

Running Head: Genetic Algorithms and Network Intrusion Detection

Genetic Algorithms and Network Intrusion Detection

Mark McFadden

MBI 640 – Data Communications & Network Security

Northern Kentucky University

Introduction

With the increase in network based attacks in general, and the world-wide access to computer networks and systems in particular, those responsible for network and computer system security need to utilize every tool available. Within this paper, the following will be examined, defined, and demonstrated. First, this paper will examine network intrusion detection. Then, genetic algorithms will be discussed. After this, the combining of genetic algorithms with intrusion detection will be reviewed. Finally, future steps will be discussed in the use of a genetic algorithm within intrusion detection.

Network Intrusion Detection

What is Network Intrusion Detection? According to Matthew Berge:

Network based intrusion detection attempts to identify unauthorized, illicit, and anomalous behavior based solely on network traffic. A network IDS, using either a network tap, span port, or hub collects packets that traverse a given network. Using the captured data, the IDS system processes and flags any suspicious traffic.

The goal of intrusion detection is to recognize attempts to sabotage in-place security controls (Berge). Specifically, network traffic is analyzed in search for system based breaches. Network breaches can take various forms. Next, an example intrusion is provided.

Example Intrusion (Denial of Service [DoS] Attack)

One such intrusion is a Denial of Service (DoS) attack. A DoS attack is typically an attempt by an attacker to prevent valid users of a service from using or having access to that service (CERT). Examples include:

- *attempts to "flood" a network, thereby preventing legitimate network traffic*
- *attempts to disrupt connections between two machines, thereby preventing access to a service*
- *attempts to prevent a particular individual from accessing a service*
- *attempts to disrupt service to a specific system or person (CERT)*

A popular Distributed DoS (DDoS) attempt is the Stacheldraht (German for "barbed wire") attack (See Figure 1 below). The assaults are directly activated by Trojan horse programs which are located on compromised systems within a network. Up to 1000 of these programs on the compromised systems can be controlled by a single remote system which is then activated by a remote client (Symantec).

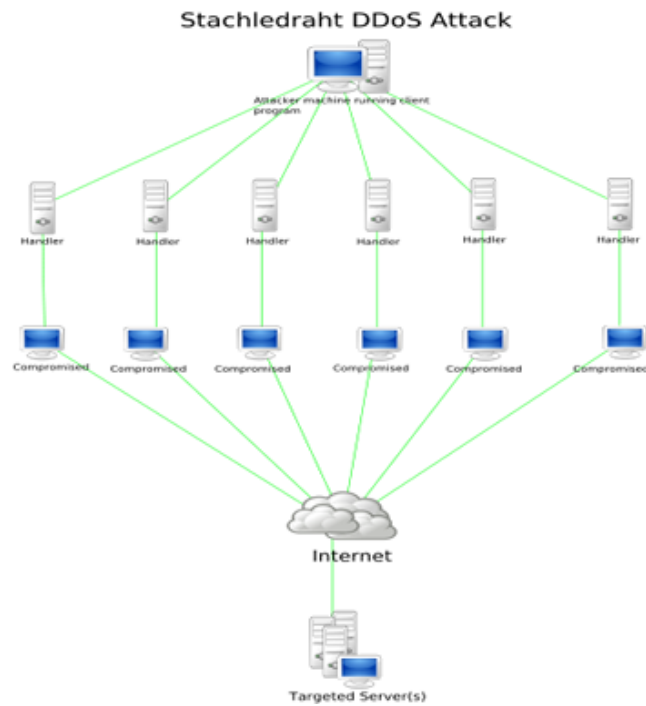


Figure 1. Stacheldraht DDoS Attack Diagram (Wikimedia)

Combining Genetic Algorithms and Intrusion Detection

Genetic Algorithms

Genetic Algorithms are utilized in various areas of data analytics and problem solving. A branch of machine learning:

Genetic algorithm is a family of computational models based on principles of evolution and natural selection. These algorithms convert the problem in a specific domain into a model by using a chromosome-like data structure and evolve the chromosomes using selection, recombination, and mutation operators. (Li, 2004, p. 1)

Moreover, genetic algorithms (GA) are good tools for acquiring optimized solutions and their use with determining rule sets for potential and actual network intrusions is both intuitive and potentially valuable (Li, p. 2). Given the above definition of a GA, an example suspect log record that shows a potential network intrusion will be reviewed. Then, the structuring of the domain-problem based chromosome will be discussed. Finally, how the GA is used in assisting rule set creation for a potential network intrusion will be examined.

Suspected Intrusion Log Record

Firewall devices are the typical the first point of entry within computer networks. Here (see Table 1), we will look at a typical content of a firewall log entry.

Time	Local time on the management station
Action	Accept, deny, or drop. Accept=accept or pass the packet. Deny=send TCP reset or ICMP port unreachable message. Drop=drop packet with no error to
Firewall	IP address or hostname of the enforcement point
Interface	Firewall interface on which the packet was seen
Product	Firewall software running on the system that generated the message

Source	Source IP address of packet sender
Destination	Destination IP address of packet
Service	Destination port or service of packet
Protocol	Usually layer 4 protocol of packet - TCP, UDP, etc.
Translation	If address translation is taking place, this field shows the new source or destination address. This only shows if NAT is occurring.
Rule	Rule number from the GUI rule base that caught this packet, and caused the log entry. This should be the last field, regardless of presence or absence of other fields except for resource messages.

Table 1. – Firewall Log Entry Content (A Firewall Log Analysis Primer).

For our purposes in the creation of input data for the Genetic Algorithm, we will look at the following example firewall log entry in Table 2.

Source IP	Source IP address of packet sender
Destination IP	Destination IP address of packet
Destination Port	Destination port or service of packet
Protocol	Usually layer 4 protocol of packet - TCP, UDP, etc.
Bytes Sent by Originator	The number of bytes in the request from the source system.
Bytes Sent by Responder	The number of bytes returned from the responding or target system.

Table 2.

Structuring the Domain-Problem Chromosome

A typical IDS rule would take the form of the following condition statement:

if {the connection has following information: source IP 125.19.54.155; destination IP address: 109.1.1.0 ~ 109.1.255.255; destination port number: 8184; the protocol used is FTP; the originator sent more than 10,000 bytes of data; and the responder sent more than 250,000 bytes of data }
then {stop the connection} (Li, p. 2).

Given that the input value modeled in Table 2 above is similar to a desired IDS Rule

Set, the input rule will be the model for the chromosome-like data structure. This input

rule within the GA will then be evolved into a fitter output, or as in this case, an IDS Rule.

For a clearer view of the IDS rule, note Table 3 below. The *Attribute* column takes the contents of the above condition and provides labels. The *Range of Values* column shows the lower and upper bounds of the rule. The suspect source rule set is displayed in the Example Values column. The *Descriptions* column displays what each item is in the suspect rule and the justification for why the rule may be a potential threat to the network and/or the systems that are nodes within a network.

