DETECTING MALICIOUS CODE IN SENSOR NETWORK APPLICATIONS USING PETRI NETS

By

PRATHIBA REDDY NALABOLU

Bachelor of Science in Computer Science and Engineering

Osmania University

Hyderabad, Andhra Pradesh

2005

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2007

DETECTING MALICIOUS CODE IN SENSOR

NETWORK APPLICATIONS USING PETRI NETS

Thesis Approved:

Dr. Johnson P. Thomas

Thesis Adviser

Dr. Nohpill Park

Dr. Venkatesh Sarangan

Dr. A. Gordon Emslie

Dean of the Graduate College

ACKNOWLEDGEMNETS

I sincerely thank my advisor Dr. Johnson P. Thomas for his continuous guidance, support and for many insightful conversations during the developement of ideas in this thesis. I would like to thank my committee members Dr. Venkatesh Sarangan and Dr. Nohpill Park for their intuitive comments.

Finally, I would also like to thank Computer Science Department for providing me assistance in the technical areas during my education here.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. REVIEW OF LITERATURE..3

III. PETRI NETS 7

3.1 Modeling Software..9

IV. DELUGE 11

4.1 Working of Deluge 12

V. SYSTEM MODEL...15

VI. ATTACKS AND APPLICATION MODELING...19

6.1 Important Assumptions..20
6.2 Attacks 21
6.3 Application...22

VII. IMPLEMENTATION 23

7.1 Attack Modeling 24
7.1.1 Selective Forwarding and Sink Hole 24
7.1.2 Sybil Attack 28
7.1.3 Denial of Service...31
7.2 Application Modeling 35
7.3 Application with Attacks 38
7.3.1 Selective Forwarding and Sink Hole 38
7.3.2 Sybil Attack 40

Chapter Page

7.3.3 Denial of Service......42
7.4 Detection Tool 43
7.4.1 Code to Petri net transformation......43
7.4.2 Detection Algorithm 48
7.4.3 Validation of Algorithm......50
7.4.4 Complexity of the of Algorithm 62

VIII. CONCLUSIONS......64

APPENDIX......66

REFERENCES 76

LIST OF TABLES

Table Page

1 Places to transitions....47

2 Transitions to Places 48

3 Category of places and transitions 48

4 Places to Transitions, sfPT....51

5 Transitions to places sfTP 51

6 Category of places and transitions, sfC....52

7 Places to transitions, appPT 52

8 Transitions to places appTP 53

9 Category table appE....53

10 Places to transitions appPT 57

11 Transitions to places appTP 58

12 Category Table appE....59

13 Places to transitions appPT 60

14 Transitions to places appTP 60

15 Category Table appE....61

LIST OF FIGURES

Figure Page

1. Petri net representing the flow of control in programs [14]................................10

2. Nodes advertise new code version..13

3. Nodes request for new code version..13

4. Nodes advertise new code version for next hop..14

5. Malicious Code Detector...15

6. Selective Forwarding..26

7. Selective Forwarding and Sink Hole..27

8. Code for Sybil Attack..29

9. Sybil Attack..30

10. Denial of Service...33

11. Petri net for Denial of Service..34

12. Application...36

13. Petri net for application...37

14. Application with Selective Forwarding..39

15. Application with Sybil Attack..40

16. Application with Denial of Service..42

17. IF Construct..44

18. FOR Construct...45

Figure Page

19. WHILE Construct 45

20. Event Send 46

21. Example to illustrate data structures 47

CHAPTER I

INTRODUCTION

The need to reprogramme sensor nodes in a wireless sensor network arises due to changing application requirements. Sensor nodes once deployed cannot be physically reached. Therefore techniques have been developed to reprogramme the motes remotely. But the process of code updation is fraught with challenges. An adversary can inject packets into the network and accomplish widespread rapid installation of corrupt code. Attackers can also hijack packets and embed malicious code within the packet. This code can launch different attacks like Selective forwarding, Denial of Service and Sinkhole attacks.

Code updation mechanisms like Remote code propagation [2], Efficient Code Distribution [1] and Viral Code Propagation [25] concentrate on reducing the bandwidth and energy usage of the node. They do not stress on propagating the code securely. Secure code propagation in Sensor Networks is important as any attacker could inject malicious code and propagate it. Currently there are no existing approaches in wireless sensor networks to identify whether the code received by a mote is free of malicious content.

The important issue is how to deal with the malicious code that has already been received by the sensor node. Malicious code modifies the behavior of the sensor node and attacks the applications as well. In this thesis we propose to identify potential attacks in the code which is received by a node during code distribution process. The key idea is to build a library of attacks, model the attacks and the sensor network application, in Petri nets. We then aim to identify malicious patterns in the application (code) and also estimate the damage caused by the attacks.

The objectives of this thesis include:

- Implement Selective Forwarding, Sink Hole attack, Denial of Service and Sybil attack.
- Define Petri net models for list of attacks mentioned above.
- Embed Selective Forwarding, Sink Hole attack, Denial of Service and Sybil attack in a sensor network application.
- Define Petri net models for the code which has been embedded with attacks
- Identify attack signatures in the sensor network application

The remainder of this document contains chapters as follows. Chapter II contains Problem background, related work and literature review pertinent to it. Chapter III reviews about Petri nets and modeling software using Petri nets. Chapter IV discusses the relevance of Deluge protocol. The System Model is discussed in Chapter V. Chapter VI talks about modeling attacks and application. Finally Chapter VII describes implementation of the solution.

CHAPTER II

REVIEW OF LITERATURE

In this section we review the previous work in the field. We first review why we need to reprogramme the sensor networks. Sensor nodes are deployed in order to gather valuable data. Once they are deployed, they need not be tended to. These nodes are usually deployed in hostile or sensitive environments like ocean beds, birds nest, or in buildings [24]. However in the course of time, the application requirements may change or a bug fix may be required. In this situation, the node needs to be reprogrammed. The problem of reprogramming the wireless sensor networks has been addressed in many ways. Some of the techniques are Remote Code Updation Mechanism [2] and Efficient Code Distribution [1]. The two mechanisms are not secure. In other words there is no way a sensor node can detect if the code received has malicious content.

To update the code in sensor networks, the code is divided into images or fragments. These fragments are transmitted part by part. This mechanism is called pipelining [1]. Due to this mechanism the nodes need not wait for the entire code image. After receiving a part or fragment they can forward it to other nodes. A malicious node can intercept the code fragment as it is being transmitted.

It can hijack the original code and send a malicious code or viruses. The malicious node can cause widespread installation of malicious code. It will also consume the network resources and bandwidth.

The code distribution mechanism proposed by Stathopoulos et. al. is 'Remote Code Update Mechanism for Wireless Sensor Networks' [2]. The protocol developed is called Multihop Over-the-Air Programming (MOAP). This mechanism proposes to use Ripple dissemination protocol for reprogramming the sensor nodes remotely. The protocol uses a publish-subscribe method. Here a set of nodes act as source, other nodes act as receivers. For a node to become a source it should have the complete code image. So a node waits for all the code image packets to arrive. When that node becomes the source, the new version of the code is advertised and other nodes subscribe a newer version of this code image.

The Ripple mechanism guarantees that the source is one hop away, follows ripple like data propagation [2]. The MOAP protocol is also vulnerable to attacks from malicious nodes or advisory [11]. An adversary can compromise a node in the network. This node can pose as a source and publish its malicious code. When the code is installed and executed, different attacks like DOS, Sybil attack could be launched on the node.

Efficient Code Distribution in Wireless Sensor Networks [1] proposed by Reijers et. al. is not resilient to malicious code injections either. It considers packet losses, communication costs but it does not consider the secure code propagation. In this scheme the new code image is built using edit script of commands that are easy to process by the nodes. The procedure for code distribution consists of four stages- Initialization, Code

image building, Verification and Loading. Energy is saved by distributed only changes to the currently running code [1].

‘Viral code Propagation in Wireless Sensor Networks’ by Levis [25] presents scalable and rapid algorithms for disseminating code through a sensor network. They show that by dynamically adjusting transmission rates, networks can reprogramme very quickly while having a low overhead when stable. In order to reduce the time taken for reprogramming and to prevent the saturation of available bandwidth, three distributed algorithms are proposed.

Slijepcevic et. al. [24] proposed a security mechanism to prevent malicious code injection into the network. They classified the types of data existing in sensor networks, and identified possible communication security threats according to that classification. They developed a multitiered security architecture where each mechanism has different resource requirements, they allow for efficient resource management, which is essential for wireless sensor networks. The security architecture called SensorWare is multitiered where each tier is based on private key cryptography. Each tier in the multitiered architecture is implemented with by using various algorithms or by using the same algorithm with adjustable parameters that change its strength and corresponding computational overhead. Using one algorithm with adjustable parameters has the advantage of occupying less memory space [24]. They characterized mobile code as sensitive data. They employed encryption to messages with code.

However, the attacker can break the encryption using ‘brute force’ approach and inject harmful code. Using encryption also results in overhead. They also did not state

how to protect the node from malicious code already at the node. This has been stated as one of their future challenges.

Some of the cases where a malicious code can hamper the working of the node and in turn the entire network are:

- The malicious code can launch DOS (Denial of Service) and Selective Forwarding attacks in the network.
- It can consume the battery life and other scarce resources on the nodes.
- In military applications it is very important to verify code updates to prevent downloading of malicious code or viruses.
- In commercial application like manufacturing, if a sensor node picks up malicious code, it could affect the profit making processes.
- In applications that require privacy to be maintained, malicious nodes try to propagate code to sensor nodes to snoop on the information.

Very little work has been done on the secure code updation in sensor networks. It is evident that all the previous work has been aimed to prevent malicious injections into the network. As far as we are aware, no one has looked at the problem of detecting malicious code present at the sensor node.

CHAPTER III

PETRI NETS

In this section we review Petri nets and their role in understanding systems. Petri nets are flowcharting technique used to model asynchronous and concurrent processes [14]. Petri nets are composed of four symbols: circles, bars, arcs and dots. A circle represents a place which models a condition in the graph. The bars are transitions in which represent actions that occur. The arcs are bidirectional connections between places and transitions. A place can only connect to a transition. A transition in turn can only connect to a place. Therefore a Petri net is a bipartite directed graph. The dots are tokens which reside in places. Tokens move from place to place upon occurrence of some rules [14].

A transition can fire when it becomes enabled. A transition is enabled when each of the input places has at least one token. When a transition fires the token from each input place goes to each output place. If a place is connected to two or more transitions then firing of one transition disables all other transitions. Another arc known as inhibitor arc connects a place to a transition and is represented with a small circle at the end of the arc. Inhibitor arc is used to model limited resources.

The definition of Petri net follows [16]:

Petri net is a 5 tuple (P, T, F, W, M_0) where

- P is a finite set of places;
- T is a finite set of transitions;
- F is a set of arcs known as flow relations;
- W is a weight function;
- $M_{0:}$ P -> {0, 1, 2….} is the initial marking

A Petri net with tokens is said to be a marked Petri net. Petri net has characteristics such as boundness and livness. A Petri net can have more than one token in a place.

Boundness: A k- bounded Petri net has k tokens in a single place at a time. If the exact value of 'k' is unknown but is known to be some finite number then the net is referred to as being 'bounded'.

Safe net: In a case where the number of tokens in a net is equal to one, it is called a 'safe net'.

Conservative net: The total number of tokens has to remain constant.

Transitions can be in any of the three states:

Dead: A transition is dead if there exists no sequence of firings, from some initial markings, which will enable the transition.

Potentially firable: If there exists a sequence of firings which enable a transition, then it is potentially firable.

Live: A transition is live, if for every possible marking that can result from some initial markings, the transition can be enabled.

3.1 Modeling Software

Petri nets can give the user graphical presentation of the code he wants to model [14]. Program structures such as IF- THEN- ELSE, DO-WHILE and PARBEGIN and PAREND can be modeled in Petri nets. Although Petri nets do not shown which path will be chosen during execution time, however they do show the structure of the code. Figure 1 shows the modeling of the following code.

```
L     S0
      DO WHILE P1
      IF P2 THEN
        S1
        ELSE
        S2
      END IF
      PARBEGIN S3, S4, S5
     PAREND
      END DO
     GOTO L
```

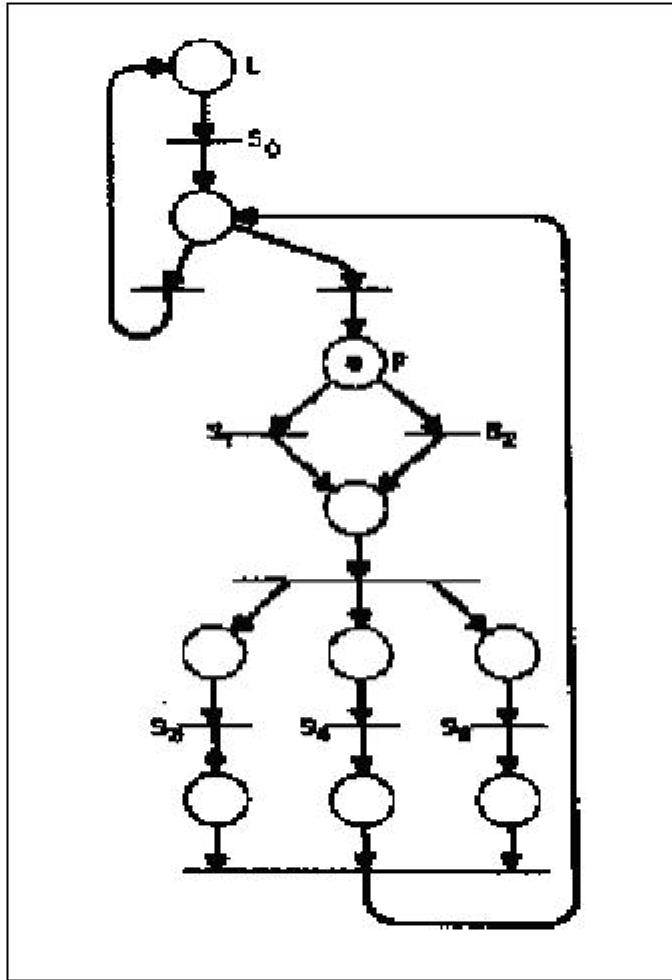


Figure 1 Example of Petri net representing the flow of control in programs [14]

CHAPTER IV

DELUGE

Deluge provides an efficient mechanism for remotely installing any code such as program binaries, to many nodes within a wireless sensor network. The following features are included with Deluge [26]:

Multihop support: Wirelessly program all nodes in a multihop network without physically handling the nodes.

