# Artificial Immune System Mapping for System Security

**V. A. Rajgure[#1], M. S. Ali[#2]**

[#]Department of Computer Science and Engineering,
Prof Ram Meghe College of Engineering and Management, Badnera, Amravati
[1]vinay.rajgure@gmail.com,[2]softalis@gmail.com

**Abstract—** In this paper some of the principles of artificial immune systems are described. A methodology is presented to demonstrate these principles. Work on AIS protection is still relatively new and, no commercial implementations are quite ready for prime time. The proposed approach presented in this paper will not directly enable you to create a realistic network-intrusion system, but there are at least four reasons why this paper worth. First, the proposed method will give you a starting point for hands-on experimentation with a simple AIS system. Second, the principles explained will get over the rather difficult initial hurdle to this area and allow understanding research papers on AIS. Third, several of the programming techniques used in this paper, in particular r-chunks bit matching and negative selection, can be useful in other programming scenarios. And fourth, it can be found that the idea of modeling a software system based on the behavior of the human immune system interesting in its own right.

**Keywords**— Artificial immune system, human immune system, lymphocyte, intrusion detection, system security.

## I. INTRODUCTION

An artificial immune system (AIS) for intrusion detection is a software system that models some parts of the behavior of the human immune system to protect computer networks from viruses and similar cyber attacks. The essential idea is that the human immune system—which is a complex system consisting of lymphocytes (white blood cells), antibodies and many other components—has evolved over time to provide powerful protection against harmful toxins and other pathogens. So, modeling the behavior of the human immune system may provide an effective architecture against cyber attacks.

Computers have become ubiquitous in today's life style. As the use of computers increases, the desire to secure the information stored in computers increases. Computer security is defined as “the protection accorded to an automated information system in order to attain the applicable objectives of preserving the integrity, availability and confidentiality of information system resources (includes hardware, software, firmware, information/data, and telecommunications)". There exist many types of Computer Security. Computer security, in general, can be expressed in three different forms: application security, host security and network security [1].

Application security consists of securing the functionalities of an application. This type of security involves checking that the application does not present vulnerabilities that can be exploited for abnormal operations. Application security is interested in providing integrity, availability and confidentiality to the software and the data in a computer system [2]. The host security's objective is to protect a single host from being attacked by malicious entities. This type of security is concerned with hardware and firmware protection. Host security can be expressed in the form of anti-virus in a personal computer or a server and can also be referred to as a physical security.

Network security consists of guaranteeing that no malicious activities happen in a network of computer systems. This type of security deals with spoofing, denial of service, intrusion detection [3]

and any other network based attacks. Network security, also known as communication security, is also concerned with protecting the transmission of information. It is highly preferable for hosts participating in a network of computers to own a host security. The applications, in the hosts participating in the network, need also to be equipped with an application security.

The field of computer security is interested in providing integrity, availability and confidentiality of information stored in a single computer and of information stored in a system of computers. The majority of the actual computer security systems consist of rule based systems. A rule based system is software that holds a list of rules to be applied when security violations are manifesting. The computer security systems are equipped with pre-constructed rules from either the vendors or the creators. These rules can be updated and expanded via a procedure called security update. Depending on the security systems used [4], the computer users or the computer network administrators may or may not have the possibility to add rules to the security systems.

During a security attack, if the security system has no rules to handle the attack, then the computer user or the network administrator has to shut the computer system(s) down and restart it (them) in a safe mode. Then, the user or the administrator has to find a security update or a security patch from either the security systems vendor or the attacked software creator to repair the problem. If an update does not exist, then the user or the administrator has to spend time and resources investigating the security attack in order to manually remove the problem. This situation slows down the productivity of the organization or the individual that uses the computer or the network of computers. In addition, the security attacks are evolving at a faster rate. [5] One attack can be mutated in different forms and can be equipped with a stealth mode. This situation complicates defending computer system(s) since most of the built-in rule based security systems lack adaptivity to new attacks.