Attribute	Range of Values	Example Values	Descriptions
Source IP address	125.0.0.0 ~ 125.150.255.255	125.19.54.155	125.19.54.155 is a suspect IP address.
Destination IP address	119.0.0.0 ~ 119.150.255.255	119.1.1.20	IP Address 119.1.1.17 ~ 119.1.1.21 are database servers.
Destination Port Number	0 ~ 65535	8184	Destination port number, indicates this is a http service. 8184 is for internal data access.
Protocol	1 ~ 20	5	The protocol for this connection FTP. 5 = FTP.
Number of Bytes Sent by Originator	0 ~ 250 KB	10.5 KB	The originator sends 7,500 bytes of data
Number of Bytes sent by Responder	0 ~ 1 MB	2.5 MB	The responders sends 250,000 bytes of data

Table 3. Rule definition for connection and range of values of each field (Sinclair, Pierce, & Matzner, p. 4)

In order for a particular domain to be suitable for a genetic algorithm the domain must be converted into numeric values, either within the GA or as raw input. These numeric values are sometimes referred to as genes and are changed at random within a range during an evolutionary cycle. The set of chromosomes during a stage of

evolution are called a population (Li, p. 1). A fitness function is used to calculate the “goodness” of each chromosome. We can convert the above example into a chromosome form, with each row as a “gene,” as described below in Table 4 below:

Source IP address	125.19.54.155 converted to 2006384639
Destination IP address	109.1.1.20 converted to 1996554516
Destination Port Number	8184
Protocol	5
Number of Bytes Sent by Originator	10500
Number of Bytes sent by Responder	2500000

Table 4.

Therefore, based on the above information, here is the example chromosome for the network intrusion detection GA:

Source IP	Destination IP	Destination Port	Protocol	Originator Bytes	Responder Bytes
2006384639	1996554516	8184	5	10500	2500000

Genetic Algorithm Use in Rule Set Creation

Now, the structure of a potential GA within Network Intrusion Rule Set Creation will be detailed. Figure 2 below shows the structure of a basic genetic algorithm. First, the GA creates a random population which is then evaluated concerning its level of fitness in a Fitness Function.

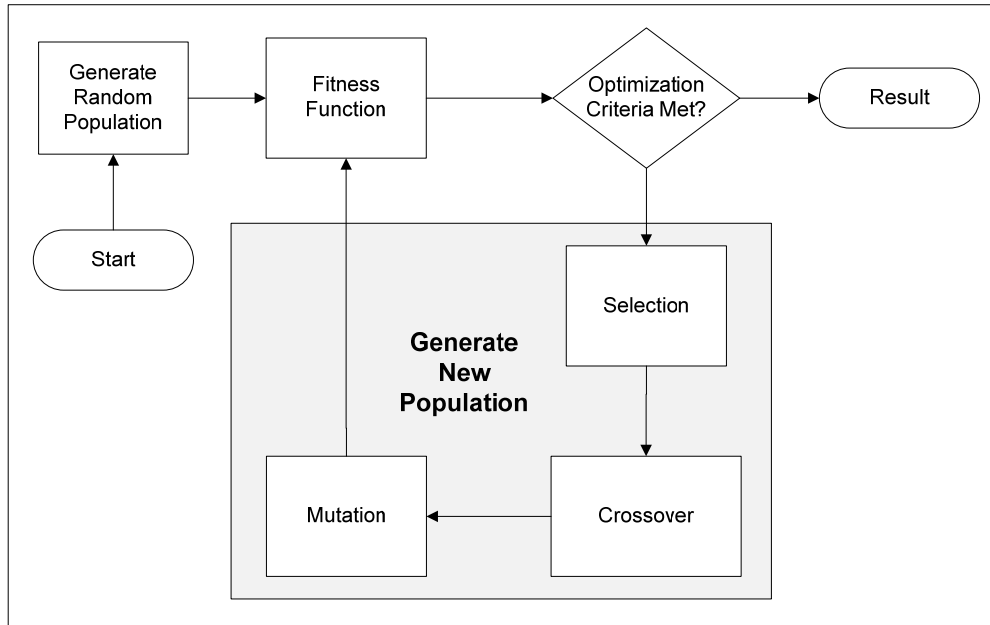


Figure 2. (Pohlheim, 2001)

Fitness Function

A GA Fitness Function typically has the following or similar steps. First, the general outcome is determined based on whether a gene (or allele) “matches” an existing data set of suspect log record that was obtained from a network device such as a firewall. Then, the function multiplies the “weight” of that field to the degree that the field value “matched” the suspect record field. Typically, the “match” value is either 1 or 0 (See Figure 3).

$$\text{Outcome} = \sum_{i=1}^6 \text{Matched} * \text{Weight}_i$$

Figure 3. (Li, p. 4).

Weight values are applied to the different genes as historically reported by network devices. For example, if the Destination IP gene historically demonstrates to be

a consist predictor of a network intrusion, its weight will be more than the other genes. Moreover, all particular genes types have the same weight value so all Protocol genes have a weight of 15, regardless of their degree of being a suspect record (See Figure 4).

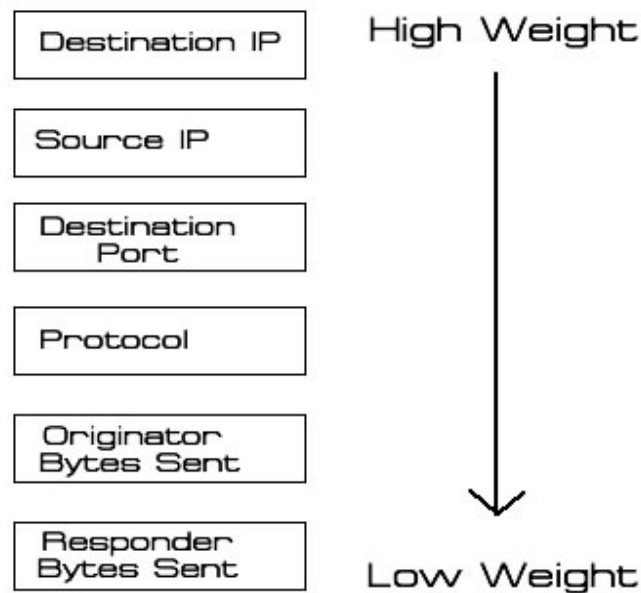


Figure 4. (Li, p. 4).

As a clarifying example, in the case of a “gene” such as a Source IP address, let us suppose that historic data from an organization’s border hardware devices such as its firewalls reveal that a Source IP address of 125.19.54.155 has attempted various intrusions targeting valuable assets such as a cluster of database systems. If the weight of a Source IP was 10, and given that the historic data supports a “match” value of 1, the outcome of the Source IP gene is 10 ($10 = 1 \cdot 10$).

Next, the delta value or absolute difference between the “outcome” of the chromosome and the suspicion_level is then computed using the following equation (See Figure 5).

$$\Delta = | \text{outcome} - \text{suspicion_level} |$$

Figure 5. (Li, p. 5).

The suspicion_level is a value that indicates if the historical gene value and the suspicious gene value are considered a “match” from historic log data. Continuing with our previous example, given that the Source IP of 125.19.54.155 was determined to be a suspicious IP address, the suspicion_level value would be higher with a value such as 8. Therefore, the delta result is a low number of 2 ($2 = |10 - 8|$).

If the delta level is high enough, a penalty value is calculated using this delta (or absolute difference) (See Figure 6). The “ranking” in the equation below indicates whether or not a network intrusion is easy to establish. Historical data should determine the value of the ranking. For example, given that Destination IP addresses of certain asset systems are well known by those within an organization, this ranking would be higher.

$$\text{penalty} = \left(\frac{\Delta * \text{ranking}}{100} \right)$$

Figure 6. (Li, p. 5).

Finally, the chromosome’s fitness is then computed using the above penalty. The scope of the fitness result is between 0 and 1 (See Figure 7).

$$\text{fitness} = 1 - \text{penalty}$$

Figure 7. (Li, p. 5).

Dr. Richard Fox, Associate Professor and Graduate Program Director in the Department of Computer Science at Northern Kentucky University, informed this student (by

personal communication, November 12, 2008) that the selection process detailed here utilizes a stochastic process which results in a random outcome allowing a range of possibilities (Stochastic Process). Dr. Fox also pointed out that there are other selection strategies that could be utilized such as those that result in the fittest chromosome in combination with the most diverse or the fittest chromosomes only.

