Thesis

Security of Virtual Coordinate based Wireless Sensor Networks

Submitted by

Divyanka Bose

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2015

Master's Committee:

Advisor: Anura Jayasumana

Sudeep Pasricha
Christos Papadopoulos

Copyright by Divyanka Bose 2015

All Rights Reserved

Abstract

Security of Virtual Coordinate based Wireless Sensor Networks

Wireless Sensor Networks (WSNs) perform critical functions in many applications such as, military surveillance, rescue operations, detection of fires and heath care monitoring. In these applications, nodes in the network carry critical and sensitive data. Thus, WSNs are prone to various kinds of attacks that target different protocols and layers of the network. Also, most of the WSNs are placed remotely that makes it difficult to implement security measures after deployment. Thus, security of WSNs needs to be considered at the initial stage of system design. In many applications, the nodes are deployed randomly, and thus are unpredictable in terms of physical network topology. Virtual Coordinate (VC) based WSNs possess significant advantages over Geographical Coordinate (GC) based WSNs. This is because VCs negate the need for physical localization of nodes, which require costly techniques like GPS. The VCs of the nodes in the network are very important for basic functionalities such as routing and self-organization. However, security of VCs has not been extensively researched even though routing algorithms rely on the correctness of the VCs for proper functioning. VC based WSNs are susceptible to attacks resulting from malicious modification of VCs of individual nodes. While the impact of some such attacks is localized, others such as Coordinate Deflation and Wormholes (tunneling) can cause severe disruptions. This thesis proposes techniques for the detection and mitigation of attacks, which are aimed at the VC based WSNs.

We propose a novel approach where coordinate attacks are identified by detecting changes in the shape of the network, extracted using Topology Maps. A comprehensive solution for

detection of coordinate-based attacks on VC systems is presented that combines Beta Reputation System and a reputation based routing scheme. Latter ensures safe communication that bypasses malicious nodes during detection process. The Coordinate Deflation and Wormhole attacks are discussed and the effect and intensity of these attacks are addressed. Two methods are proposed and compared for the detection of attacks. In the first method, the topology distortion is rated using clusters identifiable by existing VCs, thus requiring low computation and communication overhead. A measure of topology distortion is presented. The existence of a trusted base station is needed for this method. In the second method, the detection is distributed and removes the need for a base station/server. We compare the advantages and disadvantages of the two methods, and discuss the scenarios in which these algorithms maybe implemented.

Simulation based evaluations demonstrate that both the schemes efficiently detects Deflation and Wormhole attacks. We choose a variety of dense networks with different topologies and deployment characteristics for evaluation. Networks with voids, representative of physical spaces with voids, as well as randomly deployed networks are considered, to ensure the correct operation and scalability of the algorithms. We show through simulations that the detection schemes can easily differentiate between the changes in the network due to node failures, e.g., caused by battery drain, from those due to an attack. Future sensor networks are predicted to be in the scale of millions of nodes. Thus, a need for security algorithms which can be scaled are highly desirable. We show in our simulations that the proposed detection schemes can be applied to networks of larger density successfully.

Acknowledgements

First and foremost, I would like to thank my advisor, Dr. Anura Jayasumana for his guidance and support throughout the course of this thesis. It has been a great privilege to work with and learn from him. I would also like to thank Dr. Sudeep Pasricha and Dr. Christos Papadopoulos for being part of my advisory committee.

I am extremely grateful for the love and support of my family away from home, Jatin, Hanisha, Arun, Chaitanya, Pranav. Thank you for being there as my friends and my critiques. Lastly I would like to thank my parents, Nabakumar and Shukla Bose, and my sister Priyanka Bose, for their unwavering faith and love for me.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Many emerging Wireless Sensor Networks (WSNs) involve collections of large number of nodes which in unison gather and route information among each other or to a base station for information extraction and decision making. Recent advancements in technologies like micro-electro-mechanical systems and wireless networks have enabled designers to build sensor nodes which require low power and are cheap, but also with multi-functional capabilities [1]. They are used for applications such as infrastructure monitoring, health monitoring, process control and precision agriculture [2] [3]. WSNs facilitate attractive features such as continuous or adaptive monitoring, event detection and location sensing, thus promising novel capabilities and resolutions not possible with prior technologies [1].

WSNs have specific properties which make them significantly different from Mobile Adhoc Networks (MANETs) [4]. WSNs are meant to be deployed in remote locations and involve little human interaction. WSNs are also more scalable than MANETs as WSNs can cover wide areas of spaces at a lower cost. The size of WSNs vary from hundreds of nodes to thousands of nodes. The sensor devices, which are the nodes of the network, are very low power devices. Hence, it is absolutely necessary to consider regular node deaths/failures during design of algorithms in WSNs. This is an unavoidable drawback when inexpensive sensor nodes are used, a requirement for large scale deployments. Thus the network designers also need to consider changing topology of Wireless Sensor Networks. In most applications of WSNs, the information given by single node is not of much importance. Data aggregation is needed from message packets collected from multiple nodes in the network for the information to be of use. Due to these reasons, designing of algorithms for WSNs

need to consider different constraints from those of MANETs. Designers need to consider scalability, low power availability, randomness in deployment and changing topology while designing algorithms for regular functions of WSNs. Previous research work on WSNs have been mostly focused on factors like fault tolerance, scalability, production costs and hardware constraints [1]. Also, WSNs are mostly deployed in regions that are remote and cannot be monitored consistently [5]. With requirement for data aggregation from multiple nodes, data integrity is of prime importance in WSNs. To make the network robust against security attacks, the defense mechanisms should be considered at design time itself.

Vast majority of algorithms essential for WSNs, such as those for self-organization, routing, topology control, boundary detection, and event detection depend on a coordinate system. Presently there are two prevalent coordinate systems for WSNs: 1) Geographic Coordinates System (GCS), and 2) Virtual Coordinate System (VCS). GCS characterizes its nodes using the physical coordinates of the nodes. These physical location coordinates are used in the routing algorithms such as GPSR [6]. GPSR uses the router's physical locations and makes routing decisions using greedy forwarding method.

On the other hand, Virtual Coordinates System(VCS) characterizes its nodes by the connectivity of the network [7] [8]. The hop distances of a node to a subset of nodes in the network are used to maintain the coordinates of the node. This subset of nodes in the network are called the *Anchor* or *Landmark* nodes. If there are M anchor nodes in the network, each node maintains a vector $(n_1, n_2.....n_M)$, which are the node's hop distances to the M anchors. $(n_1, n_2.....n_M)$ are called the *Virtual Coordinates*(VCs) of the node. Thus, the dimensionality of the network coordinate system is dependent on the value of M [9].

VCSs have few major advantages over GCSs. In GC based routing protocols [10], the routing state table in each node grows along with the scaling of the network. Individual nodes

in WSNs are severely resource constrained, making scaling of networks in such a scenario a major concern. However, in VC based routing protocols [8] [11], the routing is done based on the lowest hop count to the destination node. Thus, the forwarding is dependent only on the neighbor's VCs. Another significant advantage of VCS is that the exact physical locations of the nodes are not needed. This overcomes the usage of costly techniques like GPS for localization.

Many operations in VCS based networks, including coordinate updating, reconfiguration and sensing, depend on its nodes' own VCs as well as those of its neighbors. The VCs of the nodes in the network need to be constantly updated for the correct operation of routine procedures. This is done, by broadcasting the hop counts of the nodes to anchor nodes periodically. Therefore, any drastic changes in the coordinates of a single node in the network, affects a major portion of the network. Effects of such changes propagate quickly through the network. As a consequence, VC based WSNs are highly susceptible to various types of malicious attacks that modify the VCs. However, link failures and node deaths are a common occurrence in sensor networks due to the low power capacity of the nodes. Thus, changes in VCs are unavoidable in WSNs. It is very important for attack detection schemes to recognize these legitimate changes in VCs due to environmental causes.