Solving the above situations requires solving the following questions. Is it possible to create a security system that is capable of learning by itself to create new security rules without waiting for security updates? Is it possible to create software that is capable of repairing or patching itself after a security attack happens? Can software and network infrastructures be equipped with the capability to learn by them to create defense procedures from attack experiences? The use of artificial immune systems in solving security problems for computers is an appealing concept for two reasons [6]. Firstly, the human immune system provides the human body with a high level of protection from invading pathogens, in a robust, self-organized and distributed manner. Secondly, current techniques used in computer security cannot cope with the dynamic and increasingly complex nature of operating systems, database management systems, computer networks and their security.

This research is aimed to illustrate how artificial immune systems can be used in computer security [7]. The main goal is to design and develop a security system that is capable of automatically learning to detect, repair and defend computer systems against attacks [8]. The purpose for a framework is for the fact that such a framework shall be generic to provide security at the levels of firmware, operating systems, application systems and communication systems.

The LYSIS computer security, in [9] was among the first researches that used artificial immune system. LYSIS created a set of fixed length binary strings, called detectors that underwent a negative selection, a tolerization and a human co-stimulation. The idea was to imitate a biological immune system. The detectors were trained to distinguish between the network's normal activities, called self, and the abnormal activities, called non-self. During the negative selection and the tolerization period, any detectors that matched a normal behavior, self, of the network were eliminated and replaced by new detectors. The matching was done using the r-contiguous bits, meaning two strings matched each other if they had r contiguous bits in common. To lower the false positives' rate, the detectors that survived the tolerization period alerted a human operator when they matched an intrusion. The human operator either validated or discarded the intrusion alert. If an intrusion was validated, then the detector that triggered the alert received a co-stimulation signal and stayed active in the system. In

the other case, the detector died and was replaced by a new one. A General Framework for Multi-Objective Optimization Immune Algorithms is proposed in [10].

Approaches, similar to the one in [11] used other detectors, called memory detectors, to perform the co-stimulation introduced previously. The approach in [11] also utilized a genetic algorithm to multiply and to propagate highly fit detectors. The fitness of each detector was calculated during the negative selection process. Artificial Immune System Inspired Intrusion Detection System Using Genetic Algorithm is proposed in [12]. A similar approach to [11] is proposed in [13] with a modification that detectors were presented in real values instead of binary. A Euclidean distance instead is employed in [13] instead of r-contiguous bits. The authors implemented two approaches: negative and positive characterizations. The negative characterization approach used negative selection. The positive characterization utilized a sample of self and eliminated non-self detectors. The authors concluded that the positive characterization was more accurate, however it required considerable training time and space. Immune system approaches to intrusion detection are reviewed in [14].

## II.HUMAN IMMUNE SYSTEM

The key elements of a highly simplified human immune system are illustrated in Figure 1. Harmful items are proteins called antigens. In Figure 1 the antigens are colored red and have sharp corners. The human body also contains many non-harmful antigens called self-antigens, or self-items. These are naturally occurring proteins and in Figure 1 are colored green and have rounded sides.

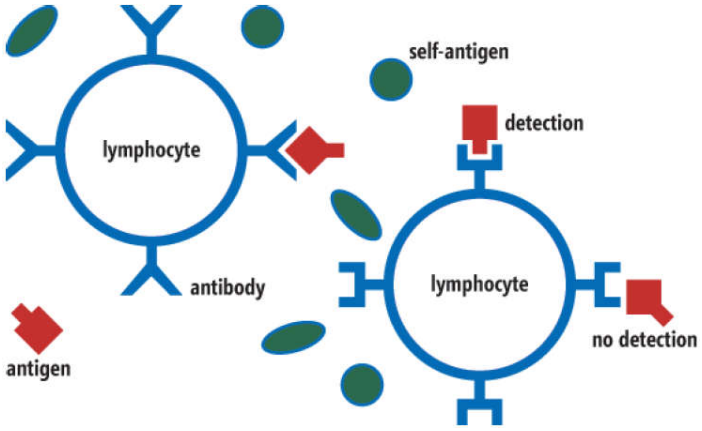


**Figure 1: Key Elements of a Simplified Human Immune System.**

Antigens are detected by lymphocytes. Each lymphocyte has several antibodies, which can be thought of as detectors. Each antibody is specific to a particular antigen. Typically, because antibody-antigen matching is only approximate, a lymphocyte will not trigger a reaction when a single antibody detects a single antigen. Only after several antibodies detect their corresponding antigens will a lymphocyte become stimulated and trigger some sort of defensive reaction.