In summary, following the running of the fitness function within the GA, the fitness level is reviewed. If the desired fitness level is not obtained, the algorithm then evolves through the selection, crossover (recombination), and mutation functions.

Selection

Once the initial population (of chromosomes) is evaluated, the GA experiments with new generations and iteratively refines the initial outcomes so that those that are most fit are more probable to be ranked higher as results. The objective is to produce new generation of chromosomes to evaluate. (Marakas, 2003, p. 142)

Crossover

In essence, the crossover operation creates new chromosomes that share optimistic characteristics of the parent chromosomes while at the same time lowering the negative attributes in a child chromosome (Marakas, 2003, p. 143). Figure 8 below provides an example of a crossover of chromosomes from the parents to their offspring.

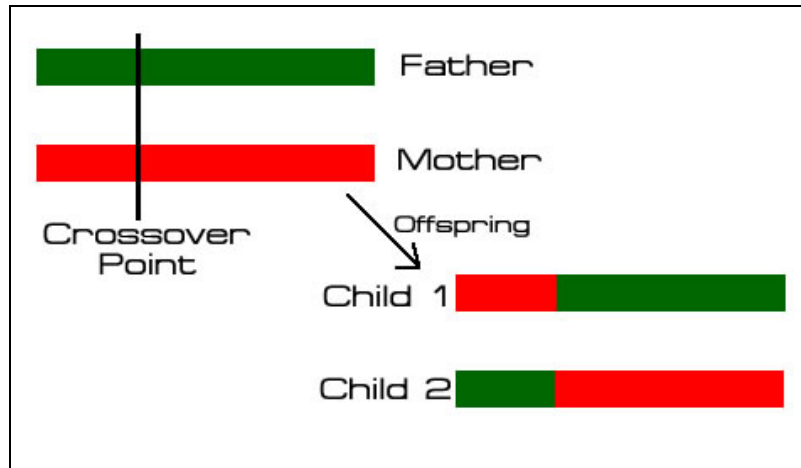


Figure 8. Crossover

Although this step is typical in most genetic algorithms, in the case of this project's chromosome (see Table 4 above) the crossover operation may not be beneficial. While a Source or Destination IP may be bound by upper and lower IP settings (as demonstrated in Table 3 above), a crossover of the IP octet values would probabilistically not be advantageous. For example, the crossover of the parental values of 209.103.51.134 and 101.1.25.193 could result in child IP addresses of 209.103.25.193 and 101.1.51.134. However, the probability that this offspring will be potential suspicious Source or Destination IP addresses is low.

Mutation

The final step in the process of generating a new population is mutation. This phase randomly alters a gene's value to create a different one (Marakas, 2003, p. 143). Figure 9 below details how a gene's (or allele's) value is changed and thereby creating a new chromosome. Concerning the applicability of this step with the network intrusion chromosome, as was the case in the crossover step above the probability of useful outcomes is minimal.

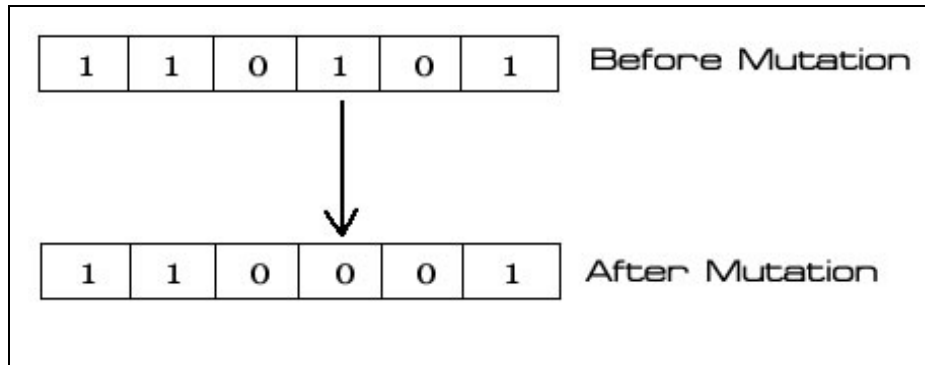


Figure 9. Mutation

The Rule Set

Essentially, the rule set is produced from the output of the GA. For example, the input of Source IP = 1829975662 (which is an IPv4 address of 109.19.54.110) | Destination IP = 1828782356 (which is an IPv4 address of 109.1.1.20) | Destination Port = 8184 | Protocol = 5 | Originator Bytes = 10500 | Responder Bytes = 250000 could produce the following rule:

if {the connection has following information: source IP 125.19.54.155; destination IP address: 119.1.1.17 ~ 119.1.1.21; destination port number: 8184; the protocol used is FTP; the originator sent more than 10,000 bytes of data; and the responder sent more than 250,000 bytes of data } then {log the intrusion and stop the connection}

Note that the Source and Destination IP input values above are a Java “double” primitive data type. This was needed to convert an IPv4 address to a Java primitive for its use in the GA.

Creating an IDS Rule Set Genetic Algorithm

Given that this student is a software developer, a test scenario to better understand the particular workings of a GA was created. After corresponding with some of the authors of the cited sources and searching for available source-code of a genetic

algorithm, I noted that I would have to develop a GA for IDS rule set creation. It should be noted that the GA steps covered above are not the exact steps that this student utilized in his genetic algorithm. What was obtained was an open-source, Java-language based genetic algorithm framework entitled JGAP, which stands for Java Genetic Algorithms Package (Meffert et al.). According to the JGAP documentation:

JGAP is designed to do almost all of the evolutionary work for you in a relatively generic fashion. However, it has no knowledge of the specific problem you're actually trying to solve, and hence has no intrinsic way of deciding if one potential solution is any better than another potential solution for your specific problem. That's where the fitness function comes in: it's a single method that you must implement that accepts a potential problem solution and returns an integer value that indicates how good (or "fit") that solution is relative to other possible solutions. The higher the number, the better the solution. The lower the number (1 being the lowest legal fitness value), the poorer the solution. JGAP will use these fitness measurements to evolve the population of solutions toward a more optimal set of solutions. (Meffert et al.)

This student's code consists of a function class (see Appendix B) which contains the problem domain and fitness code, a value class that holds the IPv4 chromosome attributes (see Appendix C), and the class containing the main method for configuration and the fitness function call (see Appendix A). This student's fitness function works with the JGAP framework by creating the six genes (see Table 4 above) and then compares them with a suspicious range of values for those genes. If they match, the outcome or returned value is set to a maximum amount for the gene. If not, then a lower value is returned. It should be noted that the "suspicious" values as well as the "weight" values are what I hard coded for test and demonstration purposes. Finally, the JGAP framework takes each gene value and returns "a more optimal" result for the genes.

Running the GA produced the following output (in Figure 10.) from the input of Source IP = 2098411163 (which is an IPv4 address of 125.19.54.155)| Destination IP = 1828782356 (which is an IPv4 address of 109.1.1.20)| Destination Port = 8184 | Protocol = 5 | Originator Bytes = 10500 | Responder Bytes = 2500000:

Source IP	125.53.8.22
Destination IP	119.19.14.44
Destination Port	50649
Protocol	7
Originator Bytes	184572
Responder Bytes	63960

Figure 10.

Please see Appendixes A, B, and C for the Java Class files of IDS.java, IDSFitnessFunction.java, and IPv4Chromosome.java for the configuration, value, and function code. In summary, the output from the GA could be base data to update a router table or firewall application making known to the system(s) that a potential intrusion is being attempted.

Conclusion

In conclusion, one can see how that a genetic algorithm can be useful in the creation of rule sets to detect network intrusions. Moreover, genetic algorithms are potential tools for optimized rules sets and the determination of potential and actual network intrusions. However, the primary author or the JGAP framework shared with this student that a more multifaceted and flexible outcomes could be obtained if Genetic Programming were utilized (Meffert).