1.1. Problem Statement

There has been significant research on VC based routing algorithms [12] [7] [13] [8]. These research works indicate that VC based WSNs are much more efficient, scalable and cost effective than GC based WSNs. However, all the VC based algorithms [8] [14] [13] [11] rely on the correctness of the VCs of the nodes in the network. Moreover, many applications

in which WSNs are employed require the network to be deployed in adversarial locations. Thus, continuous supervision is not possible for WSNs.

An attacker aiming to disrupt the routine procedures of Wireless Sensor Networks(WSNs) can easily do so by changing the VCs of nodes. These attacks are highly difficult to detect as VC changes are pretty common in WSNs due to environmental factors and the mobility of nodes. A VC based network updates its Virtual Coordinates of nodes either periodically or during a change in the network. Each node updates its VCs by seeing its neighbors' VCs. Thus, a change in the VCs of a single node propagates rapidly through the entire network. In this section, we discuss two attacks, namely, Coordinate Deflation and Wormhole attacks in detail. In Coordinate Deflation attack, the attacker intentionally modifies the VCs of the attacked nodes. In Wormhole attack, the attacker creates a tunnel between two far away nodes. This changes the VCs in the neighborhood of the attacked nodes, because VCs follow shorter hop count among all the neighbor. Here, the tunnel provides the shorter hop count.

1.1.1. Coordinate Deflation. A Coordinate Deflation attack [15] is caused by taking a legitimate node under the control of an attacker, and the attacker falsely propagating lower VC values for the attacked node. The neighbors of this node then determine incorrect VCs for themselves due to lower perceived distance through the attacked node. Coordinate Inflation, which is inflation of the VCs of the attacked node, is not as harmful as a the Deflation attack. This is because, the VCs updating procedures follow shorter hop count. So, in case of Inflation the neighbors simply ignore higher hop counts. Deflation attack can, for example, result in a "blackhole" in the attacked node. Due to the shorter hop count broadcasted by the attacked node, all the traffic is attracted towards the attacked node. Thus, Coordinate Deflation attacks cause major routing failures and makes it very difficult to detect the attack on the network.

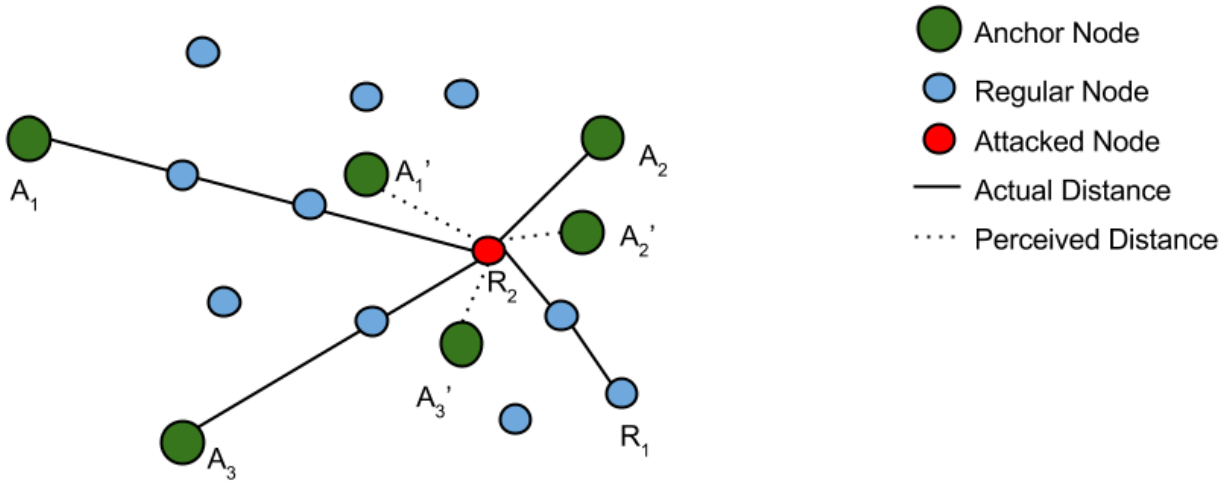


FIGURE 1.1. Coordinate Deflation Attack

Consider the network in Figure 1.1. Note that we select this trivial example for explanation purposes only, but networks may have hundreds to thousands of nodes with few percent of them acting as anchors. As seen in the network there are three anchor nodes A_1, A_2 and A_3. Before an attack, the VC of node R_2 is [3, 1, 2] and VC of node R_1 is [5, 3, 4]. Mounting a coordinate deflation attack on node R_2, its VC is modified to [1, 1, 1] by the attacker. The anchor nodes A_1, A_2, and A_3 seem much closer to the attacked node at positions A_1', A_2', and A_3' as shown in the same figure. Because of this, VC of node R_1 is changed to [3, 3, 3]. Hence for the node R_1, which was originally 5 hops away from anchor node A_1, now appears at a hop distance of 3. During routing from R_1 to A_1, the packet will assume that it has reached A_1 in 3 hops but the packet actually does not reach A_1.

An attack on a single node can affect up to 50% of nodes in the network as we shall show in subsequent sections. Multiple attacked nodes can cause the entire network to fail. This is one of the most hard-hitting and widespread attacks on a VC based WSN.

1.1.2. WORMHOLE. Another destructive attack on Virtual Coordinates is the Wormhole/Tunneling Attack [15]. WSNs use radio channels for communication. Thus, those

nodes which are within a node's radio range, are considered as it's neighbors with a hop distance of 1 from the node. A Wormhole attacker generates strong radio/physical channel between two nodes that are actually not near to each other, thus creating a tunnel. Alternatively it may provide the illusion of a tunnel without actually using a radio channel between nodes.

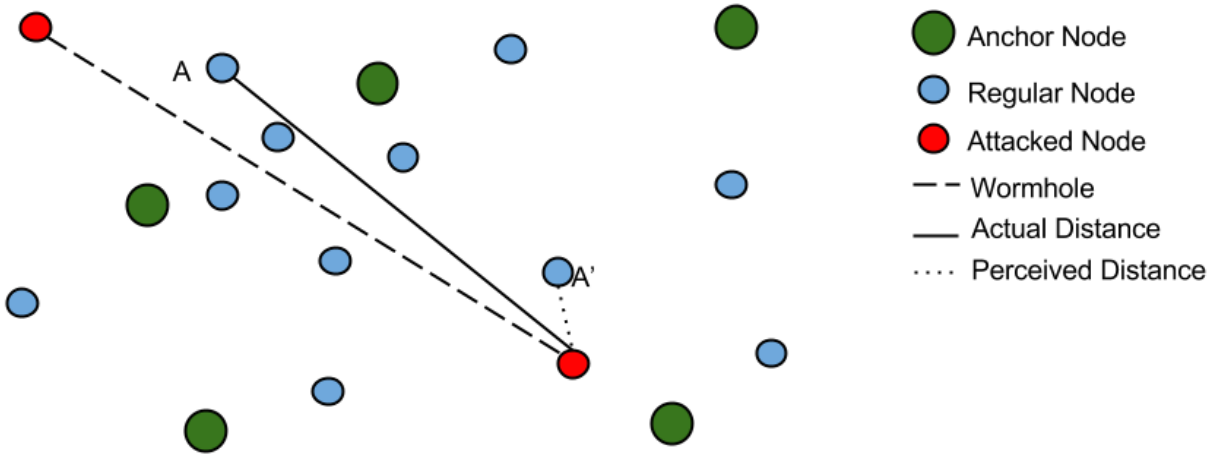


FIGURE 1.2. Wormhole Attack

Figure 1.2 represents a Wormhole attack. Due to the attack, the node A is perceived at position A'. This generates a one hop distance along the tunnel. As a result, the neighbors of A believe that they have a shorter hop count through the Wormhole. Thus, the VCs of the neighboring nodes are affected accordingly. Wormhole attack may be active or passive. Attack is said to be Passive when the attacker merely creates a tunnel between two nodes and the data passes through the Wormhole instead of the regular path. Active Wormhole attack is when the attacker creates a wormhole and gets all the packets and does not forward them ahead or misroutes them.