Notice that no lymphocyte has antibodies that detect a self-item. Real antibodies are generated by the immune system in the thymus, but any antibodies that detect self-items are destroyed before being released into the blood stream, a process called apoptosis. In terms of an intrusion-detection system, antigens correspond to TCP/IP network packets whose contents contain some sort of harmful data, such as a computer virus. Self-antigens correspond to normal, non-harmful network packets. An antibody corresponds to a bit pattern that approximately matches an unknown, potentially harmful network packet. A lymphocyte represents two or more anti bodies/detectors. Apoptosis is modeled using a technique called negative selection.

## III. PROPOSED METHODOLOGY

The proposed methodology begins by creating a set of six normal patterns (patterns that are known not to be part of some cyber attack) that represent TCP/IP network packets in binary form. This is called the self-set in AIS terminology. Of course, in a real AIS system, the self-set would likely contain tens or hundreds of thousands of patterns, and each pattern would be much larger (typically 48-256 bits) than the 12 bits used here. Next, the proposed approach creates three artificial lymphocytes. Each lymphocyte has a simulated antibody that has four bits (again, artificially small), an age and a stimulation field. The antibody field is essentially a detector. The lymphocytes are created so that none of them detect any of the patterns in the self-set. Each lymphocyte has three consecutive 0s or 1s but none of the patterns in the self-set has three consecutive equal bit values.

The Lymphocyte class has four data fields:

```
public BitArray antibody; // Detector
public int[] searchTable; // For fast detection
public int age; // Not used; could determine death
public int stimulation; // Controls triggering
```

All fields are declared with public scope for simplicity. The antibody field is a BitArray. Here BitArray is used, the idea is that using a normal array of int to represent a collection of bits is highly inefficient because each int requires 32 bits of storage. A bit array condenses 32 bits of information into a single int of storage, plus some overhead for the class. Programming languages that don't have a BitArray-like class require doing low-level bit manipulation with bit masking and bit operations. The searchTable field is an array that's used by the Detects method to greatly increase performance. The age field isn't used in proposed approach; many AIS systems track the age of a simulated lymphocyte and probabilistically kill off and generate new lymphocytes based on age. The stimulation field is a counter that tracks how many times a Lymphocyte object has detected a possible threat. In implementation, when a lymphocyte stimulation value exceeds the stimulationThreshold value of 3, an alert is triggered.

The Lymphocyte constructor accepts a single BitArray parameter that represents an antigen/detector. The searchTable array is instantiated using a private helper method named BuildTable.

```
public Lymphocyte(BitArray antibody)
{
        this.antibody = new BitArray(antibody);
        this.searchTable = BuildTable();
        this.age = 0;
        this.stimulation = 0;
}
```

One of the key parts of any AIS system is the routine that determines whether an antigen detects a pattern. Requiring an exact match isn't feasible (and doesn't mimic real antigen behavior). Early work on AIS used a technique called r-contiguous bits matching, in which both an antigen and an input pattern have the same number of bits and detection occurs when antigen and pattern match in r-consecutive bits. Later research indicated that a better detection algorithm is r-chunks bit matching. The r-chunks bit matching is similar to r-contiguous bits matching except that the antigen detector is smaller than the pattern to check, and detection occurs when the antigen matches ny subset of the pattern. For example, if an antigen is 110 and a pattern is 000110111, the antigen detects the pattern starting at index 3. If you think about r-chunks matching for a moment, it can be realized that it's

almost the same as a string substring function. The only difference is that r-chunks matching matches bits and substring matches characters.