Epidemic propagation: Continuous propagation by all nodes helps ensure reachability of those nodes with intermittent connectivity.

Store multiple program images: Each node can store multiple program images and can quickly switch your network between different programs without continuous downloading.

Golden image: A program image with minimal support for network programming stored in a safe location on external flash. This piece of code will allow for recovery.

Isolated bootloader: A piece of code that is guaranteed to execute after each reset independent of the TinyOS application. The bootloader is responsible for programming the microcontroller and recovers from programming errors by loading the Golden Image.

Deluge's multiple program image support allows different application code to coexist in the network [26]. With some additional logic to determine which program image a specific node should use, a heterogeneous network with different application code can exist. Finally, Deluge exports a very simple interface to extend its functionality. For example, Deluge can be enabled or disabled to control which nodes participate in the dissemination process. Additionally, nodes can decide which program image to use and when, thus allowing for heterogeneous networks where nodes execute different binaries [26].

With the help of deluge, we can remotely install the new program on the nodes. The program will be stored in the flash memory. Since deluge can hold up to three program binaries, different applications coexist. The program we write to detect malicious code can access the sensor application in the flash memory.

4.1 Working of Deluge

This section captures the gist of how deluge epidemically propagates new applications on motes [27].

Step1: Nodes periodically advertise their new version of the application. In the figure below, a node advertises ‘Version 2’. Its immediate neighbors realize that they have ‘Version 1’.

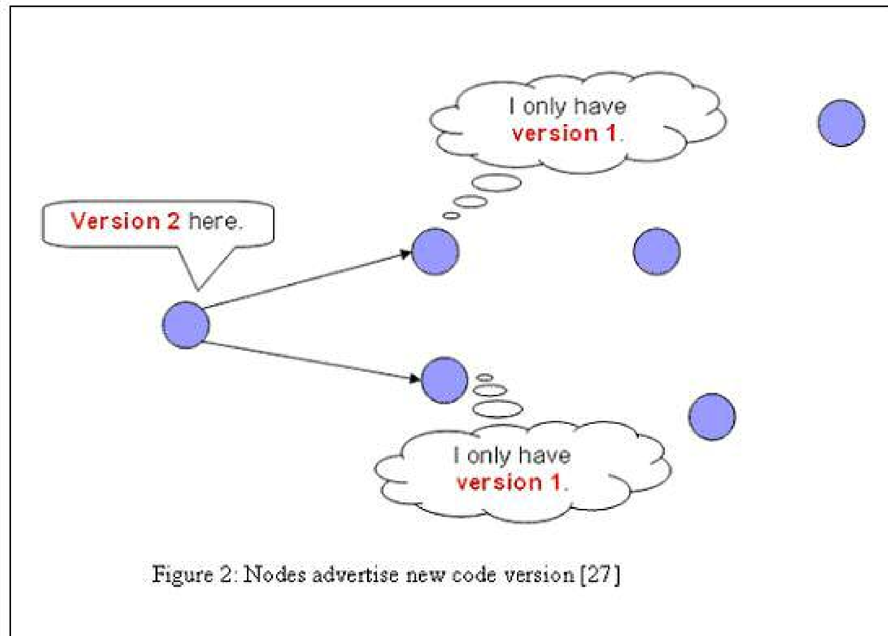


Figure 2 Nodes advertise new code version

Step 2: The neighbors request for the new versions.

Step 3: Requested data is sent.

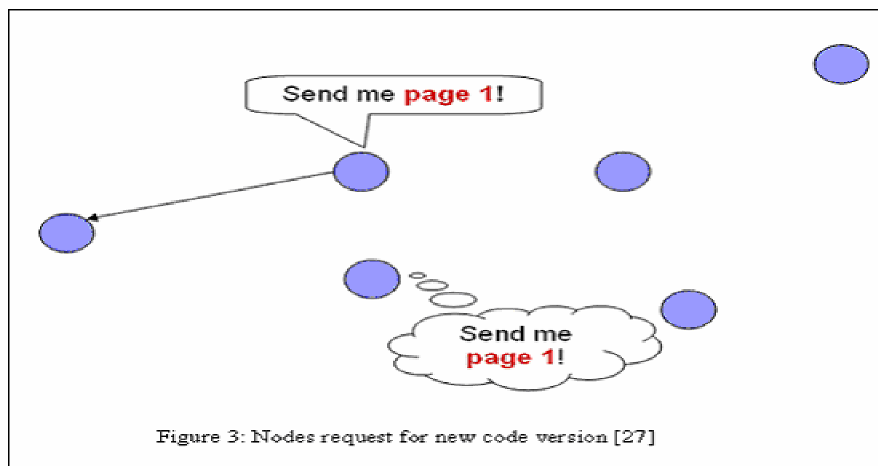


Figure 3 Nodes request for new code version

Step 4: Nodes use Nack to indicate dropped packets. Dropped packets are sent again.

Step 5: Now the neighbors of first node, advertise their new versions to next hop. These nodes in the next hop receive new versions.

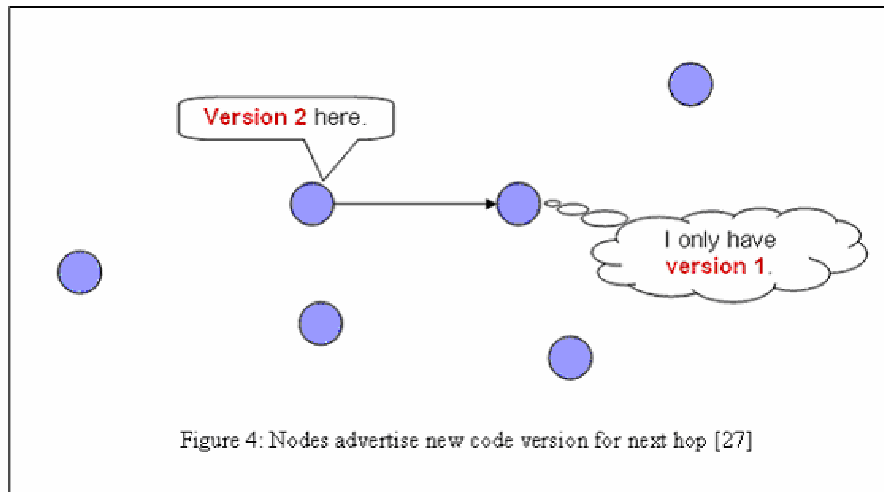


Figure 4 Nodes advertise new code version for next hop

Continuous propagation is exhibited by all the nodes, thereby reaching nodes with intermittent connectivity.

CHAPTER V

SYSTEM MODEL

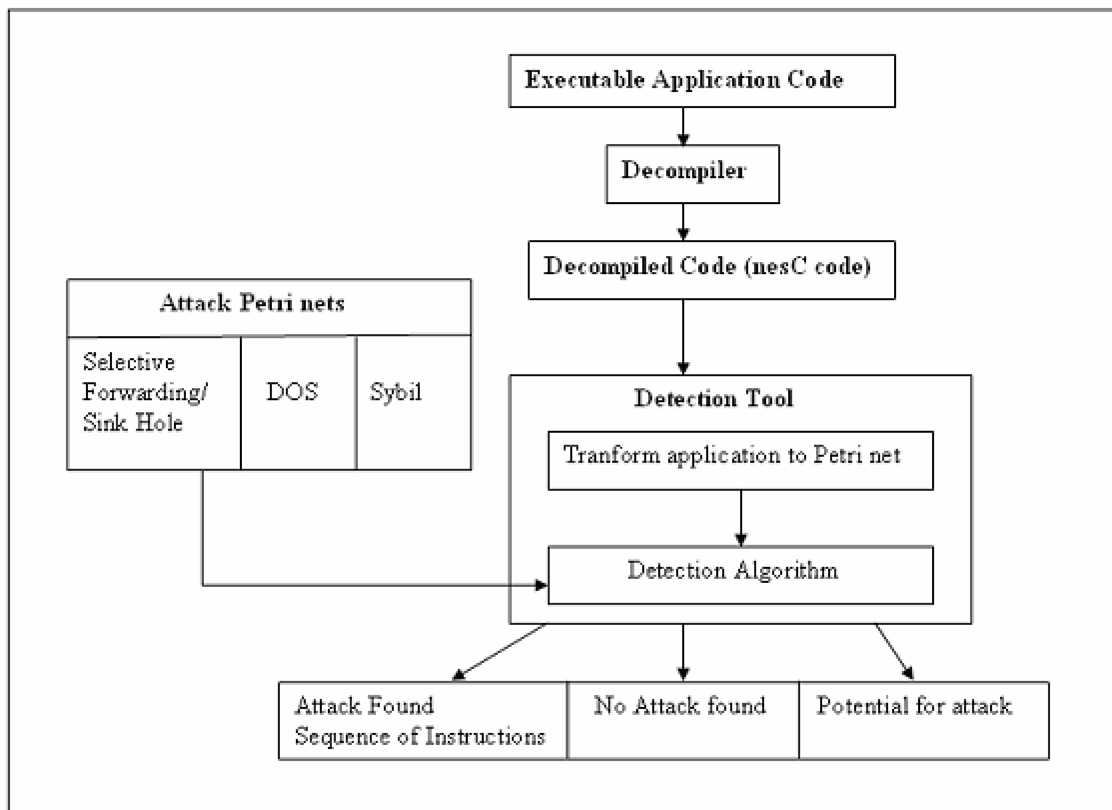


Figure 5: Malicious Code Detector

In this thesis we identify potential attacks in the code which is received by a node during code distribution process. It aims to create a library of attacks by representing

selective forwarding, DOS attacks, Sybil attack and Sink Hole attack in the form of Petri nets. This work is based on 'Static Analysis of Executables to Detect Malicious Patterns' [23]. The problem considered in this thesis is as follows: Assume we are given source code of a sensor network application which could possibly contain an attack like selective forwarding. We have to determine if the code contains any malicious content that would change the behavior of a node.

System Model Description

Figure 5 shows the system model. The main components in the system model are:

- Decompiler
- Attack Petri nets
- Detection tool
    - Transform application to attack Petri nets
    - Detection Algorithm

The function of each component is as follows:

Decompiler: It takes the executable nesC code and decompiles it to nesC source code.

Attack Petri nets: Library of attack Petri nets which are selective forwarding, denial of service, Sybil and sink hole attack.

Detection Tool: The detection tool has two functions:

- It transforms the nesC code to Petri net.
- It takes attack Petri nets and the Petri net of application, matches both of them. It uses detection algorithm that is implemented in Java. It matches both the Petri nets to detect an attack. Finally it outputs whether an attack is present or not.

Detection Algorithm uses two violation categories to decide if the application has malicious content. The violation categories are:

Existence: The fact that something exists is a violation.

Sequence: The fact that several things happen in a sequence is sufficient to detect an attack.

'Existence' indicates the potential for an attack, whereas 'Sequence' shows the presence of an attack.

Input/Output to the System

Input: The input to the system is executable code of a sensor node application. A decompiler is used to decompile executable code to nesC code. This code is given as

input to the Detection Tool. Detection algorithm takes the nesC code and attack Petri nets as inputs.

Output: If the attack is found in the application it returns the sequence of instruction representing the attack. If the attack is not found then it says the attack is not found. If some of the attack states are found but there is no complete attack signature then it says that there could be a potential for an attack.

CHAPTER VI

ATTACKS AND APPLICATION MODELING

We are going to represent sensor network attacks and application in the form of Petri nets. The main motivation to use Petri nets in representing the attacks is the possibility to understand important properties of the attacks and reason about them. They help in examining the techniques used or targeted by particular attacks [20]. One of the techniques to insert malicious code is to insert unnecessary dead code or unwanted jump instructions. In this section we discuss what places and transitions in a Petri net represent while modeling attacks and applications.

As we said before the attacks and application will be in nesC code. In nesC there are events which give rise to actions in the sensor node. For example the event ‘Send’ causes an action where a packet is to be sent to another node. The event ‘Receive’ means that a packet has been received by a node.

Attacks like selective forwarding, sink hole and Sybil attack are routing attacks that mainly take place during routing packets. For this reason, we are going to examine the events like ‘Send’ and ‘Receive’. During these events we are going to examine the actions that are taking place during forwarding a packet and after receiving it.