Future Steps

Based on the findings of applying the JGAP Framework to a theoretical network intrusion, some future steps could be:

- Utilize actual anomaly based network device data to form the input “chromosomes”, the gene range values, and the parameters for the evaluation function (See Figure 11 below).

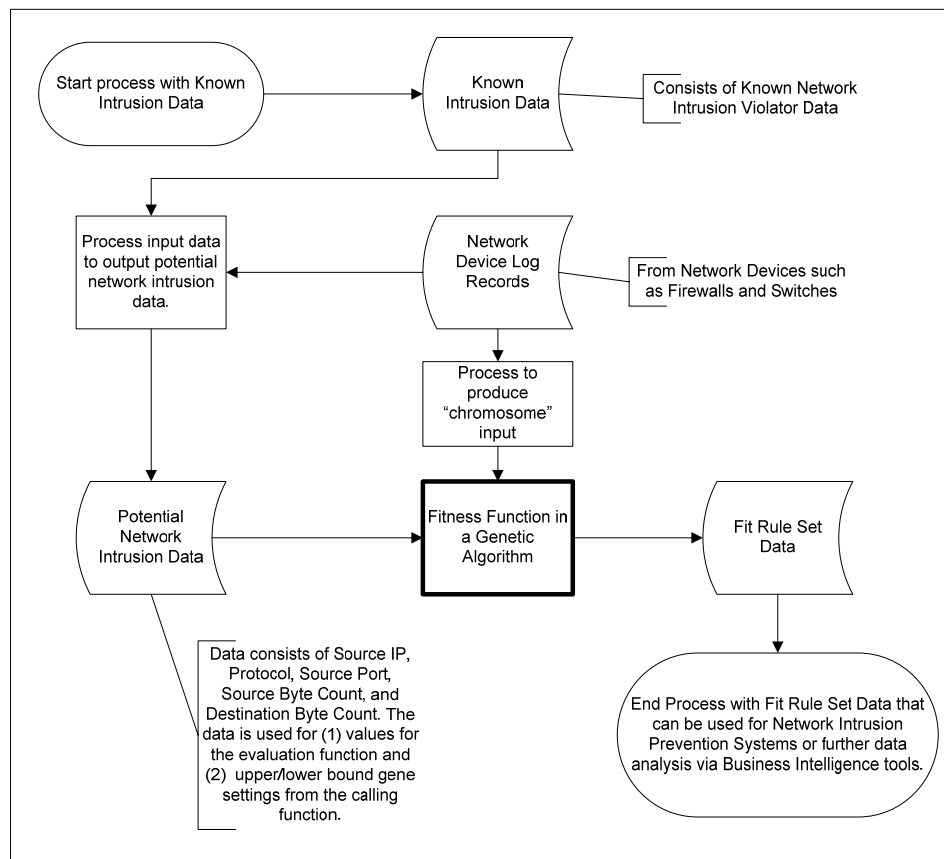


Figure 11.

- Compare GA results with existing intrusion rule sets for effectiveness.
- Data mine the GA results for patterns or data clusters and then analyze for discoveries.

- Utilize Genetic Programming, which enhance GAs since they produce dynamic programs instead of static chromosomes, which result in more multifaceted and flexible outcomes (Meffert).

References

- Berge, Matthew. *Intrusion Detection FAQ: What is Intrusion Detection?* Retrieved 2-29-08 from http://www.sans.org/resources/idfaq/what_is_id.php.
- CERT. *Denial of Service Attacks*. Retrieved 10-3-2008 from http://www.cert.org/tech_tips/denial_of_service.html.
- Li, Wei. (2004). Using Genetic Algorithm for Network Intrusion Detection. Retrieved from: <http://www.security.cse.msstate.edu/docs/Publications/wli/DOECSG2004.pdf>. Department of Computer Science and Engineering, Mississippi State University.
- Marakas, George M. (2003). *Modern Data Warehousing, Mining, and Visualization: Core Concepts*. Upper Saddle River, NJ: Pearson Education.
- Meffert, Klaus et al.: JGAP - Java Genetic Algorithms and Genetic Programming Package. URL: <http://jgap.sf.net>.
- Pohlheim, Hartmut. (2001). *Genetic and Evolutionary Algorithms: Principles, Methods and Algorithms*. Genetic and Evolutionary Algorithm Toolbox. Retrieved 10-11-2008 from: <http://www.geatbx.com/docu/algindex.html>.
- Secure Networks. (2008). *A Firewall Log Analysis Primer*. Retrieved 10-11-2008 from: <http://www.secureworks.com/research/articles/firewall-primer>.
- Sinclair, Chris., Pierce, L., & Matzner, S. (1999). *An Application of Machine Learning to Network Intrusion Detection*. Retrieved 10-11-2008 from: <http://www.acsac.org/1999/papers/fri-b-1030-sinclair.pdf>.
- Stochastic Process. (2008). Retrieved 11-12-2008 from: http://en.wikipedia.org/wiki/Stochastic_processes.
- Symantec. *DDOS Stacheldraht*. Retrieved 10-3-2008 from: http://www.symantec.com/business/security_response/attacksignatures/detail.jsp?asid=20009.
- Wikimedia. *Stacheldraht PNG*. Retrieved 10-3-2008 from: http://commons.wikimedia.org/wiki/Image:Stachledraht_DDos_Attack.svg.

```

/*-----*/
/*-----APPENDIX A - IDS.java-----*/
/*-----*/

/*
 * This file is part of JGAP.
 *
 * JGAP offers a dual license model containing the LGPL as well as the MPL.
 *
 * For licensing information please see the file license.txt included with JGAP
 * or have a look at the top of class org.jgap.Chromosome which representatively
 * includes the JGAP license policy applicable for any file delivered with JGAP.
 */
package IDS;

import java.io.*;
import org.jgap.*;
import org.jgap.data.*;
import org.jgap.impl.*;
import org.jgap.xml.*;
import org.w3c.dom.*;
import IDS.IPv4Chromosome;

/**
 * This class provides an implementation of an Intrusion Detection Solution problem
 * using a genetic algorithm.
 * @author Neil Rotstan
 * @author Klaus Meffert
 * @since 1.0
 * @author revised by Mark McFadden
 */
public class IDS {
    /**
     * The total number of times we'll let the population evolve.
     */
    private static final int MAX_ALLOWED_GENERATIONS = 200;

    /**
     * Executes the genetic algorithm to the best rule set record for IDS/IDP systems.
     * The solution will then be written to System.out.
     * @param source IP

```

```

* @param target IP
* @param target Port
* @param protocol
* @param sender Bytes
* @param responder Bytes
* @throws Exception
*/
public static void checkIDSChromosome(String sourceIP, String targetIP,
    int targetPort, int protocol, int senderBytes, int responderBytes,
    boolean initialPopulationFromXMLDoc)
    throws Exception {

// Start with a DefaultConfiguration, which comes setup with the
// most common settings.
// -----
Configuration conf = new DefaultConfiguration();

// Care that the fittest individual of the current population is
// always taken to the next generation.
// Consider: With that, the pop. size may exceed its original
// size by one sometimes!
// -----
conf.setPreserveFittestIndividual(true);

// Set the fitness function we want to use, which is our
// MinimizingMakeChangeFitnessFunction. We construct it with
// the target amount of change passed in to this method.
// -----
FitnessFunction myFunc =
    new IDSFitnessFunction(sourceIP, targetIP, targetPort, protocol, senderBytes, responderBytes);
conf.setFitnessFunction(myFunc);

//set the crossover and mutation operators

/**
 * Constructs a new instance of this CrossoverOperator with
 * the given crossover rate. No new chromosomes are x-overed.
 */
CrossoverOperator xOverOperator = new CrossoverOperator(conf, 10, false);

/**
 * Constructs a new instance of this MutationOperator with the given mutation rate.