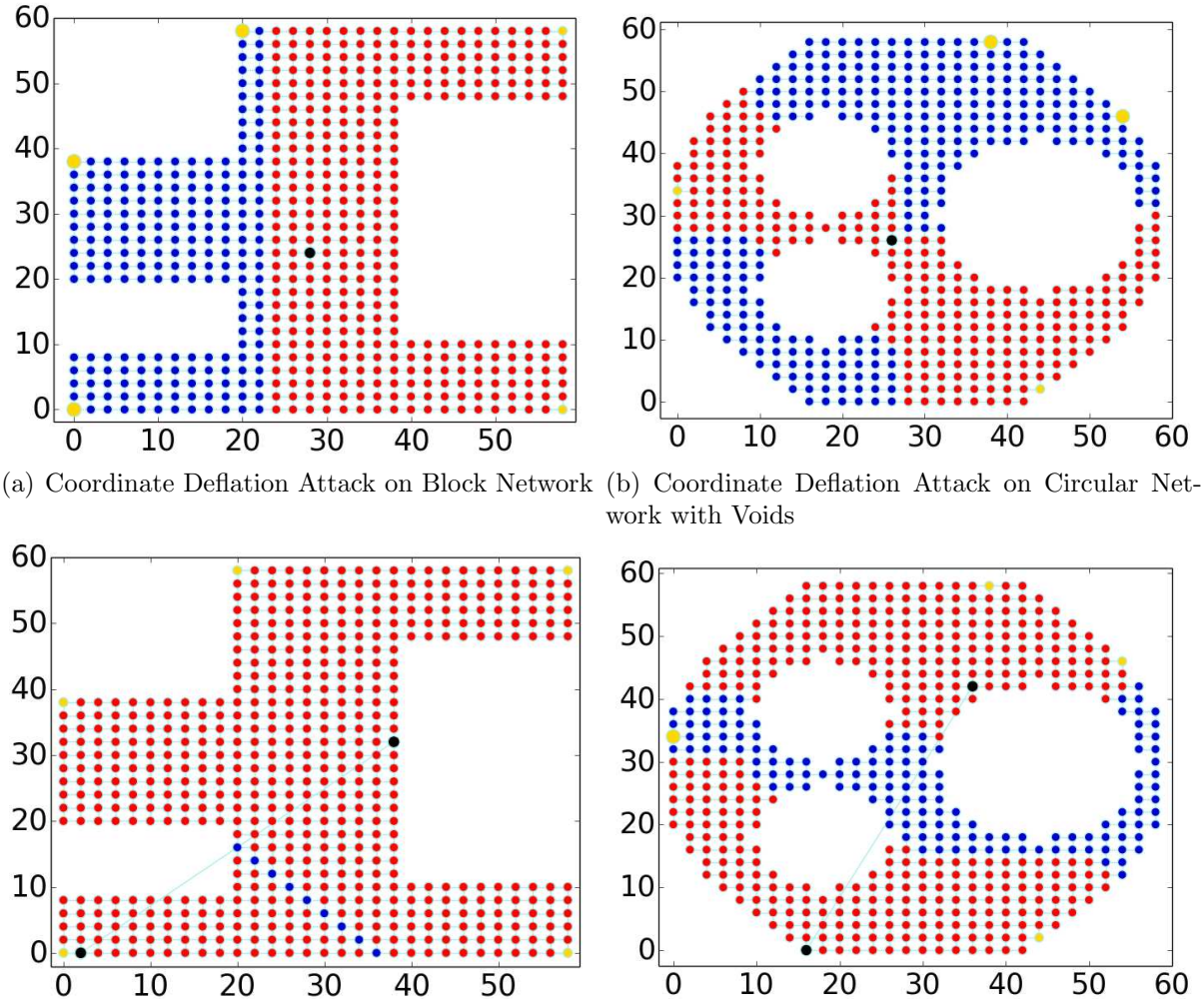
In case of using a separate radio channel between two nodes, a Passive attack does not sound harmful to the network. Instead it appears to be beneficial to the network as many

nodes are bypassed by the tunnel and don't have to do routing functions. However, the tunnel is an insecure location. Sensitive data can be listened on by the attackers. Thus, data integrity is at risk with such passive attacks.

Active attacks cause VCs of the other nodes in the vicinity to get deflated causing routing failures. Also, more packets are routed through the Wormhole as they provide a shorter hop count, but such packets may be dropped by the malicious nodes providing the Wormhole.

As we show in further sections that basing detection schemes just on VC may cause non detection of attacks. Also, changes in network due to environmental factors may be considered as an attack if only changes in VCs are considered. Both the Deflation and Wormhole attacks bring about different changes in the network. While deflation attack pull all the traffic towards itself, Wormhole just creates a tunnel between two distant nodes. The changes in the VC domain are different.

Figure 1.3 shows the affected nodes due to an attack on two example networks. The nodes in black are the attacked nodes and the nodes in red are the nodes whose VCs have changed due to the attack. The nodes in blue are the unaffected nodes. In Figure 1.3(a), each VC is of length 4 as the number of anchor nodes are 4. Two of those VCs of the attacked node were modified and more than 50% of the nodes are affected. Figure 1.3(c) shows the affected nodes on the same network due to Wormhole. As we see, the effect of the attacks are widespread and travel fast through the network. An efficient detection scheme that can detect an attack and mitigate it, is an essential part of the designing of VC based WSNs. The detection scheme needs to be able to differentiate between an attack and node deaths/failures.



(a) Coordinate Deflation Attack on Block Network

(b) Coordinate Deflation Attack on Circular Network with Voids

(c) Wormhole Attack on Block Network

(d) Wormhole Attack on Circular Network with Voids

FIGURE 1.3. Nodes affected due to an attack

1.2. CONTRIBUTION

Attack detection schemes are often hindered by the compromised nodes themselves. Obtaining information for detection is likely to be compromised by the so far unknown attack itself. Thus, first we present a reputation scheme for nodes, based on the Beta Reputation System [16], triggered by change of coordinates of a neighbor node. A trust based routing method for VC based WSNs is proposed. The reputation system is dependent on each node

assigning reputations to its neighbors based on the changes that the node sees in the neighbor's VCs. Thus, the attacker cannot propagate false reputation values for itself. Higher the reputation of a neighbor, higher is the node's trust to send high reliability packets through that neighbor. The reputation is maintained and calculated by each and every node in the network, for each of its neighbors, making the reputation scheme completely distributed.

Secondly, we introduce two novel detection techniques to detect an attack by using Topology Preserving Maps (TPM)s [17]. These detection techniques are Centralized Detection Technique (CDT) and Distributed Detection Technique (DDT). TPMs are capable of capturing layout and topology features from VCs. The changes that occur in the topology due to an attack is captured and used in the detection schemes. CDT uses the centre of gravity of the network to measure the distortion in the topology of the network. DDT employs the neighbors of the attacked nodes to make a decision about the change in VCs by using the topological distances. Simulation results clearly show the effectiveness of these schemes. We demonstrate that the detection in both the schemes are highly efficient in differentiating changes in the network due to attacks from those due to changes such as fading or node shutdown because of battery drain causing missing nodes, that are expected routinely in sensor networks. The detection algorithms are shown to be scalable, a necessity for large-scale Wireless Sensor Networks.

1.3. Outline

This dissertation is arranged as follows. Chapter 2 explains the Virtual Coordinate System of the Wireless Sensor Networks. We also discuss various attacks on WSNs and the previous work that has been done to tackle these attacks. Chapter 3 gives a background information on The Beta Reputation System. We present the reputation updating and

reputation based routing algorithm in this chapter. In Chapter 4, we study the effect of attacks and node deaths on Topology Preserving Maps. This chapter provides the basis for our detection algorithms. We present the algorithms for our detection algorithms in Chapter 5. We provide results using simulation based experiments in Chapter 6. We conclude our work and propose future work in Chapter 7.