The work is divided as

- **Attack Modeling**
    - Programming attacks in nesC using TOSSIM simulator.
    - Modeling attacks as Petri nets creating a library of attack Petri nets.

- **Application Modeling**
    - Programming the sensor application in nesC.
    - Embedding attacks in the application.
    - Modeling the above application as Petri net.

6.1 Important Assumptions

The following are assumptions made in this thesis:

- We only consider code which is infected with attacks such as:
    - Selective forwarding
    - Denial of service
    - Sink Hole
    - Sybil Attack
- The malicious code is embedded with sensor network application.
- The source code of the sensor application in nesC.

6.2 Attacks

The attacks are routing attacks that take place during forwarding packets. The attacks target packets dropping them completely/selectively and also flooding them repeatedly. Therefore, we represent the places, transitions, and tokens as follows.

Places: Sensor nodes and programming constructs in the nesC code.

Transitions: Events, actions and flow of control (branches in the code).

Tokens: Packets that being sent and received.

We have a Petri net P = {p_0, p_1, $p_{2...}$pn} of places representing interesting states or modes of the security relevant entities of the system in interest. Then we have a set T = {t_0, t_1, $t_{2...}$ tn) of transitions that represent input events, commands, or data that can cause one or more security relevant entities to change their state. This Petri net also has a set of tokens that move from place to place when transitions are fired. If token is at place it means that the attacker has gained control of that place. If pi and pj are two places where pi precedes pj, then the attacker should gain control of pi before gaining control of place pj [20].

6.3 Application

Application has embedded attack in it. We are going to examine send and receive events, where a node is involved in sending a packet and another node is involved in receiving the packet. Hence nodes are sending, receiving and forwarding packets. These events are causing some actions to take place. Transitions represent events and flow of control of code, which will be discussed in detail in chapter VII. Representation of the places, transitions, and tokens are discussed in section 6.2.

We propose to develop a program which will detect the presence of attacks in source code by utilizing the library of attack Petri nets. Consider the code to be examined is modeled as a Petri net, P. We also have a library of Petri nets which are Selective Forwarding attack, S and DOS attack, D. To say that the given code has malicious content in it, we have to determine if P has D in it. In other words, we have to find states in P that are the attack signature. This signature could represent S or D. Matching of the application Petri net and attack Petri net gives us a set of matching states and the resources used by the states.

CHAPTER VII

IMPLEMENTATION

The implementation has four parts.

- In the first part we are going to program the attacks on TINYOS 1.1 platform in nesC.
- Second part consists of transforming attacks into Petri nets as intermediate representation.
- Third part consists of embedding the attacks in sensor application and modeling it as a Petri net.
- In the final part, we are going to design and implement a program that will detect malicious code in the (sensor) applications.

In the implementation phase, we have implemented the components Attack Petri nets and Detection Tool.

7.1 Attack Modeling

7.1.1 Selective Forwarding and Sink Hole Attacks

a) **Selective Forwarding:** Selective forwarding attack influences the communication in a Multi hop network [12]. In a Multi hop network, a node forwards a message to its neighbor, thus acting as a forwarder. If a node has been compromised by an attacker, it could launch selective forwarding attack on the network. The malicious node selectively drops few packets. This node selects few nodes randomly and drops packets that are received from them. The compromised node does not drop all the packets. This is because if it drops all the packets, the link quality degrades and the multi hop protocol rejects the node from selecting it as parent node. It also does not drop all the packets as it will raise the suspicion of its neighboring nodes.

   The effectiveness of this attack on the network depends on placement of malicious node with respective to the base station and the number of packets dropped. In this case we assume that the attacker is a compromised node that is in the path to the base station. The closer the attacker is to the base station, the more number of packets are received by it [12].

b) **Sink Hole Attack:** The easiest way of creating a sink hole is to have a malicious node pretend it is a base station [12]. This can cause a big part of the network to start sending their traffic towards that node. How many nodes are affected

depends on the part of the network the malicious node is located in. This is because nodes closer to the real base station will not send traffic towards the malicious node because it is further away than the real base station [12].

In a sinkhole attack, the adversary's goal is to lure nearly all the traffic from a particular area through a compromised node, creating a metaphorical sinkhole with the adversary at the center [22]. Because nodes on, or near, the path that packets follow have many opportunities to tamper with application data, sinkhole attacks can enable many other attacks (selective forwarding, for example). Sinkhole attacks typically work by making a compromised node look especially attractive to surrounding nodes with respect to the routing algorithm. For instance, an adversary could advertise an extremely high quality route to a base station.

Effectively, the adversary creates a large 'sphere of influence', attracting all traffic destined for a base station from nodes several (or more) hops away from the compromised node. By ensuring that all traffic in the targeted area flows through a compromised node, an adversary can selectively suppress or modify packets originating from any node in the area. It should be noted that the reason sensor networks are particularly susceptible to sinkhole attacks is due to their specialized communication pattern [22].

Attack Code

Figure 6 is the code snippet which does selective forwarding in combination with sink hole attack. In the code, the node checks if the link is busy. In an attack scenario,

even when the link is not busy it does not send the packets. In normal cases, a node forwards a packet if the radio signal is not busy. So it first checks for busy radio signal. If the signal is busy the packet is sent, the action of sending a packet is independent of any other conditions. But here, Nodes 2 and 3 are attackers here. They are the sink holes that are in the path to base station. They drop packets selectively.

```
//The packet is receive by a node
event TOS_MsgPtr Receive.receive(TOS_MsgPtr m) {
        TestMsg *message = (TestMsg *)m->data;
        TestMsg *data = (TestMsg *)pkt.data;

        //Check if the packet is meant for this node
        if(m->addr ==TOS_LOCAL_ADDRESS)
        {
                //send the packet to the next hop
                data->source = TOS_LOCAL_ADDRESS;
                data->origin = message->origin;
                data->seqNo = message->seqNo;

                //If the source node is not 2 or 3, forward the packet
                if(TOS_LOCAL_ADDRESS!=2) &&
                                (TOS_LOCAL_ADDRESS!=3))
                        if(call Send.send(parent, sizeof(TestMsg), &pkt))

                        //Node 2 and 3 drop every 1 packet in 20 packets
                                if(attack!=20) &&
                                  ((TOS_LOCAL_ADDRESS==2) ||
                                    (TOS_LOCAL_ADDRESS==3)))
                {
                        if(call Send.send(parent, sizeof(TestMsg), &pkt))
                }

        }
        return m;
}
```

Figure 6: Selective Forwarding

Petri net model of attack

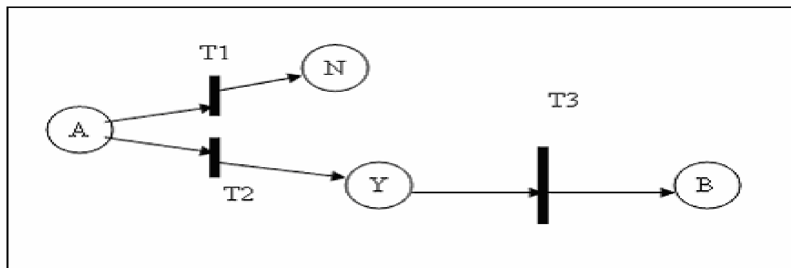


Figure 7: Selective Forwarding and Sink Hole

Places:

A: Attacker node which has been compromised

Y: A condition is satisfied. The packet is forwarded. The packet is selectively forwarded to the neighbor.

N: The condition is not satisfied. Packet is dropped. It is a sink hole.

B: Neighboring node.

Transitions:

T1: Checks for a condition. In the code it is specifically an 'IF' construct. The condition has occurred. Packet is forwarded (conditionally).

T2: Condition failed, packet dropped.

T3: The action of sending a packet. In programming construct it is a 'Send' action.

7.1.2 Sybil Attack

In a Sybil attack, a single node presents multiple identities to other nodes in the network [22]. Sybil attacks pose a significant threat to geographic routing protocols. Location aware routing often requires nodes to exchange coordinate information with their neighbors to efficiently route geographically addressed packets. It is only reasonable to expect a node to accept but a single set of coordinates from each of its neighbors, but by using the Sybil attack an adversary can 'be in more than one place at once' [22].

Attack Code

The following code launches Sybil attack by creating multiple identical nodes. The Sybil nodes send connection establishment requests to neighbors. In Multihop, data of its neighbors is kept in the neighbor table. The table has a maximum size of sixteen. When the table is full and a message from a node that is not in the table is received, the node with the lowest send quality is replaced with the new node. If a Sybil attack node assumes the identity of sixteen nodes it can remove all real neighbors from the neighbor tables of all nodes within its radio range. It can even remove the base station if the fake node's send quality is higher than the one from the base station.

```
//Creating multiple identical nodes, sending requests to neighbors, causing flooding,
request establishment packets
event TOS_MsgPtr Receive.receive(TOS_MsgPtr m) {
```

```
        Msg *message = (Msg *)m->data;
        Msg *data = (Msg *)pkt.data;
        if(m->addr == TOS_LOCAL_ADDRESS)
        {
                dbg(DBG_USR1,"Received message from %d",child);
                // attack code
                attack++;
                //send the packet to next hop
                data->source = TOS_LOCAL_ADDRESS;
                data->origin = message->origin;
                data->seqNo = message->seqNo;
                data->value = message->value;
                if(TOS_LOCAL_ADDRESS != 3) {
                        post addOperation();
                        if(call Send.send(parent, sizeof(Msg), &pkt))
                                dbg(DBG_USR1, "SENT MESSAGE TO %d", parent);
                }
                else{
                        //Sybil nodes flood packets
                        while(i!=15){
                        // sybil node creation
                        parent = TOS_LOCAL_ADDRESS + i;
                        post addOperation();
                        //each sybil node floods packets
                        if(call Send.send(parent, sizeof(Msg), &pkt))
                                dbg(DBG_USR1, "SENT MESSAGE TO %d", parent);
                                i++;
                        }
                }
        }
        return m;
 }
}
```

Figure 8: Code for Sybil Attack

Figure 9 shows the Petri net representation of the code of Figure 8, where four Sybil nodes are created. Each Sybil node then sends each neighboring node connection establishment packets.

Petri net model of Attack

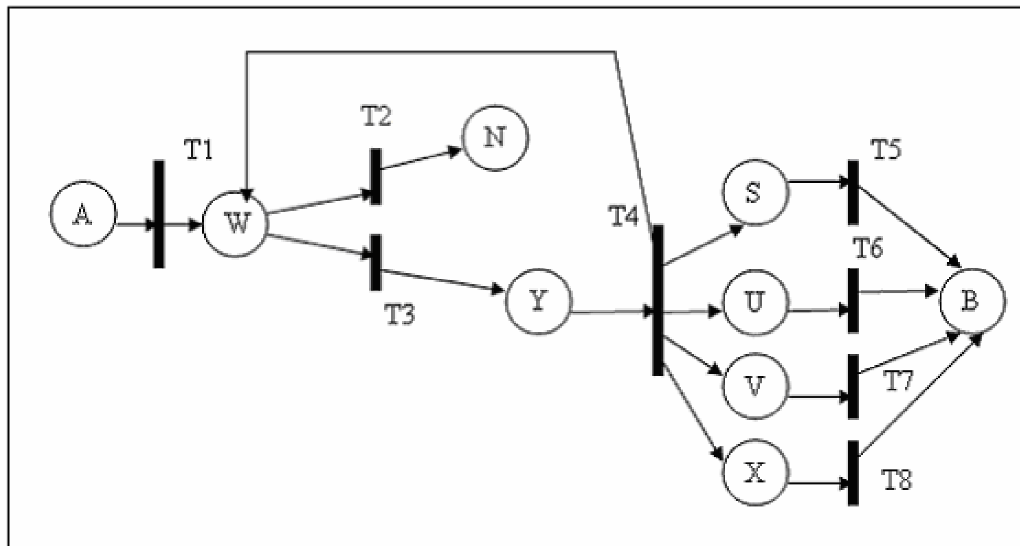


Figure 9: Sybil Attack

Places:

A: Attacker node which has been compromised

W: Checking for occurrence of a condition

Y: A condition is satisfied. Sybil attack is launched.

N: The condition is not satisfied. There are no further transitions.

S, U, V and X: Sybil identities are created. Each Sybil identity sends connection establishment requests to its neighbors.

B: Sensor Node.

Transitions:

T1: Attacker node starts a while loop

T2: Checks for a condition. The condition has not occurred.

T3: Condition is true. There are further transitions.

T4: Firing of T4 creates Sybil nodes.

T5, T6, T7, and T8: Firing of T5, T6, T7 and T8 causes the Sybil nodes to send connection establishment packets to the neighboring nodes.

7.1.3 Denial of Service

Although we usually use the term to refer to an adversary's attempt to disrupt, subvert, or destroy a network, a DoS attack is any event that diminishes or eliminates a network's capacity to perform its expected function [4]. Hardware failures, software bugs, resource exhaustion, environmental conditions, or any complicated interaction between these factors can cause a DOS. Determining if a fault or collection of faults is the result of an intentional DoS attack presents a concern of its own—one that becomes even more difficult in large-scale deployments, which may have a higher nominal failure rate of individual nodes [4].

Some of the forms of denial of service attacks are jamming, flooding and exhausting battery resources. Misdirecting packets is another common form of denial of

service attack. Compromised nodes simply forward packets along wrong paths, thereby diverting traffic away from intended destination.