```

```

* a_desiredMutationRate is the desired rate of mutation, expressed as the denominator
* of the 1 / X fraction. For example, 1000 would result in 1/1000
* genes being mutated on average. A mutation rate of zero
* disables mutation entirely
*/
MutationOperator mutationOperator = new MutationOperator(conf, 50); //1/50 gene mutation rate

/**
 * Adds a genetic operator for use in this algorithm. Genetic operators
 * represent evolutionary steps that, when combined, make up the evolutionary
 * process. Examples of genetic operators are reproduction, crossover, and mutation.
 * During the evolution process, all of the genetic operators added via this method are
 * invoked in the order they were added. At least one genetic operator must be provided.
 */
conf.addGeneticOperator(xOverOperator);
conf.addGeneticOperator(mutationOperator);

/*
 * Minimum size guaranteed for population. This is significant during evolution as
 * natural selectors could select fewer chromosomes for the next
 * generation than the initial population size was.
 */
conf.setMinimumPopSizePercent(90);

conf.setSelectFromPrevGen(0.001); //avoid local maxima?

// Now we need to tell the Configuration object how we want our
// Chromosomes to be setup. We do that by actually creating a
// sample Chromosome and then setting it on the Configuration
// object. We want our Chromosomes to each have six genes, one for
// each detection log item. We want the values (alleles) of
// those genes to be integers and/or doubles (if allele is IPv4 address),
// which represent an IDS log record. We therefore use the
// DoubleGene and IntegerGene classes to represent each of the
// genes. These classes allow us specify a lower and upper bounds,
// which we set to sensible values for each part of the IDS data.
// -----
Gene[] sampleGenes = new Gene[6];
sampleGenes[0] = new DoubleGene(conf,
    //TODO: found out why result for byte[4] result is 127, -1, -1, -1 when first octet includes 127
    IDSFitnessFunction.getDoubleFromIPv4AddressString("125.0.0.0"), //lowest IP
    IDSFitnessFunction.getDoubleFromIPv4AddressString("125.150.255.255")); //upper IP

```

```

sampleGenes[1] = new DoubleGene(conf,
    IDSFitnessFunction.getDoubleFromIPv4AddressString("119.0.0.0"), //lower IP
    IDSFitnessFunction.getDoubleFromIPv4AddressString("119.150.255.255")); //upper IP

sampleGenes[2] = new IntegerGene(conf, 0, 65535); // Port
sampleGenes[3] = new IntegerGene(conf, 1, 20); // Protocol
sampleGenes[4] = new IntegerGene(conf, 0, 260000); // Sender Bytes
sampleGenes[5] = new IntegerGene(conf, 0, 750000); // Responder Bytes
IChromosome sampleChromosome = new Chromosome(conf, sampleGenes);
conf.setSampleChromosome(sampleChromosome);

// Finally, we need to tell the Configuration object how many
// Chromosomes we want in our population. The more Chromosomes,
// the larger number of potential solutions (which is good for
// finding the answer), but the longer it will take to evolve
// the population (which could be seen as bad).
// -----
conf.setPopulationSize(250);

// Create random initial population of Chromosomes.
// Here we try to read in a previous run via XMLManager.readFile(...)
// -----
Genotype population;
if(isInitialPopulationFromXMLDoc){
    try {
        Document doc = XMLManager.readFile(new File("JGAP_IDSOutput.xml"));
        population = XMLManager.getGenotypeFromDocument(conf, doc);
    }
    catch (UnsupportedOperationException uex) {
        // JGAP codebase might have changed between two consecutive runs.
        // -----
        population = Genotype.randomInitialGenotype(conf);
    }
    catch (FileNotFoundException fex) {
        population = Genotype.randomInitialGenotype(conf);
    }
}

// If you want to load previous results from file, comment the next line!
// -----
population = Genotype.randomInitialGenotype(conf);

```

```

// Evolve the population. Since we don't know what the best answer
// is going to be, we just evolve the max number of times.
// -----
long startTime = System.currentTimeMillis();
for (int i = 0; i < MAX_ALLOWED_GENERATIONS; i++) {
    if (!uniqueChromosomes(population.getPopulation())) {
        throw new RuntimeException("Invalid state in generation "+i);
    }
    population.evolve();
}

long endTime = System.currentTimeMillis();
System.out.println("Total evolution time: " + (endTime - startTime)
    + " ms");

// Save progress to file. A new run of this example will then be able to
// resume where it stopped before.
// -----
// Represent Genotype as tree with elements Chromosomes and Genes.
// -----
DataTreeBuilder builder = DataTreeBuilder.getInstance();
IDataCreators doc2 = builder.representGenotypeAsDocument(population);

// create XML document from generated tree
XMLDocumentBuilder docbuilder = new XMLDocumentBuilder();
Document xmlDoc = (Document) docbuilder.buildDocument(doc2);
XMLManager.writeFile(xmlDoc, new File("JGAP_IDSOutput.xml"));

// Display the best solution we found.
IChromosome bestSolutionSoFar = population.getFittestChromosome();
if (bestSolutionSoFar.getFitnessValue() < 1000.0){//just log this outcome as not a suspect
    System.out.println("This best solution has a fitness value of " +
        bestSolutionSoFar.getFitnessValue());
System.out.println("It contained the following output: ");
System.out.println("\tSource IP: " + IDSFitnessFunction.getSelectedResultForDouble(bestSolutionSoFar, 0).getHostAddress());
System.out.println("\tTarget IP: " + IDSFitnessFunction.getSelectedResultForDouble(bestSolutionSoFar, 1).getHostAddress());
System.out.println("\tPort: " +
    IDSFitnessFunction.getSelectedResult(bestSolutionSoFar, 2));
System.out.println("\tProtocol: " +
    getProtocol(IDSFitnessFunction.getSelectedResult(bestSolutionSoFar, 3)));
}

```

```

System.out.println("\tSent Bytes: " +
    IDSFTnessFunction.getSelectedResult(bestSolutionSoFar, 4));
System.out.println("\tResponse Bytes: " +
    IDSFTnessFunction.getSelectedResult(bestSolutionSoFar, 5));
} else{//this is a suspect record so produce potential rule set record
    System.out.println("This best solution has a fitness value of " +
        bestSolutionSoFar.getFitnessValue());
    System.out.println("This produced the following rule set: ");
    System.out.println("\tif {the connection has following information: ");
    System.out.println("\t\tsource IP address: " + IDSFTnessFunction.getInetAddressFromInt(
        IDSFTnessFunction.getSelectedResultForDouble(bestSolutionSoFar, 0)).getHostAddress() + ";");
    System.out.println("\t\tdestination IP address: " + IDSFTnessFunction.getInetAddressFromInt(
        IDSFTnessFunction.getSelectedResultForDouble(bestSolutionSoFar, 1)).getHostAddress() + ";");
    System.out.println("\t\tdestination port number: " +
        IDSFTnessFunction.getSelectedResult(bestSolutionSoFar, 2) + ";");
    System.out.println("\t\tthe protocol used is " +
        getProtocol(IDSFTnessFunction.getSelectedResult(bestSolutionSoFar, 3)) + ";");
    System.out.println("\t\tthe originator sent more than " +
        IDSFTnessFunction.getSelectedResult(bestSolutionSoFar, 4) + " bytes of data;");
    System.out.println("\t\tand the responder sent more than " +
        IDSFTnessFunction.getSelectedResult(bestSolutionSoFar, 5) + "bytes of data ");
    System.out.println("\t\tthen {log the intrusion and stop the connection}");
}
}

/**
 * Main method.
 *
 * @param args amount of change in cents to create
 * @throws Exception
 *
 * @author Neil Rotstan
 * @author Klaus Meffert
 * @author Mark McFadden
 * @since 1.0
 */
public static void main(String[] args)
    throws Exception {
    //TODO: modify signature to for input of IPv4 object array
    /*
    String sourceIP, String targetIP, int targetPort, int protocol, int senderBytes, int responderBytes
    */
}