CHAPTER 2

Literature Review

WSNs have undergone a lot of research due to the following reasons:

(1) Scalability of network

(2) Lower cost and power requirements

(3) Ease of access to remote locations

Currently there are two prevalent types of coordinate systems for WSNs. They are 1) Geographical Coordinate System (GCS) and 2) Virtual Coordinate System (VCS). VC based WSNs are proven to be highly economical over GC based WSNs. WSNs based on GC require exact physical locations of the nodes. This require costly techniques like GPS [6] [11] which are not feasible for WSNs. VC based systems overcome this difficulty by using hop counts to a subset of nodes called the *Anchor Nodes*. We shall discuss the VCS [8] [7] in detail in this section.

Consider the network in Figure 2.1. Each node maintains a set coordinates, which are the node's hop distance to the chosen anchor nodes. The set of anchor nodes in this network are [0, 5, 12, 17]. The node 7 is at a hop count of [2, 5, 2, 5] from each of the anchor nodes respectively. This hop count is broadcasted to the neighbors so each node's VC is updated. Node 8, which is a neighbor to node 7, updates its VCs as [3, 6, 3, 6]. Since anchors 5 and 17 are at a shorter hop count through neighbors node 2 and node 14. Thus, node 8 updates its VCs to [3, 4, 3, 4]. Thus, VC updates always follow the shorter hop count to the anchor nodes.

Routing algorithms for VC based WSNs [8] [7] also follow the shortest hop distance to the destination. Thus, the need for actual physical location of the destination node is removed.

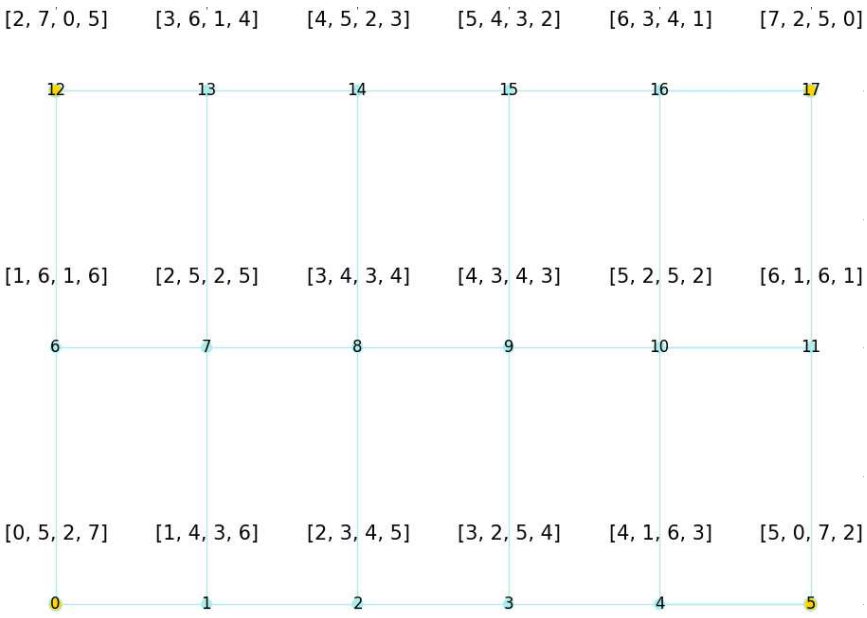


FIGURE 2.1. Example Network To explain VCS

[9] and [8] propose solutions to loop problems and void avoidance which may arise in VC domain.

WSNs are prone to attacks due to their carrying of sensitive data, ease of accessibility to sensor nodes in the network and lack of information about deployment of the network. In this chapter, we shall discuss the common attacks on Wireless Networks. We shall also discuss the prior work that has been done to tackle these attacks.

2.1. TYPES OF ATTACKS

This section discusses the attacks on Wireless Sensor Networks. This discussion is not limited to the attacks on Virtual Coordinate based WSNs only but for WSNs in general.

2.1.1. DENIAL OF SERVICE ATTACK. Denial of Service(DoS) attack [18] [19] is an attack aimed at disrupting/jamming the network, so that the network's capacity to perform regular functions is eliminated. DoS attack can be mounted by unintentional failure of nodes, or

by exhausting the memory or power of attacked nodes. In WSNs, this attack may aim the physical layer by jamming, or at network layer by malicious flooding or desynchronization.

2.1.2. Sybil Attack. In Sybil attack [19] [20], an attacked node forges multiple identities. Most of the WSNs' regular duties involve distributed functioning, that depends on data collected by multiple nodes and not just a single node. Sybil attack tries to degrade the performance of the network by compromising the integrity of the data or the node that the distributed algorithm attempts to achieve. Sybil attacks can be mounted by attacking the distributed storage, the routing, or data aggregation. Also, it can be imbibed by false voting in networks which depends on algorithms that use votes for resource allocation.

2.1.3. Bad Mouthing Attack. In bad mouthing attack [21], attackers spread bad reputations for honest nodes. Many systems use reputation system, which we shall discuss further, for security of WSNs. Due to bad mouthing attacks, good nodes get bad trust values. This causes network disruption. Many bad mouthing attacks are oscillating, i.e. they alternate good and bad reputations for nodes. Since building trust is a dynamic process, trust may keep varying as the node behavior oscillates.

2.1.4. Blackhole/Sinkhole Attack. In Blackhole attack [15] [19], the attacked node tries to attract all the traffic towards itself. The nodes choose to send data packets through the shortest route to the destination. The attacker attracts these data packets by broadcasting a shorter route to the destination. Thus, Blackhole attack causes routing failures and also compromises the integrity of the data.

2.1.5. Wormhole Attack. A Wormhole attacker [22] [15] [19] builds a tunnel between two nodes in a network. If, due to the presence of this tunnel, the transmission of packet gets a shorter route, any node will prefer to use the tunnel. Thus, the attacker can overhear

the data being sent over this tunnel channel. There are no behavioral changes in a node due to the presence of a Wormhole which is otherwise present in malicious nodes in other types of attacks. Thus, this attack is particularly hard to detect.

2.2. Security against Attacks

In this section we shall discuss the prior work that has been done to tackle the attacks mentioned in Section 2.1.

2.2.1. Cryptography. [23] proposes to tackle the attacks on the messages sent between sensor nodes by encrypting the messages using public key cryptography. Prior work on securing WSNs using cryptography are mainly divided into 3 types, namely, *trusted server* scheme, *self enforcing* scheme and *key pre-distribution* scheme [23]. In *trusted server* scheme, a server/base station is used to provide key agreements between nodes. *Self enforcing* scheme uses public key certificates to generate asymmetric key agreements between nodes. *Key pre-distribution* scheme depends on distribution of keys to the nodes before deployment. The mentioned methods using cryptography are not suitable in WSNs due to lack of a trustworthy server, incomplete knowledge of the placement of the nodes or the limited power and memory resources of the sensor nodes. [23] proposes a security method using *key pre-distribution* scheme. However, they make use of the model node deployment. The deployment knowledge is modeled using probability distribution functions.

2.2.2. Intrusion Detection System. Intrusion Detection Systems (IDS) [24] typically study the behavior of the nodes during routine functions to detect an anomaly. In [24], the authors divide the detection of an attack by an IDS into three phases. First, the system employs monitor nodes to gather data/messages in promiscuous mode to be analyzed later. In second phase, a specific set of anomaly detection rules are then applied to these gathered

messages. If a dataset of messages fails to comply with a single rule, the failure counter is incremented. Once all the message datasets are filtered using the set of rules in second phase, an attack is raised if the failure counter value is greater than a preset threshold value. This is the third and final phase of the IDS.

2.2.3. Reputation based Detection. In reputation based detection schemes like Reputation-based Framework for Sensor Networks (RFSN)[25], nodes develop and maintain reputation rating for other nodes in the neighborhood. In [25], there are two building blocks of RFSN, *Watchdog* and *Reputation*. The block *Watchdog* is responsible for monitoring the activities of the sensor nodes in the network. The *Reputation* block is used for maintaining a database of the reputation of the nodes. A Bayesian framework is used to provide output *trust* metric for nodes based on positive and negative feedback. A node maintains good reputation for another node if it is cooperative in performing an informational data exchange. Thus, the scheme in [25] addresses the case where malicious nodes do not cooperate. However, with Coordinate attacks considered in this paper, the attacked nodes actually cooperate or gives the appearance of cooperation and thus a coordinate change based reputation scheme is required.