Attack Code

In this attack scenario, an attacker repeatedly transmits packets to other nodes, in order to prevent them from servicing a request. Flooding is one form of DOS attack which has been implemented here. The attacker sends connection establishment packets to other nodes. Meanwhile if any other node sends a request packet to another node, its request is never satisfied. This is because the request might not even reach the destination as the channel is always busy with the packets from the attacker.

Figure 10 is the code snippet for a DOS attack. Here the attacker is flooding the channel with connection establishment request packets that results in requests not reaching destination and not being processed. Node 3 has been compromised here. In other words it is the attacker and it launches DOS. It is in the path to base station.

```
event result_t Timer.fired() {
        Msg *data = (Msg *)pkt.data;
        count++;
        if((count == 60) && (!update))
        {
                update = TRUE;
        }
        //Checking if the link is busy
        if(!pending) {
                pending = TRUE;
                data->origin = TOS_LOCAL_ADDRESS;
                data->source = TOS_LOCAL_ADDRESS;
                data->seqNo = seqNo;
                //DOS attack. Node 3 is the attacker, attacks by flooding packets
                if(TOS_LOCAL_ADDRESS ==3)
                {
                for(i=0;i<10;i++)
                {
                if(call Send.send(parent, sizeof(Msg), &pkt))
                {
                        dbg(DBG_USR1, "SENT MESSAGE TO %d", parent);
                        seqNo++;
                        return SUCCESS;
                }
                pending = FALSE;
                }
                }
        }
        return FAIL;
}
```

Figure 10: Denial of Service

Petri nets model of attack

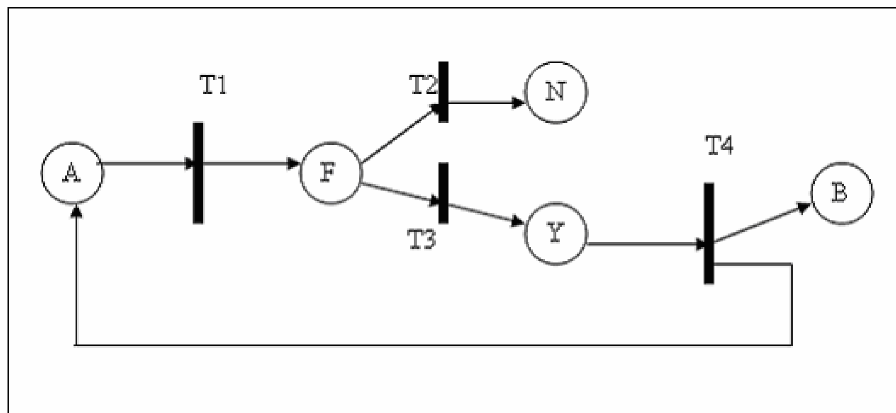


Figure 11: Denial of Service

Places:

A: Attacker node which has been compromised.

F: FOR loop begins.

Y: A condition is satisfied. Packets are flooded causing DOS attack.

N: The condition is not satisfied. There are no further transitions and attack stops.

B: Neighboring node.

Transitions:

T1: Attacker node starts a 'FOR' loop. T1 is fired when attacker receives a packet to forward. T1 places a token in places.

T2: Checks for a condition. T2 is fired when condition false

T3: T3 is fired when condition true.

T4: The action of sending a packet. In programming construct it is a 'Send' action. A token is placed in places B and A. Now T1 can fire again and the process continues. This results in the attacker repeatedly sending packets.

7.2 Application Modeling

We have used a sensor application to embed attacks in it. A sensor application resides in the sensor node and it has a particular function. The particular application that we developed has each node in the sensor network sending a packet to base station every 1000 milliseconds. When a node sends a packet to its neighbor, other nodes forward the packet to the base station. The attackers are in the path towards the base station. When the attacker node receives the packet it launches that particular attack.

```
//If the radio signal is not busy
If(!pending)
{
        // Set the radio busy
        pending = TRUE;

        //Set the message origin to local address
        data->origin = TOS_LOCAL_ADDRESS;

        //Set the message source to local address
        data->source = TOS_LOCAL_ADDRESS;

        //Set the seqNo
        data->seqNo =seqNo;

        //Send the packet to next neighbor and update seqNo
        if(call Send.send(parent, sizeof(TestMsg), &pkt))
        {
                seqNo++;
                return SUCCESS;
        }

        //Set radio to "not busy"
        pending = FALSE;
}
```

Figure 12: Application

Petri net Model of Application

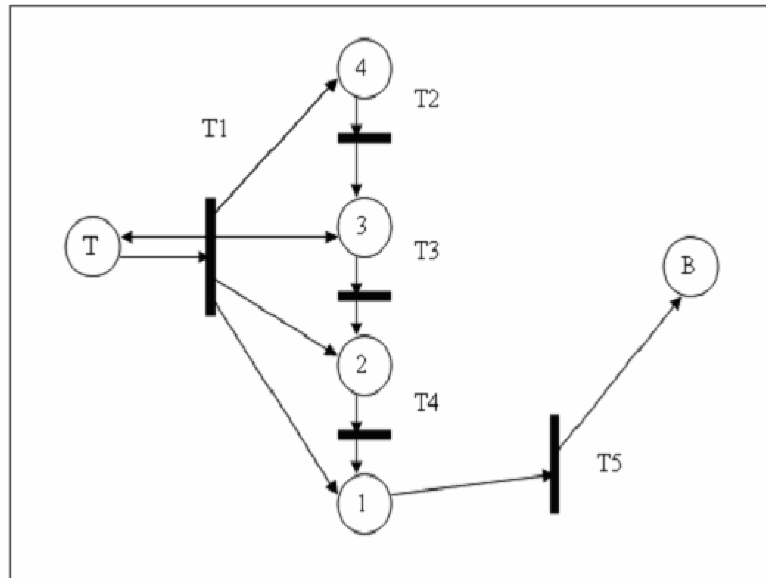


Figure 13: Application

Places:

T: This is a place for 'Timer' or a clock.

4, 3, 2 and 1: Sensor nodes.

B: Base station

Transitions:

T1: Timer firing. Nodes 4, 3, and 1 can now send packets with destination as B.

T2, T3, T4 and T5: The transitions represent 'Send' actions.

In the above figure 4, 3, 2, 1 and B are sensor nodes. Timer is clock. For every 1000 msec, the nodes start sending packets with destination as node B.

7.3 Application with Attacks

7.3.1 Selective forwarding attack and Sink Hole

The colored portion in the Figure14 shows the sensor application that has selective forwarding embedded in it. This is a global scenario in which the network has 4 nodes and a base station. Each node sends packet to base station. Node 4 sends a packet to node 3. Node 3 selectively forwards it to node 2. Node 2 forwards the packet to node 1, which in turn forwards it to the base station.

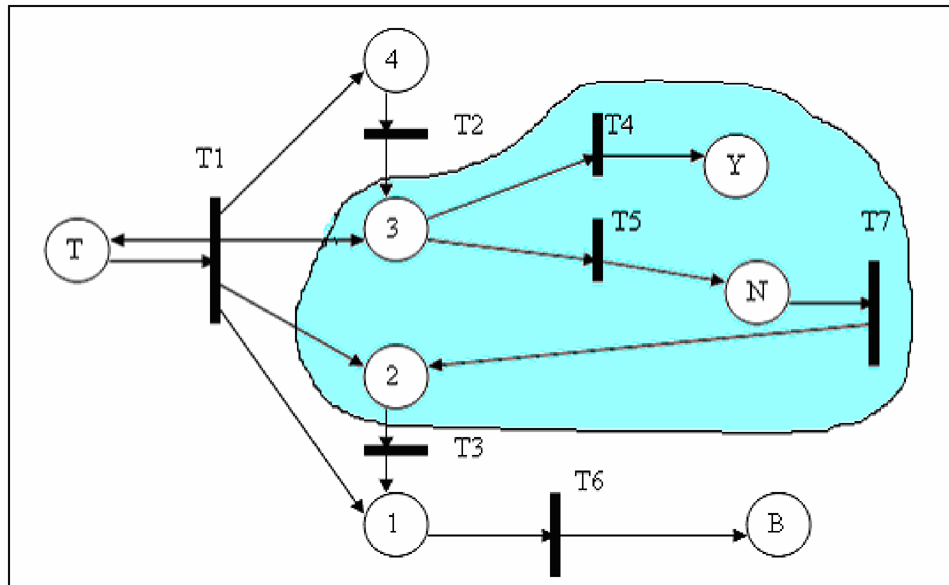


Figure 14: Application with Selective forwarding attack

Places:

T: This is a place for 'Timer'.

4, 3, 2 and 1: Sensor nodes.

B: Base station

Y: A condition is satisfied. No further transitions.

N: The condition is not satisfied. The packet is selectively forwarded to the neighbor.

Transitions:

T1: Timer firing

T2, T3: The transitions represent 'Send' actions.

T4: In the code it is specifically an "IF" constructs. Condition true.

T5: Condition false.

T6, T7: The action of sending a packet. In programming construct it is a ‘Send’ action.

7.3.2 Sybil attack

The colored portion in the Figure 15 shows sensor application that has Sybil attack embedded in it. This is a global scenario in which the network has 4 nodes and a base station. Each node sends packet to base station. Node 4 sends a packet to node 3. Node 3 is the attacker who creates Sybil nodes. These Sybil nodes send connection establishment request packets to neighboring nodes.

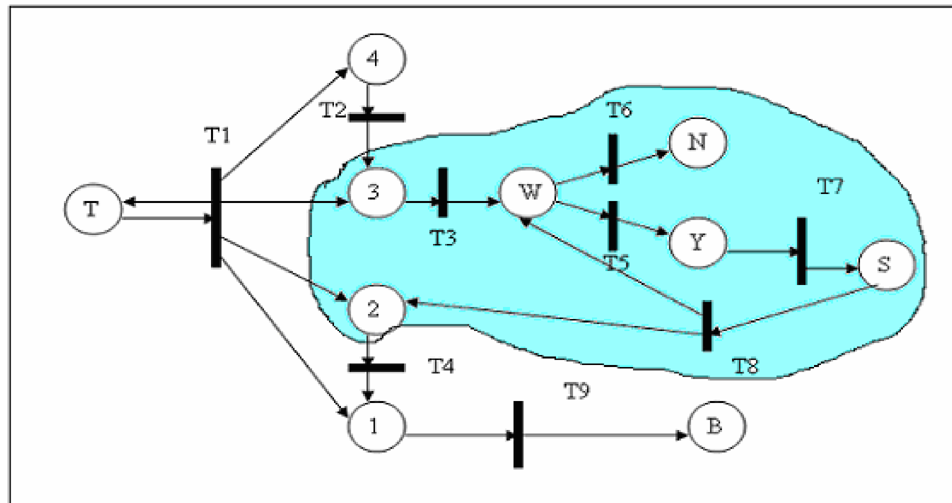


Figure: 15 Application with Sybil Attack

Places:

T: Timer

4, 2 and 1: Sensor nodes.

3: Attacker node which has been compromised

W: A while loop starts.

Y: A condition is satisfied. Create a Sybil node.

N: The condition is not satisfied.

B: Base station.

Transitions:

T1: Attacker node starts a WHILE loop.

T2, T3 and T4: The action of sending a packet. In programming construct it is a 'Send' action.

T5: Checks for a condition. In the code it is specifically a 'while' construct. Condition is true.

T6: Checks for a condition. In the code it is specifically a 'while' construct. Condition is false.

T7: Creates a Sybil identity.

T8: The action of sending a packet. In programming construct it is a 'Send' action. Also it represents a loop where packets are flooded until the condition is satisfied.

T9: The action of sending a packet. In programming construct it is a 'Send' action.

7.3.3 Denial of Service

Figure 16 shows a global scenario in a wireless sensor network. The above sensor application has Denial of Service attack embedded in it. Attacker node floods other nodes with packets causing the other requests to be not processed.

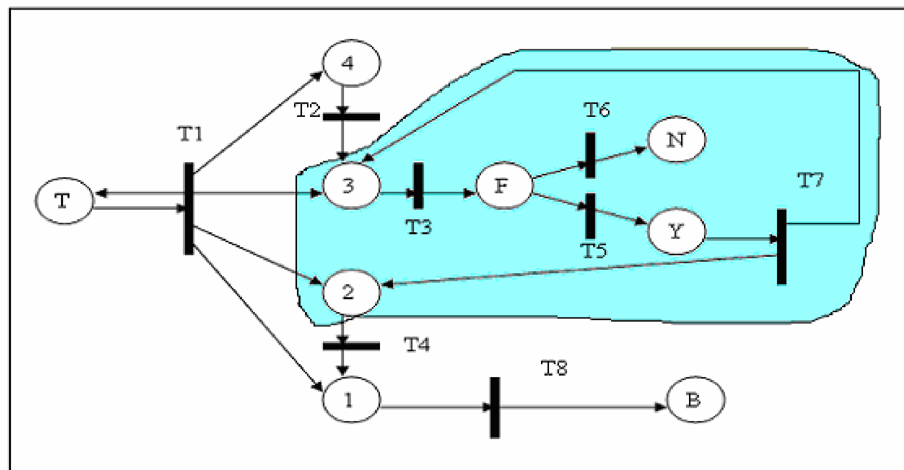


Figure 16: Application with Denial of Service

Places:

T: Timer

4, 2 and 1: Sensor nodes.