```

```

//    if (args.Length != 6) {
//        System.out.println("Syntax: IDS <Source IP> <Target IP> <Target Port> <Protocol > <Max Sender
Bytes> <Max Responder Bytes>");
//    }
//    else {
//        String sourceIP = "";
//        String targetIP = "";
//        int targetPort = -1;
//        int protocol = 0;
//        int senderBytes = -1;
//        int responderBytes = -1;
try {
//TODO:Once completed testing, uncomment code for command prompt usage
//        sourceIP = args[0];
//        targetIP = args[1];
//        targetPort = Integer.parseInt(args[2]);
//        protocol = Integer.parseInt(args[3]);
//        senderBytes = Integer.parseInt(args[4]);
//        responderBytes = Integer.parseInt(args[5]);

//pass in a chromosome array object of 5 chromosomes
IPv4Chromosome[] chromosomeArr = new IPv4Chromosome[5];
chromosomeArr[0] = new IPv4Chromosome("125.19.54.155", "109.1.1.20", 8184, 5, 10500, 250000);
chromosomeArr[1] = new IPv4Chromosome("101.1.12.114", "129.1.1.19", 45, 12, 100, 15000);
chromosomeArr[2] = new IPv4Chromosome("111.23.1.105", "129.1.1.22", 445, 2, 100000, 1500000);
chromosomeArr[3] = new IPv4Chromosome("119.1.57.113", "129.1.1.10", 115, 11, 1000, 1500);
chromosomeArr[4] = new IPv4Chromosome("10.32.114.11", "129.1.1.15", 80, 15, 10000, 150000);

//variable to determine if we want to load initial population from previously populated XML Doc
boolean isInitialPopulationFromXMLDoc = false;

//loop through the array and display the "fittest" chromosome
for(int i = 0; i < chromosomeArr.Length; i++){
    System.out.println("*****");
    System.out.println("Displaying Chromosome " + (i + 1) + " Input");
    checkIDSChromosome(chromosomeArr[i].getSourceIP(), chromosomeArr[i].getTargetIP(),
        chromosomeArr[i].getPort(), chromosomeArr[i].getProtocol(), chromosomeArr[i].
getSenderByteCount(),
        chromosomeArr[i].getResponderBytCount(), isInitialPopulationFromXMLDoc);
    System.out.println("*****");
    System.out.println();
    Configuration.reset();
}

```

```

    }
    }
    catch (IllegalArgumentException e) {
        System.out.println(e.getMessage());
        System.exit(1);
    }
//    checkIDSChromosome(sourceIP, targetIP, targetPort, protocol, senderBytes, responderBytes);
//    }
}

/**
 * @param a_pop the population to verify
 * @return true if all chromosomes in the population are unique
 *
 * @author Klaus Meffert
 * @since 3.3.1
 */
public static boolean uniqueChromosomes(Population a_pop) {
    // Check that all chromosomes are unique
    for(int i=0; i<a_pop.size()-1; i++) {
        IChromosome c = a_pop.getChromosome(i);
        for(int j=i+1; j<a_pop.size(); j++) {
            IChromosome c2 = a_pop.getChromosome(j);
            if (c == c2) {
                return false;
            }
        }
    }
    return true;
}

/**
 * provide the protocol value
 * @author Mark McFadden
 * @param protocol
 * @return
 */
private static String getProtocol(int protocol){
    String protocolStr = "";
    switch (protocol){
        case 1: protocolStr = "HTTP";
            break;
    }
}

```

```
case 2: protocol Str = "SMTP";
    break;
case 3: protocol Str = "TCP";
    break;
case 4: protocol Str = "NNTP";
    break;
case 5: protocol Str = "FTP";
    break;
case 6: protocol Str = "DNS";
    break;
case 7: protocol Str = "DHCP";
    break;
case 8: protocol Str = "TCP";
    break;
case 9: protocol Str = "FINGER";
    break;
case 10: protocol Str = "GNUTELLA";
    break;
case 11: protocol Str = "IMAP";
    break;
case 12: protocol Str = "IRC";
    break;
case 13: protocol Str = "JABBER";
    break;
case 14: protocol Str = "LDAP";
    break;
case 15: protocol Str = "MIME";
    break;
case 16: protocol Str = "POP3";
    break;
case 17: protocol Str = "SNMP";
    break;
case 18: protocol Str = "SOAP";
    break;
case 19: protocol Str = "Telnet";
    break;
case 20: protocol Str = "SSH";
    break;
default: protocol Str = "Unknown";
}

return protocol Str;
```

} }

```

/*-----*/
/*=====APPENDIX B - IDSFitnessFunction.java=====*/
/*-----*/

/*
 * This file is part of JGAP.
 *
 * JGAP offers a dual license model containing the LGPL as well as the MPL.
 *
 * For licensing information please see the file license.txt included with JGAP
 * or have a look at the top of class org.jgap.Chromosome which representatively
 * includes the JGAP license policy applicable for any file delivered with JGAP.
 */
package IDS;

import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Random;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.jgap.*;

/**
 * Fitness function for the IDS MBI-640 class.
 * @author Neil Rotstan and Klaus Meffert
 * @since 1.0
 * @author revised by Mark McFadden
 */
public class IDSFitnessFunction extends FitnessFunction {

    private static final long serialVersionUID = 1L;

    private final long m_sourceIP; //input from user
    private final long m_targetIP; //input from user
    private final int m_targetPort; //input from user
    private final int m_protocol; //input from user
    private final int m_senderBytes; //input from user
    private final int m_responderBytes; //input from user

    public static final int MAX_BOUND = new Random().nextInt(200);
    public static final int SOURCE_IP_MAX_BOUND = 1500;
    public static final int TARGET_IP_MAX_BOUND = 2000;

```

```

public static final int TARGET_PORT_MAX_BOUND = 1000;
public static final int PROTOCOL_MAX_BOUND = 750;
public static final int SENDER_BYTES_MAX_BOUND = 500;
public static final int RESPONDER_BYTES_MAX_BOUND = 250;

public IDSFitnessFunction(String a_sourceIP, String a_targetIP,
    int a_targetPort, int a_protocol, int a_senderBytes,
    int a_responderBytes) throws UnknownHostException {
    if (a_sourceIP.equals(null) || a_sourceIP.equals("")) {
        throw new IllegalArgumentException(
            "Please provide a Source IPv4address");
    }

    if (a_targetIP.equals(null) || a_targetIP.equals("")) {
        throw new IllegalArgumentException(
            "Please provide a Target IPv4address");
    }

    if (a_targetPort < 0 || a_targetPort > 65535) {
        throw new IllegalArgumentException(
            "Please provide a Target Port Address between 0 and 65535");
    }

    if (a_protocol < 1 || a_protocol > 20) {
        throw new IllegalArgumentException(
            "Please provide a Protocol between 1 and 20");
    }

    if (a_senderBytes < 0) {
        throw new IllegalArgumentException(
            "Please provide the number of Bytes Sent by Originator is greater than 0");
    }

    if (a_responderBytes < 0) {
        throw new IllegalArgumentException(
            "Please provide a number of Bytes Sent by Responder is greater than 0");
    }

    m_sourceIP = getLongFromIPv4AddressString(a_sourceIP);
    m_targetIP = getLongFromIPv4AddressString(a_targetIP);
    m_targetPort = a_targetPort;
    m_protocol = a_protocol;
}