2.2.4. Detection against Wormhole Attack. [22] introduces *leashes* to detect Wormhole attacks. Two types of leashes namely *geographical leash* and *temporal leash* are introduced. A *geographical leash* is used to ensure if the distance traveled by a packet is within a reasonable distance between the source and destination. A *temporal leash* restricts the maximum travel distance by keeping an upper bound on the packet's lifetime in transit. A Wormhole is detected if the receiver node receives a packet which has traveled more than the leash has allowed.

2.2.5. Attack Detection in VC based WSNs. [15] discusses two attack detection methods against Coordinate Deflation attacks in VC based WSNs. The first method uses the Wilcoxon signed rank test to periodically compare the current VC of the nodes with stored reference VC. For this method, a known period of time when the network is not under attack is considered. During this benign period of time, the reference VCs are generated. The second method considers a known statistical model of the network topology. The network model is determined using the method of deployment. The Monte Carlo method is used to estimate the reference hop counts of the nodes from the statistical network topology model. Once the reference hop counts are generated, the current hop counts are compared periodically to detect attacks.

2.3. Summary

Previous work on security of WSNs is focused on detecting attacks by studying behavior of the nodes in the neighborhood. However, techniques like cryptography are unsuitable for WSNs as they are low power devices and cannot handle the overhead computations for public key cryptography. Leashes can be modified by the attacker itself to be shown as honest. Meager amount of research has been done for detecting attacks in Virtual Coordinate based WSNs using the Virtual Coordinates. The VCs have a good amount of inherent connectivity information which can be harnessed to detect attacks. In the further chapters, we present the detection techniques and the motivation behind them.

CHAPTER 3

The Reputation System for VC based WSNs

In this section, firstly we shall present background information about the Beta Reputation System [16]. Reputation Systems are widely used in e-commerce trade to maintain good behavior in systems. A reputation system collects positive and negative feedbacks of systems and uses the past transactions to predict reliability in future transactions. Secondly, we shall present the reputation system using Beta Reputation System for VC based WSNs. Thirdly, we present the routing in VC based WSNs using the reputation rating of neighbors.

3.1. The Beta Reputation System

The Beta Reputation System [16] has a strong basis in Statistics and can be easily implemented in distributed networks. We use this system in VC based WSNs to maintain the reputation of nodes by its neighbors. Beta Reputation System is based on the Beta probability density function indexed by two parameters α and β.

The beta distribution $f(p|\alpha,\beta)$ is expressed as

$$f(p|\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}, \quad \text{where } 0 \leq p \leq 1, \alpha, \beta > 0 \tag{3.1}$$

p is a probability variable. $f(p|\alpha,\beta)$ represents the second order probability for a given p. p being continuous and $p \in [0,1]$, $f(p|\alpha,\beta)$ is a very small value. Therefore the expectation value of p is used to define a reputation rating. The probability expectation of Equation 3.1

is expressed as

$$E(p) = \frac{\alpha}{\alpha + \beta} \tag{3.2}$$

Consider an entity x which gives a reputation value to another entity t. If the integer value of positive feedbacks given by x to t are r_t^x and the integer value of negative feedbacks given by x to t are s_t^x, then Equation 3.2 can be expressed as

$$Rep(r_t^x, s_t^x) = \frac{r_t^x}{r_t^x + s_t^x} \tag{3.3}$$

The expectation value in Equation 3.3 is called the Reputation Rating and is denoted as $Rep(r_t^x, s_t^x)$ [16]. The Reputation Rating value gives a measure of how much an agent can be trusted and how it is expected to behave in future transactions. Equation 3.3 gives the Reputation Rating value in between 0 and 1.

3.2. Algorithm for Reputation System

In this section, we present the algorithm based on The Beta Reputation System [16] to characterize the reputation of the nodes in the network. During initial setup of network, each node maintains a database of its neighbors VCs along-with its own. Any changes in the Virtual Coordinates of any node in the network is observed by its neighbors. We use this change in neighbor's VCs to build our reputation system. Each node maintains a reputation value for each of its neighbors. The routing of packets which need high reliability is done using these reputation values by forwarding the packets only to trusted neighbors. The Beta Reputation System measures the reliability of the neighbor by taking an integer value of positive and negative feedbacks by the node.

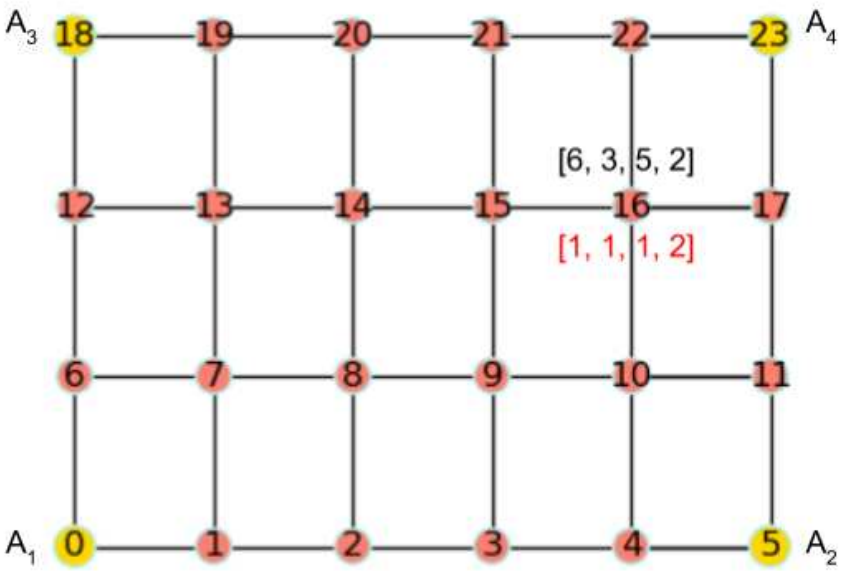


FIGURE 3.1. Example Network to explain Beta Reputation

We shall elaborate the Reputation System by using an example shown in Figure 3.1. At initial setup, each node initializes the reputation of neighbor nodes as 1. Consider node 15 which has its neighbor reputation as (21:1, 14:1, 9:1, 16:1), which is ($Neighbor$: $NeighborReputation$). We simulate the node 16 to be attacked. The original Virtual Coordinates of the node 16 are [6, 3, 5, 2]. We mount a Coordinate Deflation attack on node 16 which broadcasts the new VCs of node 16 as [1, 1, 1, 2].

We now describe the calculation of the positive feedback value r and the negative feedback value s as explained in Section 3.1.

3.3. Calculation of r and s values

Let node j be a neighbor of node i. If node j undergoes changes in its VC, its reputation is re-evaluated by node i. Using the original and changed VCs of node j, node i calculates the positive and negative feedback for the node j. The procedure to derive these values of positive feedback r_j^i and negative feedback s_j^i is shown in Algorithm 1.

Algorithm 1 Algorithm for calculation of r and s

Let the number of anchors be M

Original VC of node j : $[x_0, x_1, x_2.....x_M]$

Changed VC of node j : $[y_0, y_1, y_2.....y_M]$

$r_j^i = 0$

$s_j^i = 0$

for $k = 0$ to M **do**

    if $x_k = y_k$ **then** $r_j^i = r_j^i + 1$

    else $s_j^i = s_j^i + diff(x_k, y_k)$

    end if

end for

return r_j^i, s_j^i

Referring back to the network in Figure 3.1, node 15 receives the changed VC of node 16 from original value [6, 3, 5, 2] to deflated value [1, 1, 1, 2]. The r_{16}^{15} value given by node 15 to 16 is 1, since only 1 element of VC has not changed. The s_{16}^{15} value is 11 which is the sum of the difference of the VC elements, $(6-1)+(3-1)+(5-1)$. Further, the Reputation Rating calculation is presented.