3: Attacker node which has been compromised

Y: A condition is satisfied. Packets are flooded causing DOS attack.

N: The condition is not satisfied. No further transitions take place.

B: Base station.

Transitions:

T1: Timer fires.

T2 and T4: Each node starts sending packets to the base station. In programming construct it is a “Send” action.

T5: Checks for a condition. In the code it is an ‘IF’ construct. Condition true.

T6: Checks for a condition. In the code it is an ‘IF’ construct. Condition false

T7: The action of sending a packet. In programming construct it is a ‘Send’ action. Also it represents a loop where packets are flooded until the condition is satisfied. T7 also places a token in place 3. Because of this T3 repeatedly fires and the process is repeated.

T8: The action of sending a packet. In programming construct it is a ‘Send’ action.

7.4 Detection Tool

7.4.1 Code to Petri net transformation

For transforming the code to Petri nets, we examine the events present in the application. Events cause some actions to take place. Some of the events in TinyOS are ‘Send’ and ‘Receive’

Send: “Send” is an event of sending a packet to the neighboring node.

Receive: Receive is an event of receiving a packet from neighboring node. After receiving the packet, a node checks if the packet is meant for it. Then it forwards the packet to its neighbor. It is in this situation where the different types of attacks take place. If the forwarder node becomes compromised, it launches the attacks.

When there is a send and receive event, it means that when a node is sending, another node is receiving, so we have two places to represent the source and destination nodes.

'IF' construct is modeled as follows:

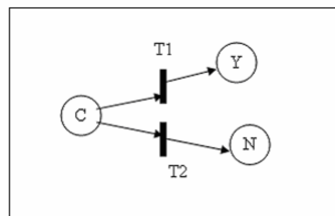


Figure 17: IF Construct

C is a place for the 'IF' construct. 'IF' construct has two execution (flow of control) paths represented by places Y and N. Transition T1 is fired if the condition in 'IF' statement is satisfied. Transition T2 is fired if the condition in 'IF' statement is not satisfied.

When a 'FOR' construct occurs, we represent it as follows:

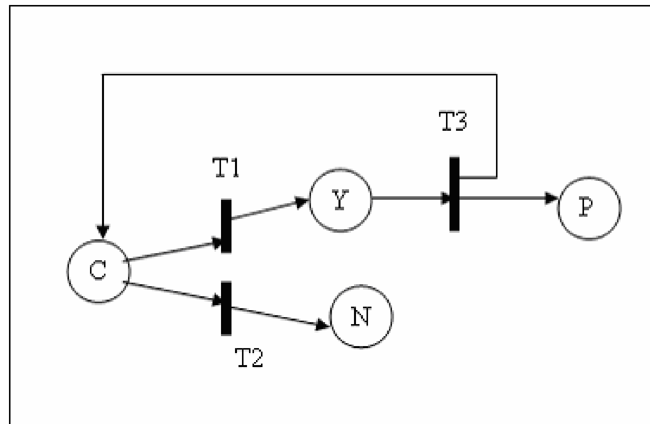


Figure 18: FOR Construct

When a ‘WHILE’ statement occurs it is represented as follows:

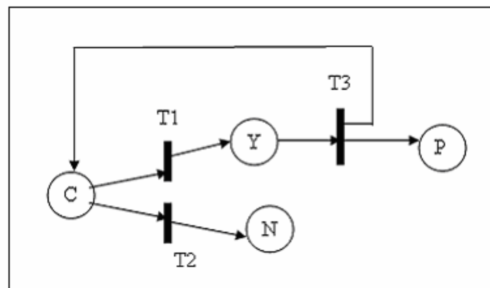


Figure 19: WHILE Construct

When the event ‘Send’ occurs, they are represented as in Figur 20. The dot in S is a packet which is being sent to R.

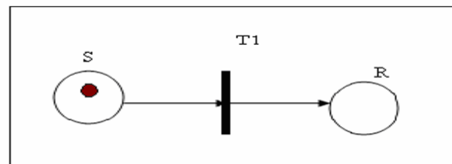


Figure: 20 Event Send

Data Structures Used:

We have used adjacency lists to represent Petri nets. Each petri nets has 3 lists, one list shows links between Places to Transitions and another table shows links from Transitions to Places. The third list shows what each transition and place stand for. In other words it indicates if a place is a node or a program construct like ‘IF’, ‘FOR’, or ‘WHILE’ and whether a transition is a Send/Receive or a flow of control.

Adjacency lists are also convinient when we try to establish a sequence between the states and transitions.

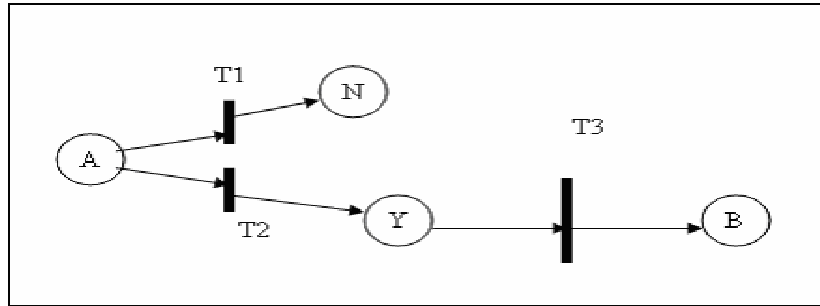
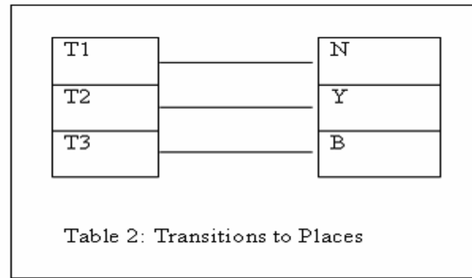


Figure: 21 Example to illustrate data structures

Place		
A	T1	T2
N		
Y	T3	
B		

Table 1: Places to Transitions



T1	N
T2	Y
T3	B

Table 2: Transitions to Places

Nodes	A	B	
IF	A	T1	T2
YES	Y		
NO	N		
SEND	T3		
RECEIVE			

Table 3: Category of places and transitions

7.4.2 Detection Algorithm

Input: Application and attack Petri nets.

Output: Sequence of instructions that represent an attack found in A.

1) Take input as the sensor application in nesC.
2) Initialize the adjacency lists for attack Petri nets. Attack Petri nets are SFi_{ji} (Selective Forwarding), $S_{i,j}$ (Sybil) and $D_{i,j}$ (Denial of Service) .
3) Initialize each Category List $SFC_{i,j,}$ $SC_{i,j,}$ $DC_{i,j}$ respectively.
4) Start scanning the application to find events.
5) When an event 'Send' or 'Receive' is found, construct Petri net $A_{i,j}$ from the program constructs 'IF', 'FOR', 'WHILE', Send.send().
6) Construct a Category list for the above application Petri net $AC_{i,j}$ with categories as NODES, SEND, RECEIVE, IF, FOR, WHILE, YES, NO.
7) Prove Existence violation category. Take the attack library Petri net Category lists compare them with the Category list of the application net. If some set of states are matching with a particular attack, then goto STEP 8 otherwise goto STEP 10.
8) Prove Sequence violation category. Take the adjacency lists of that particular attack found in STEP 7. Compare the adjacency lists of application with the attack to check if the sequences between the states are present. If the attack sequence is found in the application then STEP 9 otherwise goto STEP 11.
9) Return 'Attack found', also return the sequence of attack instructions.
10) Return 'Attack not found'.

11) Return ‘Potential for an attack’ as STEP 7 found some random states which are not in a sequence.

12) End

7.4.3 Validation of Algorithm

The Detection Tool produces a Petri net from the application and it uses the attack Petri nets to find the presence of attacks in the application. If the attack is found it returns the sequence of instructions of the attack. The detection algorithm has been programmed in Java.

The Detection Tool was tested on the following:

1) Applications consisting Selective Forwarding (with Sink Hole), Sybil and Denial of Service were input to the Detection Tool and the tool detected the presence of attacks. It returned the sequence of attack instructions. Table 4 is showing Selective Forwarding.

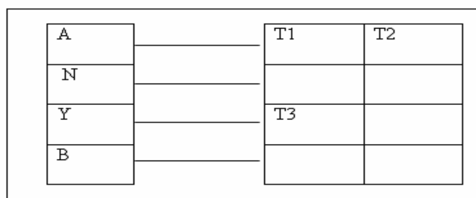


Table 4: Places to Transitions, sfPT

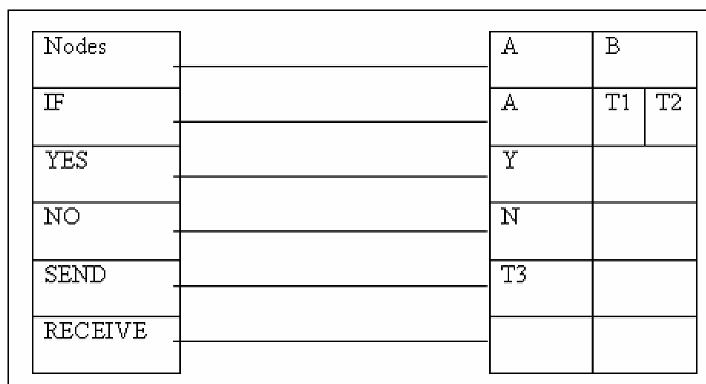


Table 5: Transitions to Places, sfTP

Nodes	A	B	
IF	A	T1	T2
YES	Y		
NO	N		
SEND	T3		
RECEIVE			

Table 6: Category of places and transitions, sfC

Example: Applicationon with Selective Forwarding

S1	T4	T5
Y	T7	
N		
S2		
TIMER	T1	

Table 7: Places to transitions, appPT

T1	Timer	S1	S2
T4	N		
T5	Y		
T7	S2		

Table 8: Transitions to Places, appTP

Nodes	A	B	
IF	A	T1	T2
YES	Y		
NO	N		
SEND	T3		
RECEIVE			
FOR			
WHILE			

Table 9: Category table, appE

The process of detection is as follows

- Prove Existence

  Number of nodes in appC = Number of nodes in sfC

  appC has IF construct, so does sfC

  appC has SEND event so does sfC

  A match with selective forwarding attack.

  Hence the attack states exist.

- Prove Sequence

sfC(Nodes) = A	appC(Nodes) = S1
sfPT(A) = T1, T2	appPT(S1) = T4,T5
sfQueue = T1,T2	appQueue = T4,T5
sfC(T1,T2) = IF	appC(T4,T5) = IF

sfQueue(head) = T1

sfPT(T1) = N

add N to sfQueue(tail)= T1,T2,N

sfC(N) =NO

delete sfQueue(head) = T2, N

appQueue(head) = T4

appTP(T4) = N

add N to appQueue(tail) = T4,T5,N

appC(N) =NO

delete appQueue(head) =T5, N

sfQueue(head) = T2

sfTP(T2) = Y

sfC(Y) = YES

add Y to sfQueue = T2, Y

delete T2 from sfQueue = Y

appQueue(head) = T5

appTP(T5) = Y

appC(Y) = YES

add Y to appQueue = T5,Y

delete T5 from appQueue = Y

sfQueue(head) = Y	appQueue(head) = Y
sfPT(Y) = T3	appTP(T5) = T7
add T3 to sfQueue(tail) = N,T3	add T5 to appQueue(tail) = T5,T7
sfC(T3) =SEND	appC(T7) = SEND
delete sfQueue(head) = SEND	delete appQueue(head) = T7

sfQueue(head) = T3	appQueue(head) = T7
sfTP(T3) = B	appTP(T7) = S2
sfC(B) = Node	appC(S2) = Node
sfPT(B) = { }	appPT(B) = { }

All the states are in a sequence, so there is Sequence violation.

Since there is existence and sequence, the application has selective forwarding attack.

2) Application without any particular attack was tested and it resulted in the output 'Potential for attack'. It also gave the Petri nets states and transitions found. Here the Petri net found did not match any particular attack. There is no sequence in the states found. But some of the states are random and do match. This could mean there is a potential for attack.

Example: Application with potential for attacks

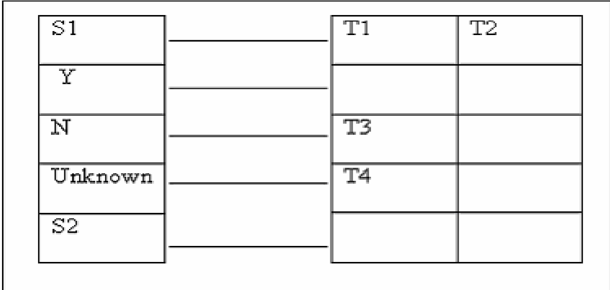


Table 10: Places to transitions, appPT

Transition	Place		
T1	Y		
T2	N		
T3	P		
T4	S2		

Table 11: Transitions to Places, appTP

Nodes	S1	S2	
IF	S1	T1	T2
YES	Y		
NO	N		
SEND	T4		
RECEIVE			
FOR			
WHILE			

Table 12: Category table, appE

The detection process is as follows:

- Prove Existence

  Number of nodes in appC = Number of nodes in sfC

  Number of nodes in appC = Number of nodes in DC

  Number of nodes in appC = Number of nodes in SC

  appC has IF construct which matches with Selective forwarding.

  appC has SEND event so does sfC, DC, SC

  There is existence of some attack states. One of the states matches with Selective forwarding. Another state matches with sybil and denial of service attack.