A simple approach to r-chunks matching would be to examine the pattern starting at index 0, then at index 1, then 2 and so on. However, this approach is very slow in most situations. There are several sophisticated substring algorithms that preprocess the smaller detector string to create an array (usually called a table). This search table can be used to skip ahead when a mismatch is encountered and greatly improve performance. In situations where the smaller detector string is repeatedly used to check different patterns—as in AIS intrusion detection—the time and memory needed to create the search table is a small price to pay for dramatically improved performance.

The Lymphocyte class Detects method uses the Knuth-Morris-Pratt substring algorithm applied to a BitArray. The Detects method accepts an input pattern such as 000110111 and returns true if the current object's antigen, such as 110, matches the pattern. The code for the Detects method is listed below:

```
public bool Detects(BitArray pattern) // Adapted KMP algorithm
{
        int m = 0;
        int i = 0;
        while (m + i < pattern.Length)
        {
                if (this.antibody[i] == pattern[m + i])
                {
                        if (i == antibody.Length - 1)
                        return true;
                        ++i;
                }
                else
                {
                        m = m + i - this.searchTable[i];
                        if (searchTable[i] > -1)
                        i = searchTable[i];
                        else
                        i = 0;
                }
        }
        return false; // Not found
}
```

The Detects method assumes the existence of the search table. Recall that the Lymphocyte constructor calls a helper method BuildTable to create the searchTable field. The code for BuildTable is listed below:

```
private int[] BuildTable()
{
        int[] result = new int[antibody.Length];
        int pos = 2;
        int cnd = 0;
        result[0] = -1;
        result[1] = 0;
        while (pos < antibody.Length)
        {
```

```
                if (antibody[pos - 1] == antibody[cnd])
                {
                        ++cnd; result[pos] = cnd; ++pos;
                }
                else if (cnd > 0)
                        cnd = result[cnd];
                else
                {
                        result[pos] = 0; ++pos;
                }
        }
        return result;
}
```

The Lymphocyte class defines overridden GetHashCode and ToString merthods. The GetHashCode method is used to prevent duplicate Lymphocyte objects

```
public override int GetHashCode()
{
        int[] singleInt = new int[1];
        antibody.CopyTo(singleInt, 0);
        return singleInt[0];
}
```

This implementation assumes that the BitArray antibody field has 32 bits or less. In realistic situations where the antibody field has more than 32 bits, dealing with duplicate Lymphocyte objects is not so simple. One approach would be to define a custom hash code method that returns a BigInteger type. The Lymphocyte ToString method used in the proposed approach is:

```
public override string ToString()
{
        string s = "antibody = ";
        for (int i = 0; i < antibody.Length; ++i)
                s += (antibody[i] == true) ? "1 " : "0 ";
        s += " age = " + age;
        s += " stimulation = " + stimulation;
        return s;
}
```

Standard (non-AIS) computer virus-detection software such as Microsoft Security Essentials work by storing a local database of known computer virus definitions. When a known virus pattern is detected, an immediate alert is triggered. Such antivirus systems have trouble dealing with variations on existing viruses, and fail entirely in most situations when faced with a completely new virus. AIS intrusion detection works in the opposite way by maintaining a set of input patterns that are known to be non-harmful and then triggering an alert when an unknown pattern is detected. This allows AIS intrusion-detection systems—in principle, at least—to detect new viruses. However, dealing with false positives—that is, triggering an alert on a non-harmful input pattern—is a major challenge for AIS systems. A real AIS intrusion-detection system would collect many thousands of normal input patterns over the course of days or weeks. These normal self-set patterns would be stored either on a local host or a server. Also, a real AIS system would continuously update the self-set (and the induced set of lymphocytes) of normal patterns to account for normal changes in network traffic over time. The proposed method in this paper creates an artificial, static self-set using method LoadSelfSet:

```
public static List<BitArray> LoadSelfSet(string dataSource)
{
```

```
        List<BitArray> result = new List<BitArray>();
        bool[] self0 = new bool[] { true, false, false, true, false, true,
        true, false, true, false, false, true };
        // Etc.
        bool[] self5 = new bool[] { false, false, true, false, true, false,
        true, false, false, true, false, false };
        result.Add(new BitArray(self0));
        // Etc.
        result.Add(new BitArray(self5));
        return result;
    }
```

The method accepts a dummy not-used dataSource parameter to suggest that in a realistic scenario the self-set data would not be hardcoded. The BitArray constructor, somewhat surprisingly, accepts an array of bool values where true represents a 1 bit and false represents a 0 bit. The self-set data is generated in such a way that no self-item has more than two consecutive 0s or 1s.