3.4. Reputation Rating

The Reputation Rating of node j by node i is calculated using Equation 3.3. This equation uses positive feedback r_j^i and negative feedback s_j^i as inputs.

For the network in Figure 3.1, the Reputation Rating Rep_{16}^{15} is $\frac{1}{1+11} = 0.08$. The neighbor reputation database of node 15 is now updated to (21:1, 14:1, 9:1, 16:0.08). The VC of node 15 are updated as required due to the changes in VC of node 16. The updated VC of node

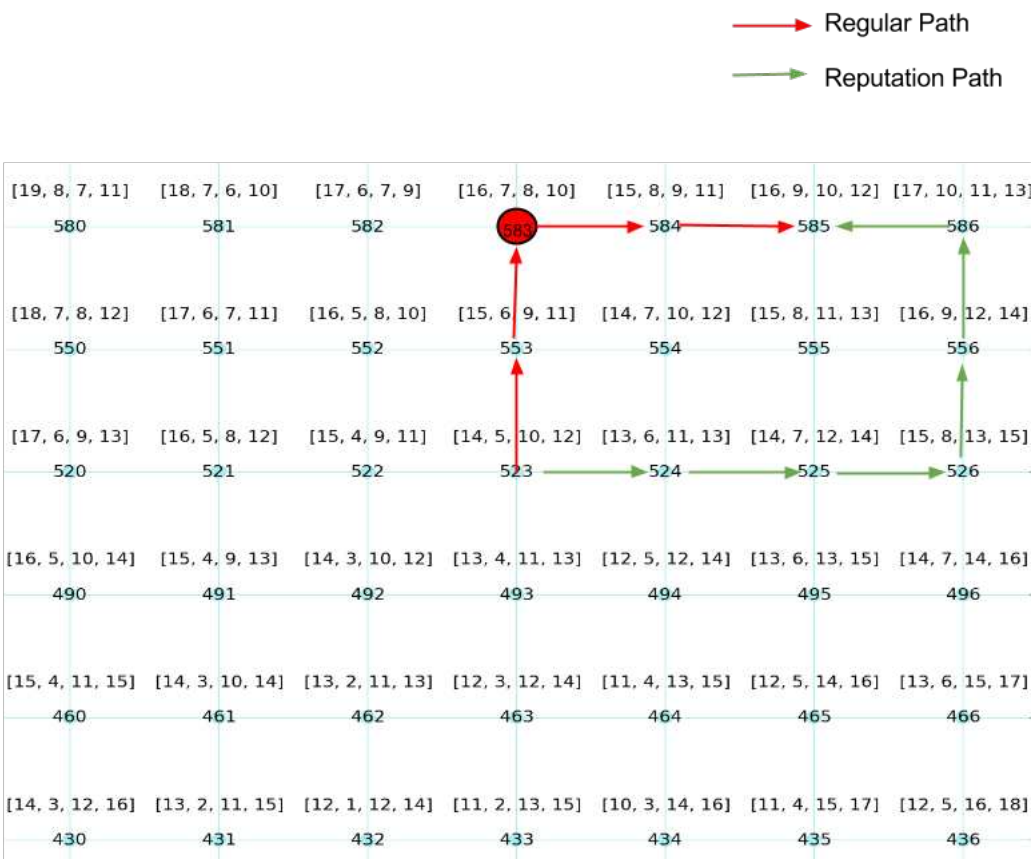


FIGURE 3.2. Example to explain Reputation based Routing

15 further propagates to others and similarly each node's neighbor reputation in the entire network gets updated. This reputation system is used to ensure that during communication of high reliability packets, node 15 sends its data via a neighbor node other than 16, as it has the lowest reputation.

3.5. REPUTATION BASED ROUTING

In this section, the routing of packets using the reputation values are discussed. Let us consider an example portion of a bigger network in Figure 3.2. In this network, node 583 is one of the attacked nodes in the network. Table 3.1 shows some sample reputation values for few of the neighborhood nodes. Consider that we need to route a high reliability packet

TABLE 3.1. Reputation Values for Neighborhood Nodes

Node Number	**Neighbor Reputation Set**
523	{522:1, 553:0.1, 524:0.9, 493: 1}
524	{523:0.4, 554:0.2, 525: 1, 494:1}
525	{524:0.5, 555:0.4, 526:1, 495:1}
526	{525:0.5, 556:0.8, 496: 1}
556	{526:0.5, 555:0.4, 586:1}
586	{585:0.7, 556:0.7}

from node 523 to node 585. Considering only the shortest hop count, a data packet would follow the path 523→ 553→ 583→ 584→ 585.

Now we present how the packet will travel considering the reputation values. Node 523 checks its neighbor reputation dataset and verifies whether the next node, 553 is the closest node to the destination node. Since node 553 has very low reputation value, the packet is sent to node 524, which has the next shortest hop count to the destination. Node 524 verifies the same and sends the packet to node 525. Thus, the path followed by the packet will be 523→ 524→ 525→ 526→ 556→ 586→ 585. The node sends the data to the next best neighbor with shortest hop count but a neighbor with high reliability. The number of hops required in the path without reputation check was 4 and in the path with reputation check is 5. But the data has less probability to be tampered. Thus, detection packets can be sent using reputation routing to avoid modification from the attacker. The algorithm for reputation routing is shown in Algorithm 2

Algorithm 2 Algorithm for Reputation Routing

1: Let the number of neighbors for a node be n

2: Let the neighbors be $N_1, N_2....N_n$

3: Let their reputation values be $R_1, R_2....R_n$

4: The node's reputation dataset is $N_1 : R_1, N_2 : R_2....N_n : R_n$

5: Find the neighbor closest to destination. Let the neighbor be N_m.

6: **if** $R_m >$Acceptable Reputation Value **then** Send Packet to the neighbor N_m

7: **else** Search for next closest neighbor to the destination

8: **end if**

9: Go to Step 5

CHAPTER 4

Effects of VC based Attacks on TPMs

In this chapter, we shall see how an attack affects the topological map of the network. We generate the topological map using TPM [17]. Firstly, we discuss the generation of Topology Preserving Maps(TPMs) from the VCs. A TPM [17] is, to a very high degree, homeomorphic to the physical layout of the network. Physical voids and connectivity of the nodes which are transparent in Virtual Coordinate domain are clearly seen in the Topological Coordinate domain.

4.1. Topology Preserving Maps

In Virtual Coordinate System, the number of anchor nodes represent the dimensionality of the coordinate system. TPMs provide layout maps that capture information about physical layout, shapes and features of 2-D and 3-D sensor networks. TPMs are derived from the VCs via a partial Singular Value Decomposition(SVD) of the nodes [17]. An alternate approach using Directional VCs is described in [26]. Consider a network with N nodes and M anchor nodes. A matrix P is a NxM matrix with the i^{th} row being the *M-length* VC of node *i*. The Singular Value Decomposition is applied on matrix P that generates *U, S* and V matrices as follows,

$$P = U.S.V^T \tag{4.1}$$

Using the unitary property of V, the projection of P onto V is

$$P_{SVD} = P.V \tag{4.2}$$

$V^{(j)}$ is denoted as the j^{th} basis column of V. Let the VC of node i be denoted as $[i_0, i_1...i_m]$, where $i_0, i_1...i_m$ correspond to the distance to each of the M anchors. Then the Topological Coordinates of node i, (x_T^i, y_T^i) is described as