```

```

    m_senderBytes = a_senderBytes;
    m_responderBytes = a_responderBytes;
}

/**
 * Determine the fitness of the given Chromosome instance. The higher the
 * return value, the more fit the instance. This method should always
 * return the same fitness value for two equivalent Chromosome instances.
 *
 * @param a_subject the Chromosome instance to evaluate
 *
 * @return positive double reflecting the fitness rating of the given
 * Chromosome
 * @since 2.0 (until 1.1: return type int)
 * @author Neil Rotstan, Klaus Meffert, John Serri.
 *
 * @author Mark McFadden - Fall 2008
 * @since 2.1
 */
public double evaluate(IChromosome a_subject) {
//    System.out.println(a_subject.toString());

    // Take care of the fitness evaluator. It could either be weighting higher
    // fitness values higher (e.g. DefaultFitnessEvaluator). Or it could weight
    // lower fitness values higher, because the fitness value is seen as a
    // defect rate (e.g. DeltaFitnessEvaluator)

    int x = new Random().nextInt();
    int y = new Random().nextInt();
    boolean defaultComparison = a_subject.getConfiguration().
        getFitnessEvaluator().isFitter(x, y);
//    System.out.println("x: " + x + "; " + "y: " + y);

    double fitness;
    if (defaultComparison) {
        fitness = 0.0d;
    }
    else {
        fitness = MAX_BOUND/2;
    }

/**

```

```

* fitness steps
* @author Mark McFadden
*/
// Step 1: First check the source IP address against a list of known
// black listed IP addresses
// -----
try {
    double sourceIPResult = checkSourceIPv4(m_sourceIP);
    if(sourceIPResult == SOURCE_IP_MAX_BOUND){//is a blacklisted source IP
        fitness += sourceIPResult;
    }else{
        if (defaultComparison) {
            fitness -= environmentalPressure(SOURCE_IP_MAX_BOUND/x, sourceIPResult);
        }else{
            fitness += environmentalPressure(SOURCE_IP_MAX_BOUND/y, sourceIPResult);
        }
    }
} catch (UnknownHostException uhe) {
    uhe.printStackTrace();
}

// Step 2: next check the target IP address against a list of known
// company asset IP addresses
// -----
try {
    double targetIPResult = checkTargetIPv4(m_targetIP);
    if(targetIPResult == TARGET_IP_MAX_BOUND){//is an asset target IP
        fitness += targetIPResult;
    }else{
        if (defaultComparison) {
            fitness -= environmentalPressure(TARGET_IP_MAX_BOUND/x, targetIPResult);
        }else{
            fitness += environmentalPressure(TARGET_IP_MAX_BOUND/y, targetIPResult);
        }
    }
} catch (UnknownHostException uhe) {
    uhe.printStackTrace();
}

// Step 3: then check the port address against a list of known
// attacks by port
// -----

```

```

double targetPortResult = checkPort(m_targetPort);
if(targetPortResult == TARGET_PORT_MAX_BOUND){//is an asset target port IP
    fitness += targetPortResult;
}else{
    if (defaultComparison) {
        fitness -= environmentalPressure(TARGET_PORT_MAX_BOUND/x, targetPortResult);
    }else{
        fitness += environmentalPressure(TARGET_PORT_MAX_BOUND/y, targetPortResult);
    }
}

// Step 4: check the protocol against a list of known
// attacks by protocol
// -----
double protocolResult = checkProtocol(m_protocol);
if(targetPortResult == PROTOCOL_MAX_BOUND){//is a suspicious protocol
    fitness += protocolResult;
}else{
    if (defaultComparison) {
        fitness -= environmentalPressure(PROTOCOL_MAX_BOUND/x, protocolResult);
    }else{
        fitness += environmentalPressure(PROTOCOL_MAX_BOUND/y, protocolResult);
    }
}

// Step 5: check the amount of input bytes is not greater than 250K
// otherwise a potential DOS attack
// -----
double senderByteAmountResult = checkSenderByteAmount(m_senderBytes);
if(targetPortResult == SENDER_BYTES_MAX_BOUND){//is a suspicious byte amount
    fitness += senderByteAmountResult;
}else{
    if (defaultComparison) {
        fitness -= environmentalPressure(SENDER_BYTES_MAX_BOUND/x, senderByteAmountResult);
    }else{
        fitness += environmentalPressure(SENDER_BYTES_MAX_BOUND/y, senderByteAmountResult);
    }
}

// Step 6: check the amount of output bytes is not greater than 1MB
// otherwise a potential illegal query
// -----

```

```

double responderByteAmountResult = checkResponderByteAmount(m_responderBytes);
if(targetPortResult == RESPONDER_BYTES_MAX_BOUND){//is a suspicious byte amount
    fitness += responderByteAmountResult;
}else{
    if (defaultComparison) {
        fitness -= environmentalPressure(RESPONDER_BYTES_MAX_BOUND/x, responderByteAmountResult);
    }else{
        fitness += environmentalPressure(RESPONDER_BYTES_MAX_BOUND/y, responderByteAmountResult);
    }
}

// Finally, make sure fitness value is always positive and then return it.
// -----
// System.out.println(fitness);
return Math.max(1.0d, fitness);
}

/**
 * @author Mark McFadden
 * To assist in the avoidance of local maxima
 * @param a_maxFitness - double of maximum fitness
 * @param generationalFitness
 * @return double
 */
protected double environmentalPressure(double a_maxFitness, double generationalFitness) {
    boolean isFit = new Random().nextBoolean();//artificial environmental pressure
    if (isFit) {
        return a_maxFitness;
    }
    else {
        // we arbitrarily work with half of the maximum fitness as basis for non-
        // optimal solutions (concerning change difference)
        if (generationalFitness * generationalFitness >= a_maxFitness / 2) {
            return 0.0d;
        }
        else {
            return a_maxFitness / 2 - generationalFitness * generationalFitness;
        }
    }
}
}

/**

```

```

* Evaluate Responder Byte Amount
* @author Mark McFadden
* @param bytes
* @return double
*/
private double checkResponderByteAmount(int bytes) {
    //if return amount is > 250KB then illegal query made
    if(bytes > 250000){
        return RESPONDER_BYTES_MAX_BOUND;
    }
    return 10;
}

/**
* Evaluate Sender Byte Amount
* @author Mark McFadden
* @param bytes
* @return double
*/
private double checkSenderByteAmount(int bytes) {
    //if sending amount is > 10K then potential attack
    if(bytes > 10000){
        return SENDER_BYTES_MAX_BOUND;
    }
    return 20;
}

/**
* Evaluate Protocol
* @author Mark McFadden
* @param protocol
* @return double
*/
private double checkProtocol(int m_protocol) {
    //5 is FTP protocol
    if(m_protocol == 5){
        return PROTOCOL_MAX_BOUND;
    }
    return 30;
}

/**

```

```

* Evaluate Port
* @author Mark McFadden
* @param port number
* @return double
*/
private double checkPort(int port) {
    //port 8184 is for internal data access
    if(port == 8184){
        return TARGET_PORT_MAX_BOUND;
    }
    return 40;
}

/**
* Evaluate Target IP
* @author Mark McFadden
* @param long of IPv4 number
* @return double
*/
private double checkTargetIPv4(long m_targetip2) throws UnknownHostException {
    //make sure the source IP addresses if between 125.19.54.125 and 125.19.54.175
    if(m_targetip2 >= getLongFromIPv4AddressString("109.19.54.0") &&
        m_targetip2 <= getLongFromIPv4AddressString("109.19.54.255")){
        return TARGET_IP_MAX_BOUND;
    }
    return 60;
}

/**
* Evaluate Source IP
* @author Mark McFadden
* @param long of IPv4 number
* @return double
*/
private double checkSourceIPv4(long m_sourceip2) throws UnknownHostException {
    //make sure not 109.19.54.0 to 109.19.54.255 which is black listed
    if(m_sourceip2 >= getLongFromIPv4AddressString("125.19.54.125") &&
        m_sourceip2 <= getLongFromIPv4AddressString("125.19.54.175")){
        return SOURCE_IP_MAX_BOUND;
    }
    return 50;
}