The proposed approach uses utility method ShowSelfSet, which calls helper method BitArrayAsString, to display the self-set:

```
    public static void ShowSelfSet(List<BitArray> selfSet)
    {
        for (int i = 0; i < selfSet.Count; ++i)
        Console.WriteLine(i + ": " + BitArrayAsString(selfSet[i]));
    }
    public static string BitArrayAsString(BitArray ba)
    {
        string s = "";
        for (int i = 0; i < ba.Length; ++i)
            s += (ba[i] == true) ? "1 " : "0 ";
        return s;
    }
```

After referring how the human immune system works, it is identified that the lymphocyte set should contain only Lymphocyte objects that do not detect any patterns in the selfset. Method CreateLymphocyteSet is listed below:

```
    public static List<Lymphocyte> CreateLymphocyteSet(List<BitArray> selfSet,
    int numAntibodyBits, int numLymphocytes)
    {
        List<Lymphocyte> result = new List<Lymphocyte>();
        Dictionary<int, bool> contents = new Dictionary<int, bool>();
        while (result.Count < numLymphocytes)
        {
            BitArray antibody = RandomBitArray(numAntibodyBits);
            Lymphocyte lymphocyte = new Lymphocyte(antibody);
            int hash = lymphocyte.GetHashCode();
            if (DetectsAny(selfSet, lymphocyte) == false &&
                contents.ContainsKey(hash) == false)
            {
                result.Add(lymphocyte);
                contents.Add(hash, true);
            }
        }
```

```
        return result;
    }
```

In AIS terminology, method CreateLymphocyteSet uses negative selection. A random Lymphocyte object is generated and then tested to see if it does not detect any patterns in the self-set, and also that the lymphocyte is not already in the result set. This approach is rather crude, and there are other approaches that are more efficient. A Dictionary collection is used with a dummy bool value to track existing Lymphocyte objects; the HashSet in the .NET Framework 4.5 and later is a more efficient alternative. A random Lymphocyte object is created by generating a random BitArray:

```
    public static BitArray RandomBitArray(int numBits)
    {
        bool[] bools = new bool[numBits];
        for (int i = 0; i < numBits; ++i)
        {
            int b = random.Next(0, 2); // between [0,1] inclusive
            bools[i] = (b == 0) ? false : true;
        }
        return new BitArray(bools);
    }
```

Helper method DetectsAny accepts a self-set, and a lymphocyte scans through a self-set and returns true if any pattern in the self-set is detected by the antigen in the lymphocyte:

```
    private static bool DetectsAny(List<BitArray> selfSet,Lymphocyte lymphocyte)
    {
        for (int i = 0; i < selfSet.Count; ++i)
            if (lymphocyte.Detects(selfSet[i]) == true) return true;
        return false;
    }
```

The proposed approach uses a utility method ShowLymphocyteSet to display the generated Lymphocyte objects:

```
    public static void ShowLymphocyteSet(List<Lymphocyte> lymphocyteySet)
    {
        for (int i = 0; i < lymphocyteySet.Count; ++i)
            Console.WriteLine(i + ": " + lymphocyteySet[i].ToString());
    }
```