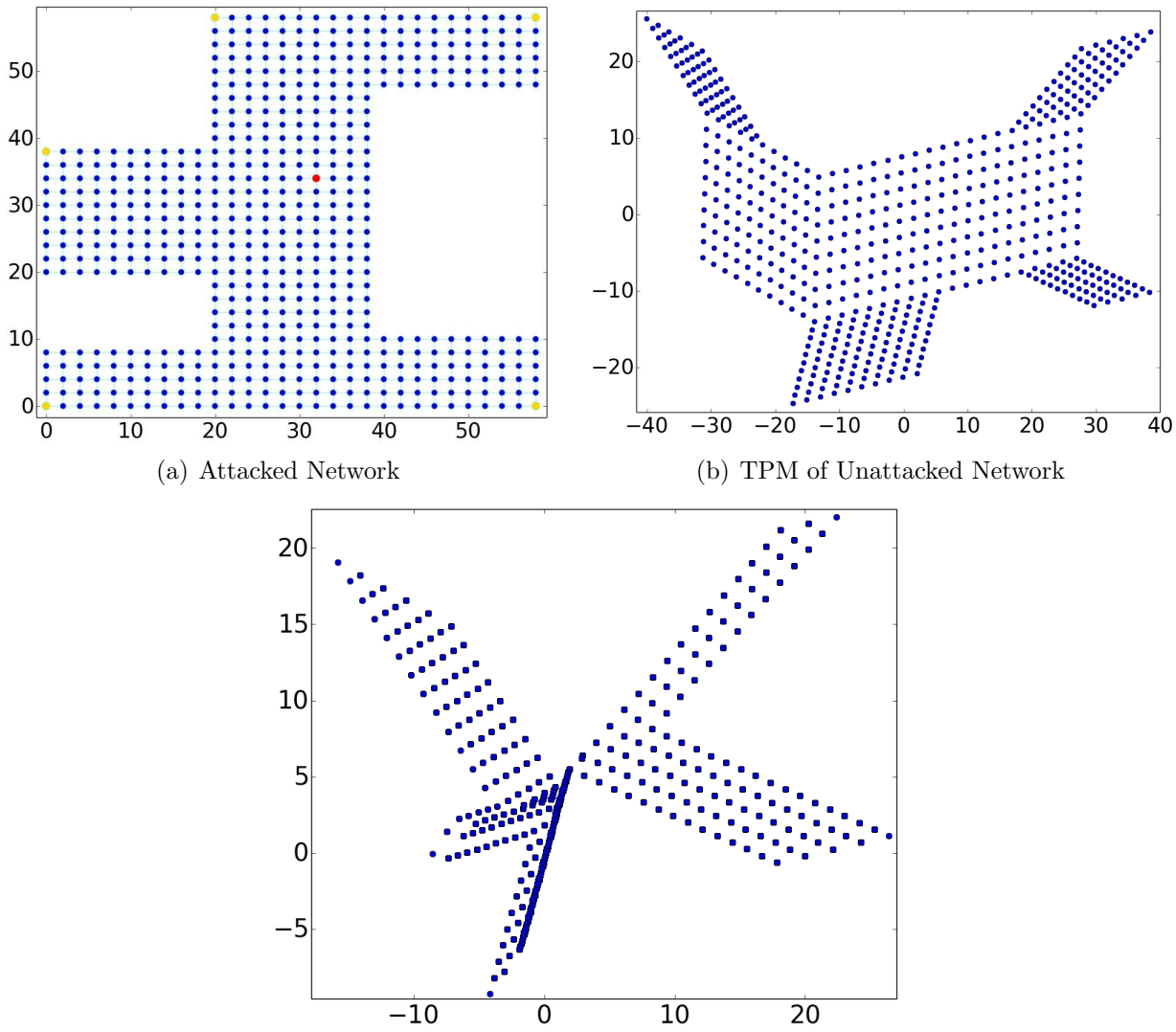
$$(x_T^i, y_T^i) = ([i_0, i_1...i_m].V^{(2)}, [i_0, i_1...i_m].V^{(3)}) \tag{4.3}$$

A computationally efficient approach that uses only a small subset of rows of P is described in [17]. The Topology Preserving Maps regenerate the original topological features of the network, such as voids and edges. It also recovers directional relationships that are lost in the VC representation.

4.2. Topological Changes

In this section, we compare the topological changes that occur in the network due to a Coordinate Deflation attack, Wormhole attack and node deaths. The topology maps are generated using the algorithm mentioned in Section 4.1. The anchor selection for the network is done using the Extreme Node Search(ENS) Algorithm [26]. The ENS algorithm attempts to choose the anchor nodes which are farthest apart and those which are the corner nodes of the network.

Figure 4.1(a) shows an odd shaped Block network of 550 nodes with 5 anchor nodes. Figure 4.1(b) shows the corresponding topological map of the unattacked network. This network is mounted with a Coordinate Deflation attack. The attacked node is shown in red and its original VC of [33, 30, 18, 18, 25] is deflated to [3, 4, 5, 6, 2]. After the attack, the topological map is recreated for the changed Virtual Coordinates. The resultant map is shown in Figure 4.1(c). It is noted that Coordinate Deflation causes the topological map to converge towards the attacked node. Thus, a blackhole is created at the attacked node.



(a) Attacked Network

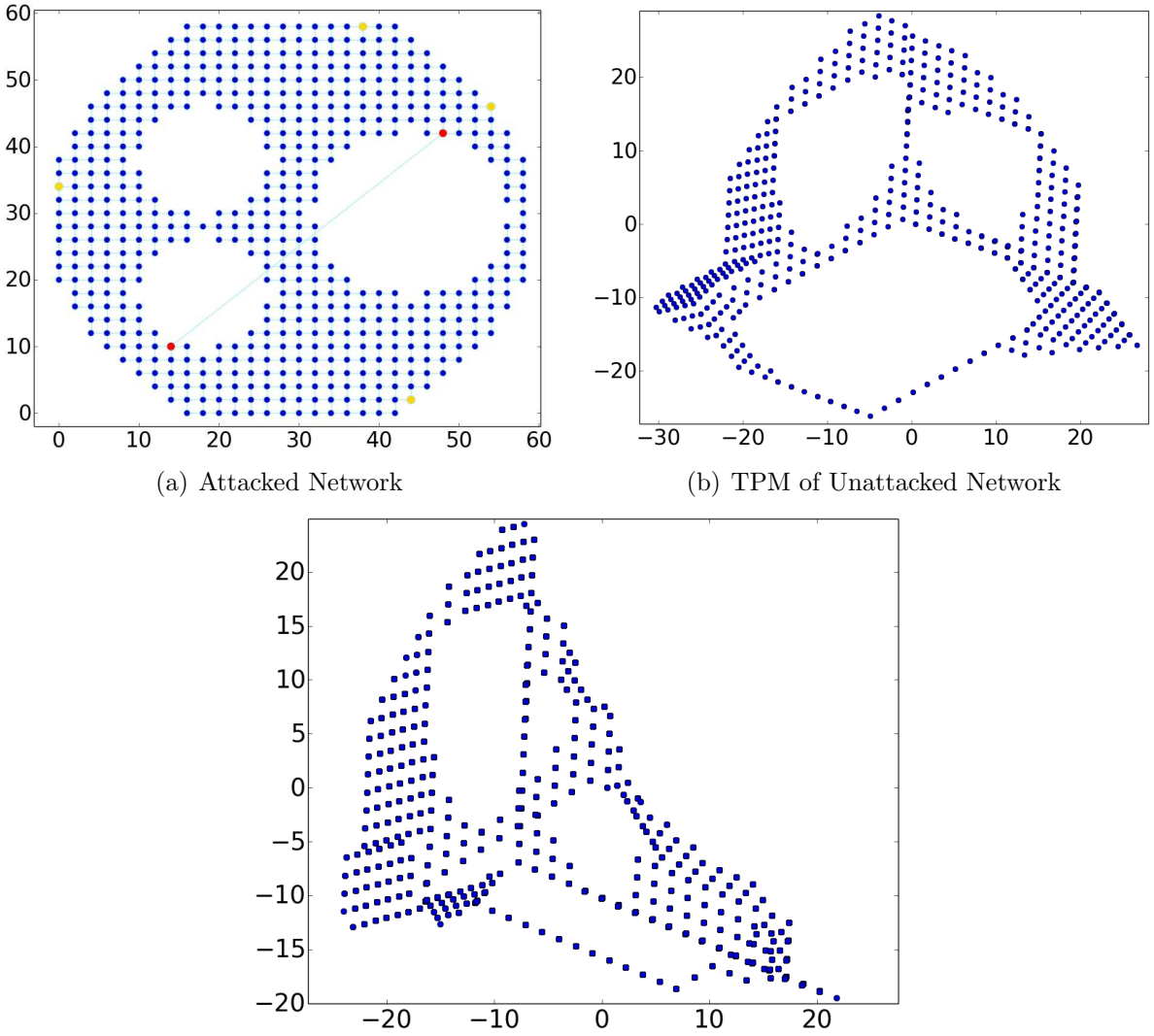
(b) TPM of Unattacked Network

(c) TPM of Attacked Network

FIGURE 4.1. Block Network mounted with Coordinate Deflation Attack

Another network is shown in Figure 4.2(a) which is a Circular Network with voids and has 4 anchor nodes selected by the ENS algorithm. This network is attacked by a Wormhole between the two highlighted nodes in red. Figure 4.2(b) shows the topological map of the unattacked network. Figure 4.2(c) shows the topological map due to the changes in the VC post the Wormhole attack. Wormhole causes the topological map to fold over.