  So we can say that there is a potential for an attack.

3) Application without attacks was input to the Tool and it resulted in 'No attacks found'. Table 13 shows an example without any attacks.

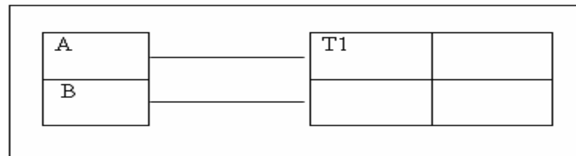


Table 13: Places to transitions, appPT

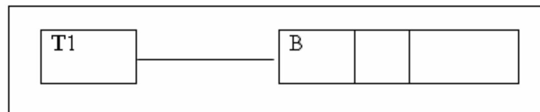


Table 14: Transitions to Places, appTP

Nodes	A	B
IF		
YES		
NO		
SEND	T1	
RECEIVE		
FOR		
WHILE		

Table 15: Category table, appE

The detection process is as follows

- Prove Existence

  Number of nodes in appC != Number of nodes in sfC

  Number of nodes in appC != Number of nodes in DC

  Number of nodes in appC != Number of nodes in SC

  appC has no IF/FOR/WHILE construct.

  appC has SEND event which is not an attack state.

There is no match between the attack net and the application net. Hence, no existence. Since there is no existence of any attack states, we can say that there is no attack in the application.

7.4.4 Complexity of the algorithm

The main constraint is space due to limited storage in a sensor node. Memory is needed for adjacency lists of attacks and application. We are also using two FIFO queues to establish a 'sequence' between the petri net states. We store petri net places and transitions in the queue. This algorithm uses a strategy similar to Breadth First Search (BFS) Algorithm. BFS uses a FIFO queue to put the root node in the queue and it explores all the unexplored nodes.

An alternative to adjacency list is an adjacency matrix to represent petri nets. But matrices occupy more space than a list. Use of adjaceny lists over matrices has considerably reduced the use of memory.

Advantages of adjacency lists:

1. If the Petri net is not very dense, this representation is great, because it doesn't waste memory locations for non-existent edges.
2. It is easy to list all of the edges coming out of a node.

From a computational perspective of the algorithm, it heavily relies on finding programming patterns in the application. When patterns are found a petri net is

constructed. For the detection of attacks, the algorithm searches the adjacency lists of attacks and the application to match the sequence of states. Therefore this algorithm does not depend on complex computations of any kind. Computations if any have to be kept very low as the nodes do not have high processing capabilities.

The algorithm tries to match the petri nets by establishing a ‘sequence’ between the states. A petri net is a bipartite graph of nodes P (that are places) and transitions T (that are edges). In the worst case when the sensor application is large, the petri net constructed is dense. The algorithm will have a large number of places and transitions to examine. The complexity depends on the denseness of the application Petri net.

The complexity of the detection algorithm is Big oh(P+T+A), where P is the number of places, T is the number of transitions and A is the number of arcs in application Petri net. Therefore, for these attacks the algorithm is practical for sensor networks.

CHAPTER VIII

CONCLUSIONS

In this thesis we investigate the vulnerabilities present during code distribution process and detect the malicious code present in sensor applications. The approach to use Petri nets to model the attacks helps us in understanding the behavior of the attacks and find patterns in attacks. We programmed the attacks in nesC to find patterns in them. The attack code was then transformed into Petri nets. We then inserted malicious code/attacks in a sensor application. The application was transformed into a Petri net as well. Using the attack Petri nets and application Petri net, we tried to detect the attacks in the application.

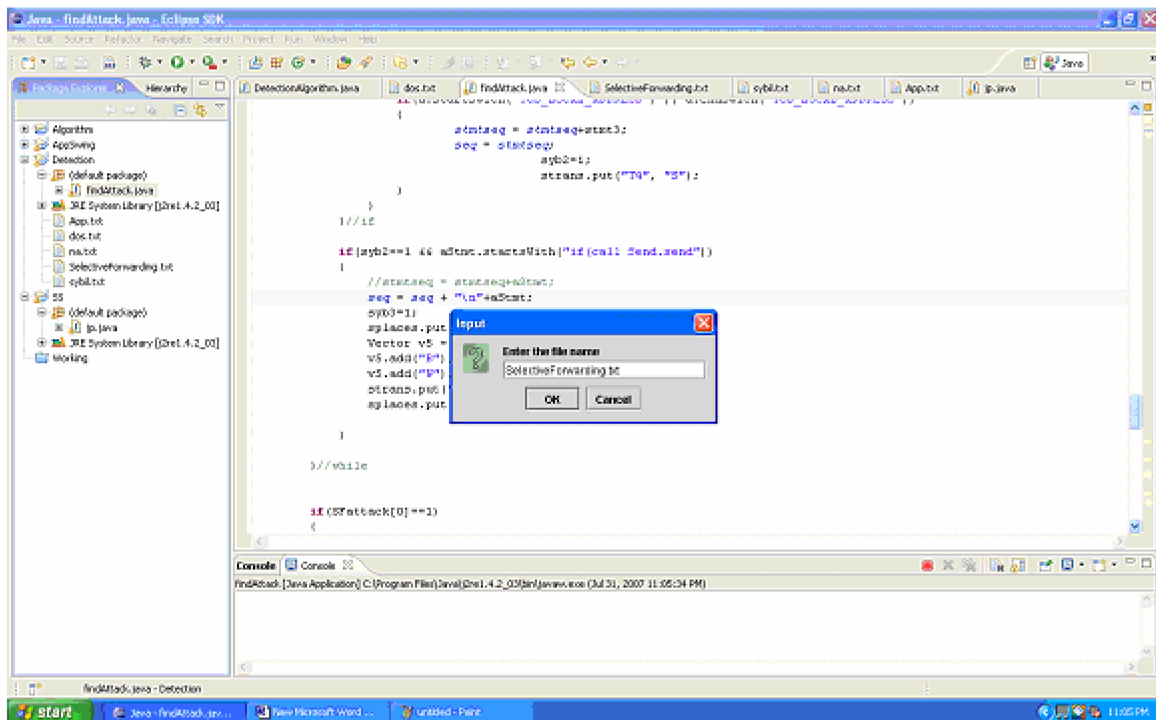
To experimentally validate our proposed model, we have developed a Detection Tool that will use the attack Petri nets to detect malicious code in the applications. This tool when tested on a sensor application detected the particular attacks present in it. It detected the sequence of instructions that closely matched the states in the attack Petri nets.

In future this approach can be expanded to identify more attacks and behavior of those attacks. It also consists of finding more categories for matching the structures of application and attack Petri nets. Future work in this area includes working on executable code to find attack signatures. Working on the executable code would eliminate the need for a decompiler. The sensor application code may also contain a combination of attacks such as Denial of Service with selective forwarding. The code could also contain obfuscated attacks. In the case of obfuscated attacks, the attacks are hidden within the code, in other words they do not look like attacks, as they are not continuous. Therefore, there are a lot of possibilities to expand this work in future.

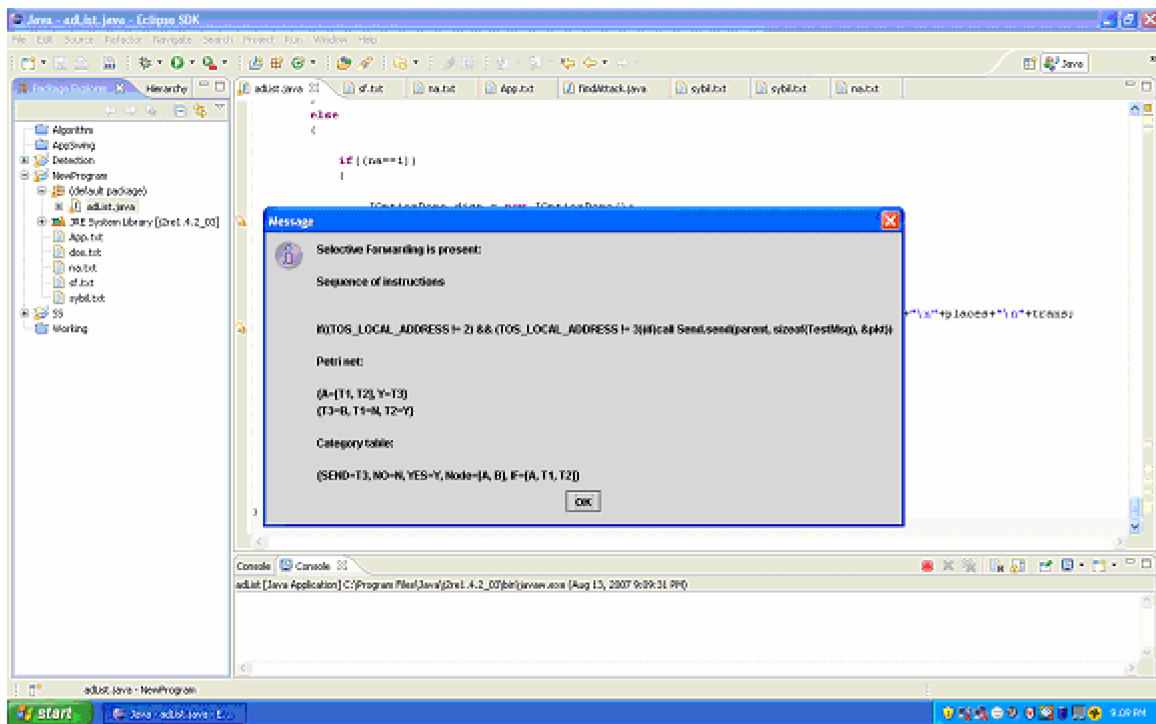
APPENDIX

I. Input: SelectiveForwarding.txt

The screen shot below shows a dialog box which asks the name of the sensor application file. The Detection Tool when run prompts for the name of the file. The name of the file “SelectiveForwarding.txt” is given as input here.

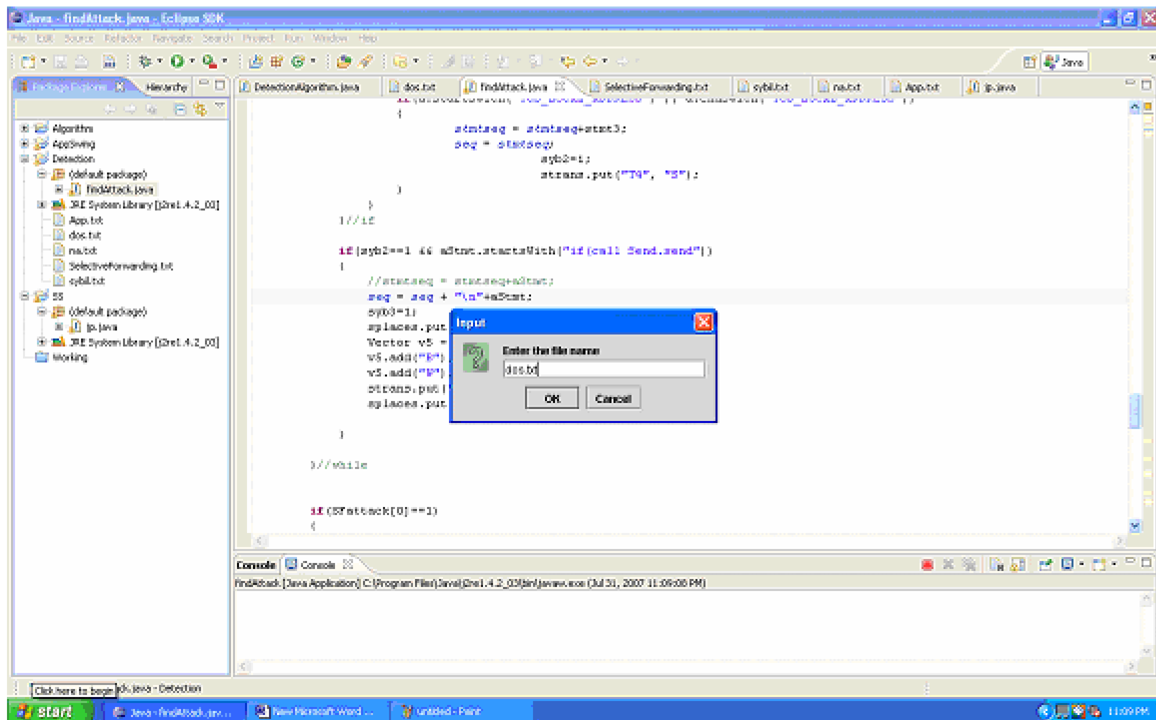


Output: The screen shot below shows the output when the input is given to the detection tool. It recognizes the attack patterns in the input application. The dialog box below shows that the application has selective forwarding in it. The sequences of instructions which are malicious are shown. Also the Petri net representation for that particular attack is shown. A, Y, N and B are places. T1, T2 and T3 are transitions.

