:S/C:C/I:C/A:C' paramvalue='16' />

<! NEEDS RANGE GEN0005002>
<parameter distribution="uniform"
```

```
feasibleRangeEnd="MaxInt" feasibleRangeStart="0"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='15' />
score ='AV:L/AC:L/Au:S/C:C/I:C/A:C' paramvalue='16' />

<! GEN0005003>
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='enabled'
/>
score ='AV:L/AC:L/Au:S/C:C/I:C/A:C' paramvalue='disabled'
/>

<! GEN000510>
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='hidden' />
score ='AV:L/AC:L/Au:S/C:P/I:N/A:N' paramvalue='viewable'
/>

<! GEN000520>
<parameter distribution="uniform"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='user' />
score ='AV:L/AC:L/Au:S/C:C/I:C/A:C' paramvalue='root' />

<! GEN000540>
<parameter distribution="uniform"
feasibleRangeEnd="MaxInt" feasibleRangeStart="0"
score ='AV:L/AC:H/Au:S/C:C/I:C/A:C' paramvalue='24' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='25' />

<! GEN000580>
<parameter distribution="uniform"
feasibleRangeEnd="MaxInt" feasibleRangeStart="0"
score ='AV:L/AC:H/Au:M/C:C/I:C/A:C' paramvalue='13' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='14' />

<! GEN000900>
<parameter distribution="uniformOption"
score ='AV:L/AC:M/Au:S/C:P/I:P/A:P' paramvalue='home' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='other' />

<! GEN000920 >
<parameter distribution="bitflip"
score ='AV:L/AC:H/AU:M/C:N/I:N/A:N'
paramvalue='*00*00*00' />
score ='AV:L/AC:L/AU:N/C:C/I:C/A:C'
paramvalue='*1*******' />
score ='AV:L/AC:L/AU:N/C:C/I:C/A:C'
paramvalue='*01******' />
score ='AV:L/AC:L/AU:N/C:C/I:C/A:C'
paramvalue='*00*1****' />
score ='AV:L/AC:L/AU:N/C:C/I:C/A:C'
paramvalue='*00*01***' />
```

```
score ='AV:L/AC:L/AU:N/C:C/I:C/A:C'
paramvalue='*00*00*1*' />
score ='AV:L/AC:L/AU:N/C:C/I:C/A:C'
paramvalue='*00*00*01' />

<! GEN000930>
<parameter distribution="uniformOption"
score ='AV:L/AC:L/Au:N/C:C/I:C/A:C' paramvalue='extended
acl' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='none' />

<! GEN000940>
<parameter distribution="uniformOption"
score ='AV:L/AC:L/Au:N/C:C/I:C/A:C'
paramvalue='unrestricted' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N'
paramvalue='restricted' />

<! GEN000945>
<parameter distribution="uniformOption"
score ='AV:L/AC:L/Au:N/C:C/I:C/A:C'
paramvalue='unrestricted' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N'
paramvalue='restricted' />

<! GEN000950>
<parameter distribution="uniformOption"
score ='AV:L/AC:L/Au:N/C:C/I:C/A:C'
paramvalue='preloaded' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='none' />

<! GEN000980>
<parameter distribution="uniformOption"
score ='AV:N/AC:L/Au:S/C:C/I:C/A:C' paramvalue='anywhere'
/>
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='console'
/>

<! GEN001000>
<parameter distribution="uniformOption"
score ='AV:N/AC:L/Au:N/C:C/I:C/A:C' paramvalue='none' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='secure' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='disabled'
/>

<! GEN001020>
<parameter distribution="uniformOption"
score ='AV:L/AC:L/Au:N/C:C/I:C/A:C' paramvalue='allowed'
/>
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='denied' />
```

```
<! GEN001100>
<parameter distribution="uniformOption"
score ='AV:N/AC:L/Au:N/C:C/I:C/A:C'
paramvalue='cleartext' />
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N'
paramvalue='encrypted' />

<! GEN001120>
<parameter distribution="uniformOption"
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='anywhere'
/>
score ='AV:L/AC:H/Au:M/C:N/I:N/A:N' paramvalue='terminal'
/>
```

Vita

Robert Walter Smith

BORN: September 11, 1988, Elizabethton Tennessee

UNDERGRADUATE STUDY:

Wake Forest University

Winston-Salem, North Carolina

B. S. Computer Science, 2011

GRADUATE STUDY:

Wake Forest University

Winston-Salem, North Carolina

M. S. Computer Science, 2014

PROFESSIONAL SOCIETIES:

Upsilon Pi Epsilon, 2010-2014

PUBLICATIONS:

"Evolutionary Moving Target Cyber Defense", Proceedings of the Workshop on Genetic and Evolutionary Computation in Defense, Security, and Risk Management, GECCO 2014

AWARDS:

Finalist in the Wake Forest Graduate School 3 Minute Thesis Competition, 2014