## IV. IMPLEMENTATION DETAILS AND RESULTS

The proposed approach is implemented as a single C# console application named Artificial Immune Systsem basedn Intrusion Detection (AISID). Visual Studio 2013 ultimate is used for the implementation, but any version of Visual Studio that has the Microsoft .NET Framework 3.0 or later should work. The overall program structure is listed below:

```
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Collections; // for BitArray
    using System.Threading.Tasks;
    namespace AISID
    {
```

```
  class AISID
  {
static Random random;

static void Main(string[] args)
{
 Console.WriteLine("\n Artificial Immune System based Intrusion Detection\n");
 random = new Random(1);
 int numPatternBits = 12;
 int numAntibodyBits = 4;
 int numLymphocytes = 3;
 int stimulationThreshold = 3;
 Console.WriteLine("Loading self-antigen set ('normal' historical patterns)");
 List<BitArray> selfSet = LoadSelfSet(null);
 ShowSelfSet(selfSet);
 Console.WriteLine("\nCreating lymphocyte set using negative selection" +
     " and r-chunks detection");
    List <Lymphocyte> lymphocyteSet = CreateLymphocyteSet(selfSet, numAntibodyBits,
    numLymphocytes);
ShowLymphocyteSet(lymphocyteSet);
Console.WriteLine("\nBegin AISID simulation\n");
int time = 0;
int maxTime = 6;
while (time < maxTime)
{
 Console.WriteLine("================================================");
 //Console.WriteLine("time = "+ time);
 BitArray incoming = RandomBitArray(numPatternBits);
 Console.WriteLine("Incoming pattern = " + BitArrayAsString(incoming) + "\n");
 for (int i = 0; i < lymphocyteSet.Count; ++i)
 {
  if (lymphocyteSet[i].Detects(incoming) == true)
  {
   Console.WriteLine("Incoming pattern detected by lymphocyte " + i);
   ++lymphocyteSet[i].stimulation;
   if (lymphocyteSet[i].stimulation >= stimulationThreshold)
   Console.WriteLine("Lymphocyte " + i + " stimulated!" +
                 " Check incoming as possible intrusion!");
   else
   Console.WriteLine("Lymphocyte " + i + " not over stimulation threshold");
  }
  else
   Console.WriteLine("Incoming pattern not detected by lymphocyte " + i);
 }
 ++time;
 Console.WriteLine("================================================");
}  // while
Console.WriteLine("\nEnd AISID\n");
Console.ReadLine();
```

```
    } // Main
} // class AISID
```

Figure 2 shows the structure of the project under solution explorer. The result of the implemented system is shown in figure 3.

After the system has been initialized, the proposed approach begins a tiny simulation with six input patterns. The first input is detected by lymphocyte 1, but because each lymphocyte has an activation threshold, the lymphocyte doesn't trigger an alarm. At time t = 3, lymphocyte 1 detects another suspicious input but, again, is not yet over the threshold. But at time t = 5, lymphocyte 1 detects a third suspicious input packet and a simulated alert is triggered.

## V. CONCLUSIONS

The proposed methodology and it's implementation details presented in this paper should give researchers a solid basis for hands-on experimentation with an AIS. Researchers have suggested many options. For example, the proposed methodology in this paper fires an alert when a single lymphocyte detects unknown input patterns more than some threshold number of times. The idea here is that real pathogens emit many antigens.