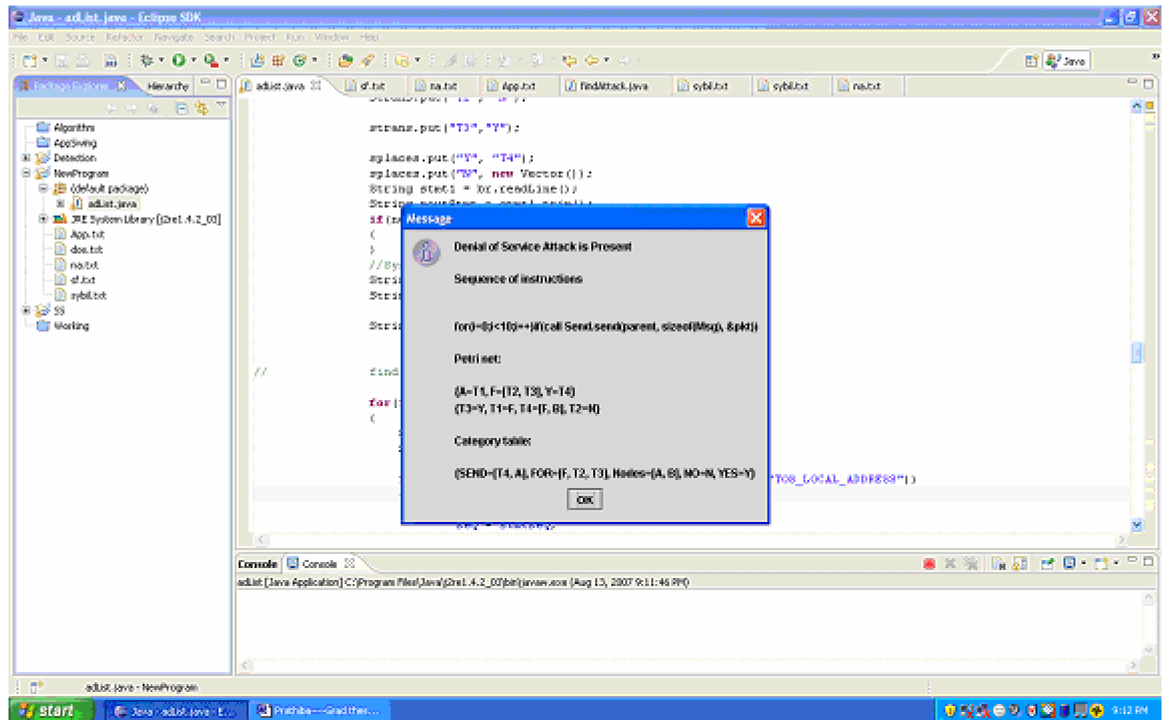


II. Input: dos.txt

This screen shot shows the input dialog box. But this time the input given is dos.txt which contains the nesC code of the application.

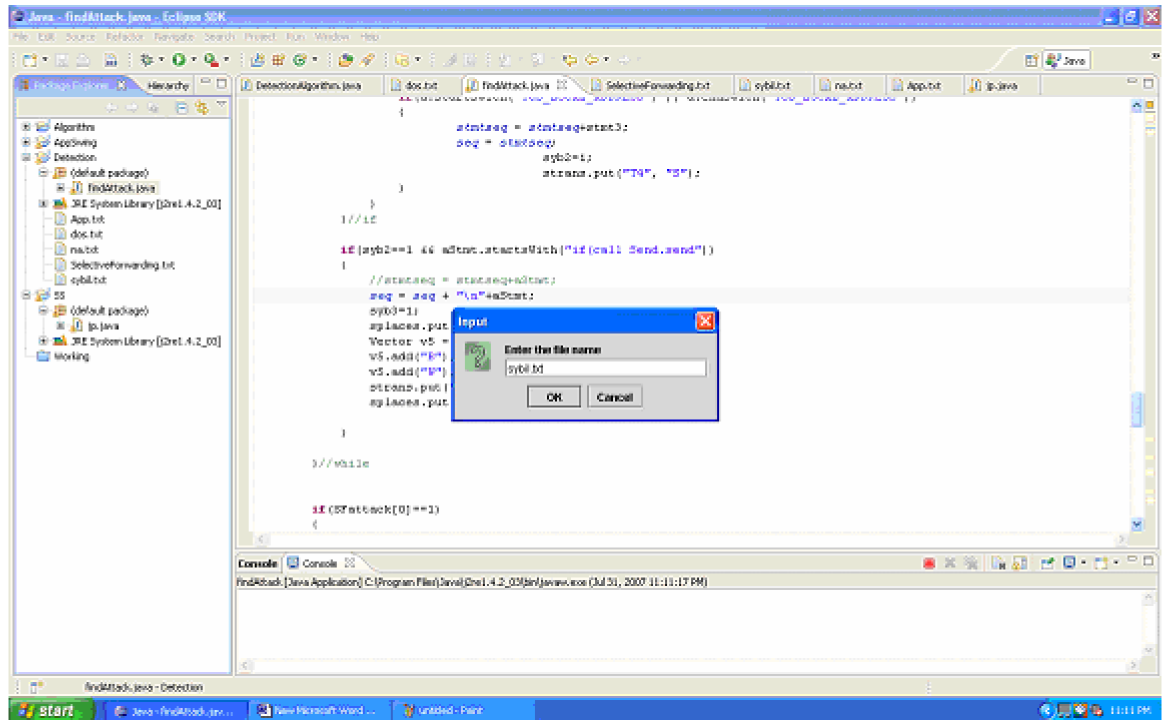


Output: This screen shot shows the output which shows that the application has denial of service attack present in it. The sequences of instructions along with the Petri net representations of the attack are shown.

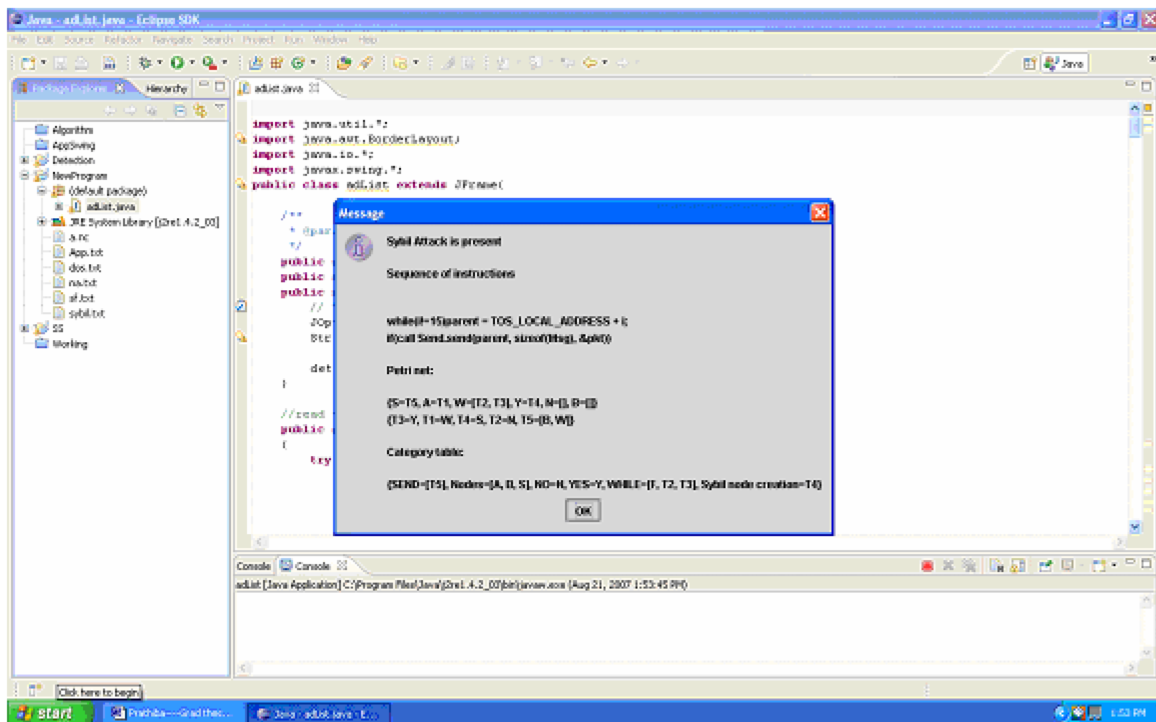


III. Input: sybil.txt

This screen shot shows the input dialog box where the input is give n as a file containing nesC code. The file is sybil.txt

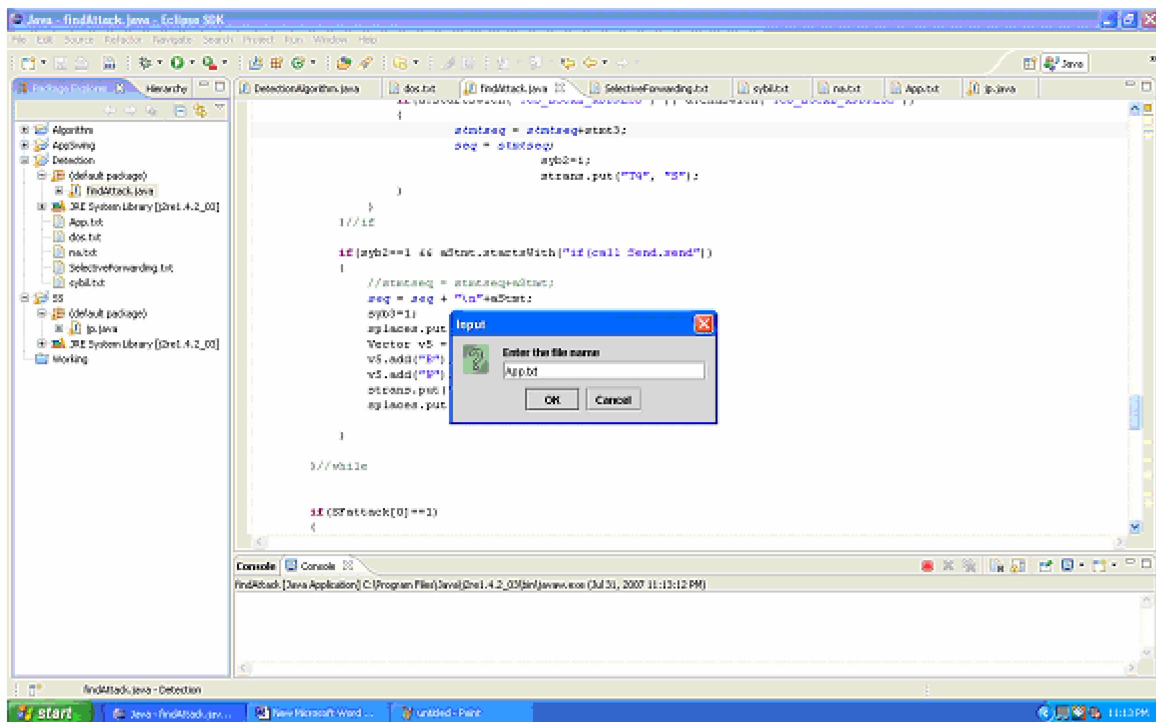


Output: This screen shot shows the output when an application infected with Sybil attack is given to the Detection Tool. The output shows that Sybil attack is found. The sequences of corrupt instructions and the Petri net representation of the attack are shown.

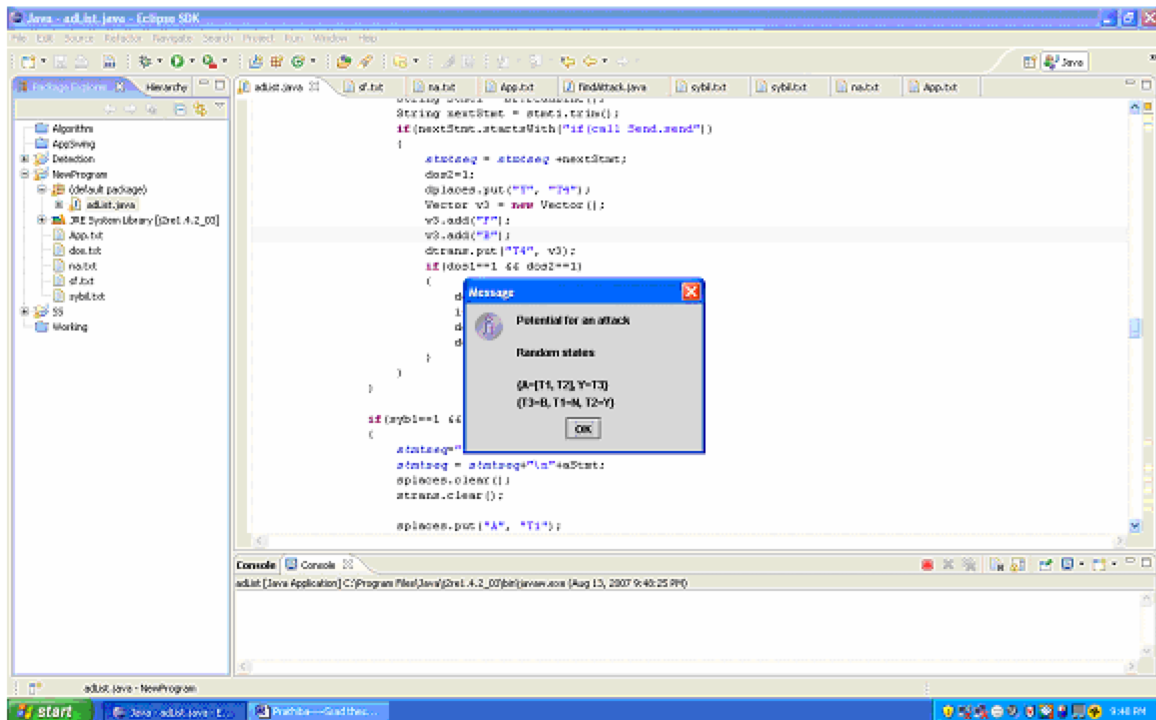


IV. Input: App.txt

This screen shot shows the input given to the detection tool which is application without any particular attacks.