```

```

/**
 * Get IPv4 int value
 * @author Mark McFadden
 * @param IPv4 number
 * @return int
 */
public static int getIntIPv4Address(String ipAddress) throws UnknownHostException{
    return getIPv4Address(ipAddress).hashCode();
}

/**
 * Get IPv4 InetAddress
 * @author Mark McFadden
 * @param IPv4 number
 * @return InetAddress
 */
private static InetAddress getIPv4Address(String ipAddress) throws UnknownHostException{
    Pattern pattern = Pattern.compile("(\\d{1,3})\\. (\\d{1,3})\\. (\\d{1,3})\\. (\\d{1,3})");
    Matcher matcher = pattern.matcher(ipAddress);
    byte[] targetIPUpperAddressByteArray = new byte[4] ;
    if(matcher.matches())
    {
        targetIPUpperAddressByteArray[0] = new Integer(matcher.group(1)).byteValue() ;
        targetIPUpperAddressByteArray[1] = new Integer(matcher.group(2)).byteValue() ;
        targetIPUpperAddressByteArray[2] = new Integer(matcher.group(3)).byteValue() ;
        targetIPUpperAddressByteArray[3] = new Integer(matcher.group(4)).byteValue() ;
    }
    else{
        throw new IllegalArgumentException("Input parameter must be an IPv4 Address: \"192.168.1.25\"");
    }

    return InetAddress.getByAddress(targetIPUpperAddressByteArray) ;
}

/**
 * Get JGAP Framework result for int gene
 * @author Mark McFadden
 * @param a_potentialSolution IChromosome
 * @param a_position int position on chromosome
 * @return int value of result
 */

```

```

public static int getSelectedResult(IChromosome a_potentialSolution, int a_position) {
    Integer outcome = (Integer)a_potentialSolution.getGene(a_position).getAllAlleles();
    return outcome.intValue();
}

/**
 * Get JGAP Framework result for double gene
 * @author Mark McFadden
 * @param a_potentialSolution IChromosome
 * @param a_position position on chromosome
 * @return int value of result
 */
public static int getSelectedResultForDouble(IChromosome a_potentialSolution, int a_position) {
    Double outcome = (Double)a_potentialSolution.getGene(a_position).getAllAlleles();
    return outcome.intValue();
}

/**
 * Get IPv4 InetAddress
 * @author Mark McFadden
 * @param intValue of IPv4 address
 * @return InetAddress
 * @throws UnknownHostException
 */
public static InetAddress getInetAddressFromInt(int intValue) throws UnknownHostException{
    return InetAddress.getByAddress(getInetAddressByteArray(intValue));
}

/**
 * @author Mark McFadden
 * @param ipAddress - IPv4 Address String
 * @return long of IPv4 Address
 * @throws UnknownHostException
 */
public static long getLongFromIPv4AddressString(String ipAddress) throws UnknownHostException{
    Pattern pattern = Pattern.compile("(\\d{1,3})\\. (\\d{1,3})\\. (\\d{1,3})\\. (\\d{1,3})");
    long amount;
    Matcher matcher = pattern.matcher(ipAddress);
    if(matcher.matches())
    {
        amount = (long) ((Math.pow(2, 24) * new Integer(matcher.group(1))) +
            (Math.pow(2, 16) * new Integer(matcher.group(2))) +

```

```

        (Math.pow(2, 8) * new Integer(matcher.group(3))) +
        new Integer(matcher.group(4)));
    }
    else{
        throw new IllegalArgumentException("Input parameter must be an IPv4 Address: \"192.168.1.25\"");
    }

    return amount;
}

/**
 * Returns the double value of the IPv4 Address
 * @author Mark McFadden
 * @param ipAddress
 * @return primitive double of IPv4 Address
 * @throws UnknownHostException
 */
public static double getDoubleFromIPv4AddressString(String ipAddress) throws UnknownHostException{
    Pattern pattern = Pattern.compile("(\\d{1,3})\\. (\\d{1,3})\\. (\\d{1,3})\\. (\\d{1,3})");
    double amount;
    Matcher matcher = pattern.matcher(ipAddress);
    if(matcher.matches())
    {
        amount = (Math.pow(2, 24) * new Integer(matcher.group(1))) +
            (Math.pow(2, 16) * new Integer(matcher.group(2))) +
            (Math.pow(2, 8) * new Integer(matcher.group(3))) +
            new Integer(matcher.group(4));
    }
    else{
        throw new IllegalArgumentException("Input parameter must be an IPv4 Address: \"192.168.1.25\"");
    }

    return amount;
}

/**
 * @author Mark McFadden
 * @param intIPv4Address
 * @return byte array of the IPv4 Address
 */
private static byte[] getInetAddressByteArray(int intIPv4Address){
    byte[] ipBytes = new byte[4];

```

```
ipBytes[0] = (byte) ((int)IPv4Address & 0xff000000) >> 24);
ipBytes[1] = (byte) ((int)IPv4Address & 0xff0000) >> 16);
ipBytes[2] = (byte) ((int)IPv4Address & 0xff00) >> 8);
ipBytes[3] = (byte) (int)IPv4Address & 0xff);

return ipBytes;
}
}
```

```
/*-----*/
/*=====APPENDIX C - IPv4Chromosome.java=====*/
/*-----*/
```

```
package IDS;
```

```
/**
```

```
 * @author Mark McFadden
 * @version 1.0
 * IPv4 Chromosome Value Object
 */
```

```
public class IPv4Chromosome {
```

```
    private String sourceIP;
    private String targetIP;
    private int port;
    private int protocol;
    private int senderByteCount;
    private int responderByteCount;
```

```
/**
```

```
 * @return a system log value object that will act as the IDS solution domain chromosome
 * @param source IP off the
 * @param target IP
 * @param port
 * @param protocol
 * @param senderByteCount
 * @param responderByteCount
 */
```

```
public IPv4Chromosome(String sourceIP, String targetIP, int port,
    int protocol, int senderByteCount, int responderByteCount){
```

```
    this.sourceIP = sourceIP;
    this.targetIP = targetIP;
    this.port = port;
    this.protocol = protocol;
    this.senderByteCount = senderByteCount;
    this.responderByteCount = responderByteCount;
```

```
}
```

```
public String getSourceIP() {
```

```
    return sourceIP;
```

```
}
```

```
public void setSourceIP(String sourceIP) {
    this.sourceIP = sourceIP;
}

public String getTargetIP() {
    return targetIP;
}

public void setTargetIP(String targetIP) {
    this.targetIP = targetIP;
}

public void setPort(int port) {
    this.port = port;
}

public int getPort() {
    return port;
}

public void setProtocol(int protocol) {
    this.protocol = protocol;
}

public int getProtocol() {
    return protocol;
}

public void setSenderByteCount(int senderByteCount) {
    this.senderByteCount = senderByteCount;
}

public int getSenderByteCount() {
    return senderByteCount;
}

public void setResponderBytCount(int responderBytCount) {
    this.responderByteCount = responderBytCount;
}

public int getResponderBytCount() {
    return responderByteCount;
}
```

}

}