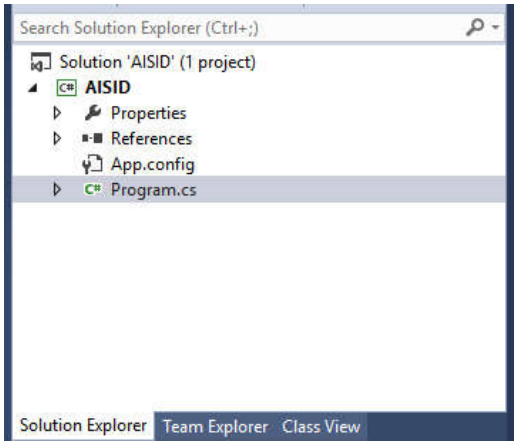


**Figure 2: Implemented System Structure.**

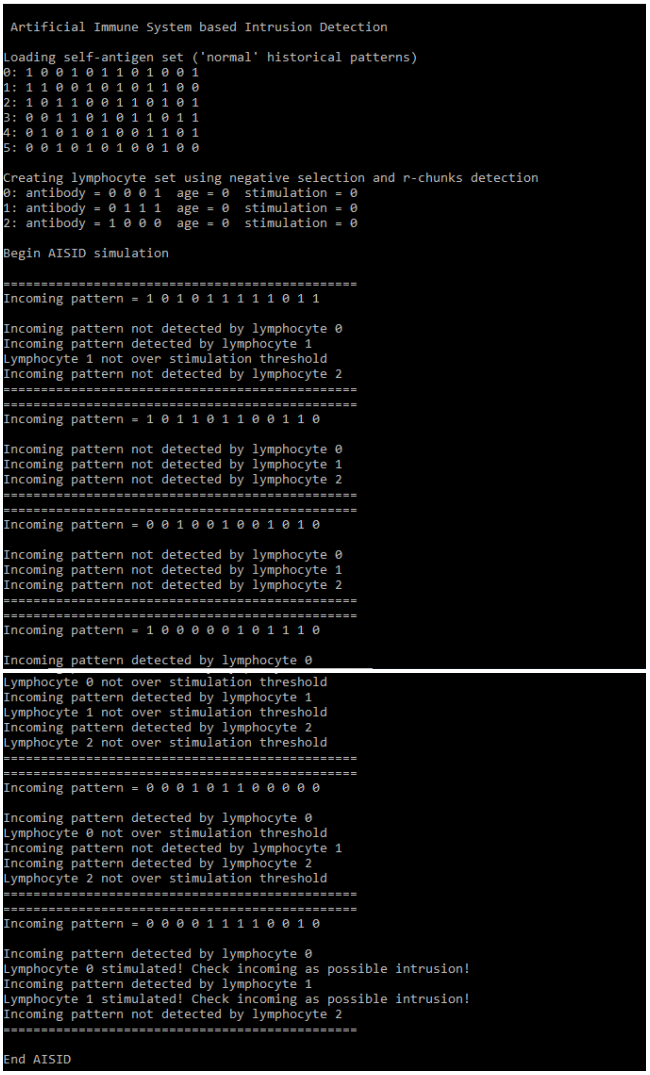


**Figure 3: Results of the proposed implemented methodology for intrusion detection using AIS.**

Another possibility is for the AIS system to trigger an alert only after multiple different lymphocytes detect the same unknown pattern. It's important to point out that AIS is not intended to be a single solution for intrusion detection. Rather, it's meant to be part of a multilayer defense that includes traditional antivirus software. Additionally, because work with an AIS is still relatively young, there are many unanswered questions.

## REFERENCES :

[1] AIS Web (2012), http://www.artificial-immune-systems.org/algorithms.shtml, Basic Immune Inspired Algorithms, (Last Modified: 25th November 2012).

[2] Amira Sayed A. Aziz, Mostafa A. Salama, Aboul ella Hassanien and Sanaa El-Ola Hanafi , "Artificial Immune System Inspired Intrusion Detection System Using Genetic Algorithm", Informatica 36 (2012) 347–357

[3] Chen Yunfang , "A General Framework for Multi-Objective Optimization Immune Algorithms", I.J. Intelligent Systems and Applications, 2012, 6, 1-13 , Published Online June 2012 in MECS (http://www.mecs-press.org/) DOI: 10.5815/ijisa.2012.06.01

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

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

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

[7] Fernando Esponda, Stephanie Forrest, and Paul Helman. "A formal framework for positive and negative detection". IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, 34(1):357–373, 2004.

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

[9] Hofmeyr, S. A. and S. A. Forrest (2000). Architecture for an artificial immune system. IEEE Transactions on Evolutionary Computation 8, 443-473.

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

[11] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous system call detection. ACM Transactions on Information and System Security, 9(1):61–93, 2006.

[12] Oppliger, R . Security technologies for the World Wide Web. Norwood, MA,USA: Artech House, Inc.

[13] Solofoarisina Arisoa Randrianasolo, M.S., "Articial Intelligence in Computer Security: Detection, Temporary Repair and Defense" Ph.D. dissertation, Dept. Comput. Sci., Texas Tech University, May, 2012.

[14] Marzie Tabatabaefar, Maryam Miriestahbanati, Jean-Charles Gregoire, "Network Intrusion Detection through Artificial Immune System," in Proceedings of 2017 Annual IEEE International Systems Conference, May 2017.