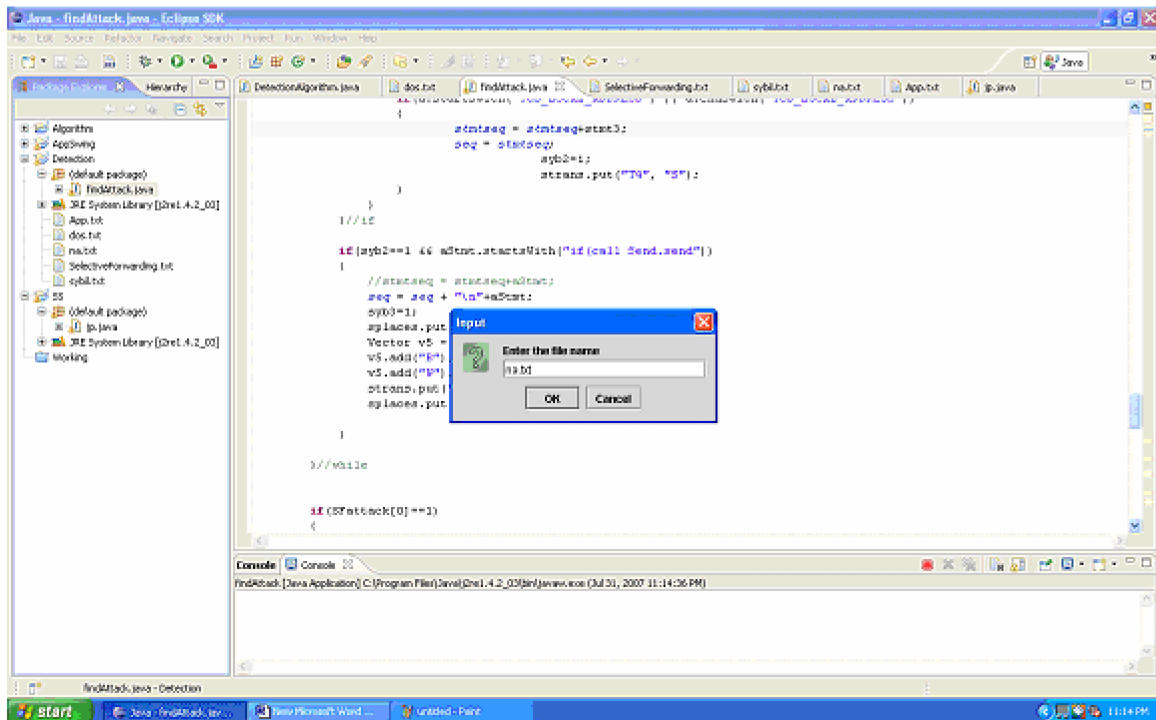


Output: This screen shot below shows the output . The output is “Potential for attack”. It also shows the Petri nets states and transitions found. Here the Petri net found does not match any particular attack. There is no sequence in the states found. But some of the states do match. This could mean there is a potential for attack.

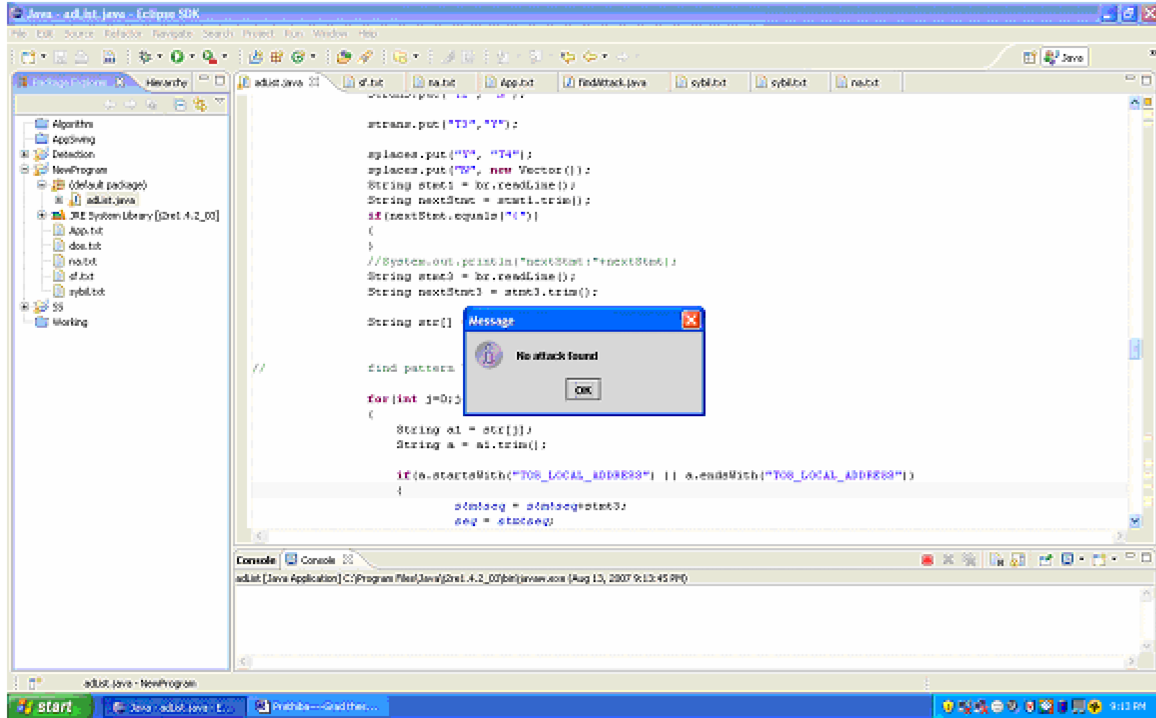


V. Input: na.txt

This screen shot shows the input given to the Detection Tool.



Output: The output shows that ‘No attack found’. There are no attacks in the application.



REFERENCES

[1] Reijers N. and Langendoen K., “Efficient Code Distribution in Wireless Sensor Networks”, Proceedings of the 2nd ACM International conference on Wireless sensor networks and applications WSNA '03, 2003

[2] Stathopoulos T., Heidemann J. and Estrin D., “A Remote Code Updation Mechanism for Wireless Sensor Networks”, Technical Report Centre for Embedded Networked Sensing (CENS), 2003

[3] Bergeron J., Debbabi M., Desharnais J., Erthioui M.M., Lavoie Y. and Tawbi N., “Static Detection of malicious Code in Executable Programs”

[4] Wood A.D and Stankovic J.A., “A Taxonomy for Denial-of-Service Attacks in Wireless Sensor Networks”, Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems, CRC Press, 2004 (invited paper).

[5] McDermott J.P., “Attack Net Penetration Testing**”,** Proceedings of the 2000 workshop on new security paradigms, pp. 15-21, 2001

[6] Ho Y., Frincke D. and Tobin Jr. D. “Planning, Petri Nets and Intrusion Detection”, 21st Proceedings of National Information Systems Security Conference, pp. 346-361, 1998

[7] Romer K, “Programming Paradigms and Middleware for Sensor Networks”, Institute for Pervasive Computing, Zurich

[8] Lifton J., Seetharam D., Broxton M., and Paradiso J., "Pushpin Computing System Overview: A Platform for Distributed, Embedded, Ubiquitous Sensor Networking", Proceedings of the First International Conference on Pervasive Computing, 2002

[9] Lanigan P.E., Gandhi R. and Narasimhan P., "Poster Abstract: Secure Dissemination of Code Updates in Sensor Networks", Proceedings of 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys '05), pp. 278 - 279. November 2005

[10] Porras P.A., Cheung S., Almgren M., "Malicios Code Outbreak Discovery: Issues and Approaches", International Position Paper, DARPA Malicious Code Defense Workshop, Denver, Colorado, August 21-23, 2002

[11] Deng J., Han R. and Mishra S., "Secure Code Distribution in Dynamically Programmable Wireless Sensor Networks", Proceedings of the fifth international conference on Information processing in sensor networks, pp. 292 – 300, 2006

[12] Datema S., "A Case Study of Wireless Sensor Network Attacks", MS Thesis in Computer Science, Delft University of Technology

[13] Voght H., Ringwald M., Strasser M., "Intrusion Detection and Failure Recovery in Sensor Nodes", Tagungsband INFORMATIK 2005, Workshop Proceedings, LNCS, Heidelberg, Germany, September 2005.

[14] Angelo G., "Tutorial on Petri Nets", ACM SIGSIM Simulation Digest archive pp. 10-25, 1983

[15] Petri Nets, http://www.cse.fau.edu/~maria/COURSES/CEN4010-SE/C10/10-7.html [last accessed – April, 2007]

[16] Yao W. and He X., "Mapping Petri Nets to Parallel Programs in C, C++", Department of Computer Science, North Dakota State University, COMPSAC '96 - 20th Computer Software and Applications Conference, 1996.

[17] Ali H., "A New Model for Monitoring Intrusion Based on Petri Nets", Information Management & Computer Security, Volume 9, pp. 175-182, 2001

[18] De P., Liu Y., and Das S., "Modeling Node Compromise Spread in Wireless Sensor Networks using Epidemic Theory", Proceedings of the 2006 International Symposium on World of Wireless, Mobile and Multimedia Networks, pp. 237 – 243, 2006

[19] Zhang Q., Yu T. and Ning P., "A Framework for Identifying Compromised Nodes in Sensor Networks", Proceedings of 2nd IEEE Communications Society/CreateNet International Conference on Security and Privacy in Communication Networks (SecureComm 2006), pp. 1- 10, August 2006

[20] Godse A., "Petri Net Based Model for Protocol Damage Estimation and Protection", MS Thesis, Department of Computer Science, Oklahoma State University, 2006

[21] Da Silva A., Martins M., Rocha B., Lourero A., Ruiz L. and Wong H., "Decentralized Intrusion Detection in Wireless Sensor Networks", Proceedings of the 1st ACM international workshop on Quality of service & security in wireless and mobile networks, pp. 16 – 23, 2005

[22] Karlof C., Wagner D., "Secure Routing in Wireless Sensor Networks: Attaks and Countermeasures", Proceesings of first IEEE International Workshop on Sensor Network Protocols and Applications, pp. 113 - 127, 2003

[23] Christodorescu M., Jha S., "Static Analysis of Executables to Detect Malicious Patterns", Proceedings of the 12th USENIX Security Symposium, pp. 169 -186, August 2003

[24] Slijepcevic S., Tsiatsis V., Zimbeck S., Potkonjak M. and Srivastava M., "On Communication Security in Wireless Ad-Hoc Sensor Networks", Proceedings of the 11th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, pp. 139 – 144, 2002

[25] Levis P., "Viral Code Propagation in Wireless Sensor Networks", EECS Department, University of California, Berkeley

[26] Hui J., "Deluge 2.0 TinyOS Network Programming", Fourth International TinyOS Technology Exchange (TTX4) 2004, www.cs.berkeley.edu/~jwhui/research/deluge/deluge-manual.pdf [last accessed - May 24, 2004]

[27] Hui J. "Deluge 2.0 Tiny OS Network Programming", University of California, Berkely, www.tinyos.net/ttx-02-2005/developments/ttx2005-deluge.ppt [last accessed - May 24, 2004]

VITA

Prathiba Reddy Nalabolu

Candidate for the Degree of

Master of Science

Thesis: DETECTING MALICIOUS CODE IN SENSOR NETWORK APPLICATIONS USING PETRI NETS

Major Field: Computer Science

Biographical:

Personal Data: Born in Andhra Pradesh, India, on June 8th, 1984, the elder daughter of Ramana Nalabolu and Uma Nalabolu.

Education: Graduated from Neeraj Public School, Hyderabad, India in May 1999; received Bachelor of Engineering degree in Computer Science & Engineering from Osmania University, Hyderabad, Andhra Pradesh, India in May 2005. Completed the requirements for Master of Science degree with a major in Computer Science at Oklahoma State University in December 2007.

Professional Memberships:

1) Phi Kappa Phi

2) Golden Key International Honor Society

Name: Prathiba Reddy Nalabolu   Date of Degree: December, 2007

Institution: Oklahoma State University   Location: Stillwater, Oklahoma

Title of Study: DETECTING MALICIOUS CODE IN SENSOR NETWORK APPLICATIONS USING PETRI NETS

Pages in Study: 79   Candidate for the Degree of Master of Science

Major Field: Computer Science

Scope and Method of Study: Sensor Network reprogramming is essential and crucial for software updates or bug fixes on nodes. Reprogramming is fraught with many challenges. Attackers can easily insert malicious code during code propagation. Such code propagates in an epidemic way from one source to many, bringing down the whole network. The proposed model called Malicious Code Detector is based on Petri nets. The strategy to use Petri nets to model attacks is to understand the properties of the attacks. The proposed system model detects the malicious code at each node. The main components of the System are a Detection Tool and Library of attack Petri nets.

Findings and Conclusions: The Detection Tool's function is to transform code to Petri nets. The Detection Tool based on a Detection Algorithm that uses the attack Petri nets and the sensor application to detect the presence of corrupt code/attacks in it. Detecting such corrupt code can lead to shutting down of that particular node and most importantly preventing further propagation of this malicious code throughout the network.

ADVISER'S APPROVAL: Dr. JOHNSON P. THOMAS