The drastic changes in the topological maps of the pre-attacked and post-attacked networks are visually seen in Figure 4.1 and Figure 4.2. Many changes in the Virtual Coordinates



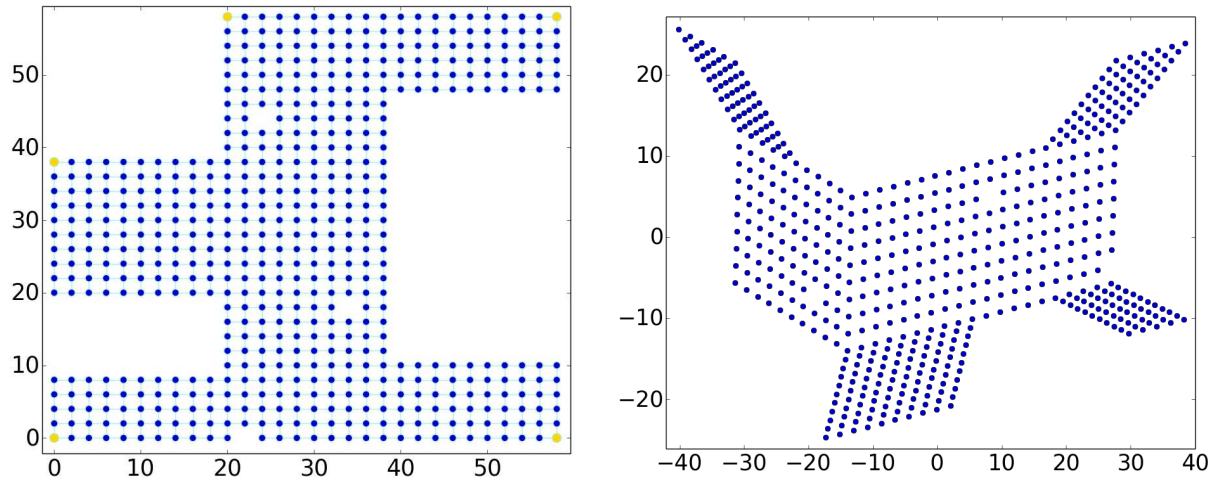
(a) Attacked Network

(b) TPM of Unattacked Network

(c) TPM of Attacked Network

FIGURE 4.2. Circular Network with Voids mounted with Wormhole attack

are due to legitimate changes. WSNs nodes are low power devices which can lead to unpredictable node deaths. These node deaths may cause changes in the Virtual Coordinates of its neighbors. However in dense networks, which we are using for our simulations, the percentage of node's affected due to a node death ranges between 0 to 5%. Figure 4.3(a) has 3 node deaths in the Block Network shown in Figure 4.1(a). The corresponding topological



(a) VC Network after Node Death

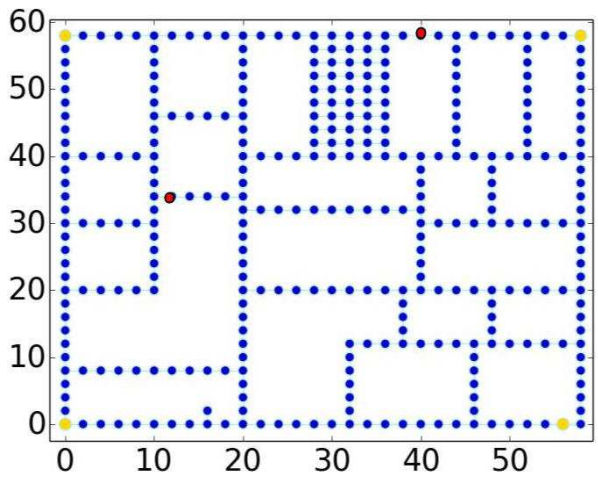
(b) TPM after Node Death

FIGURE 4.3. Odd-Shaped Network and its topology map in presence of node deaths

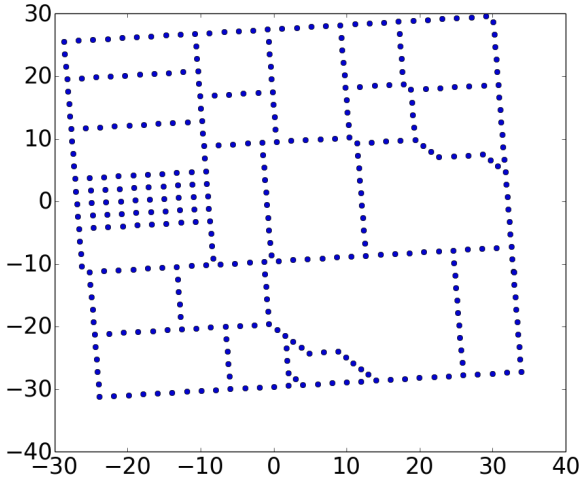
map after the node death is shown in Figure 4.3(b). It is assumed that the base station periodically updates the topological changes which are considered legitimate. Thus, we run our experiments assuming only one event occurs at a time i.e., either node death or an attack.

In sparse networks, the death of a single node brings about major topological changes. Consider the network in figure 4.4(a). All the nodes are critical in terms of connectivity. The nodes shown in red are made to die. The original TPM is shown in Figure 4.4(b) and the TPM after the node deaths is shown in Figure 4.4(c). As we see, even though only two nodes have been made to die, there are drastic changes in the topological map. This would make it difficult to differentiate between a node death and an attack. We show the performance of the techniques in terms of scalabilty in our simulations.

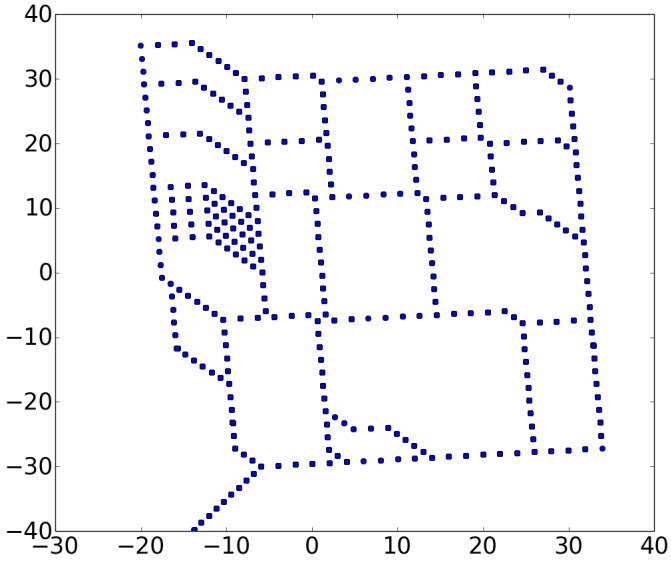
Thus, we see that the change in Virtual Coordinates are appropriately manisfested into the topology maps. An attack which may not be easily detectable in Virtual Coordinate domain can be easily captured in the topological domain. We make use of this fact in our detection techniques.



(a) VC Network



(b) TPM of Unattacked Network

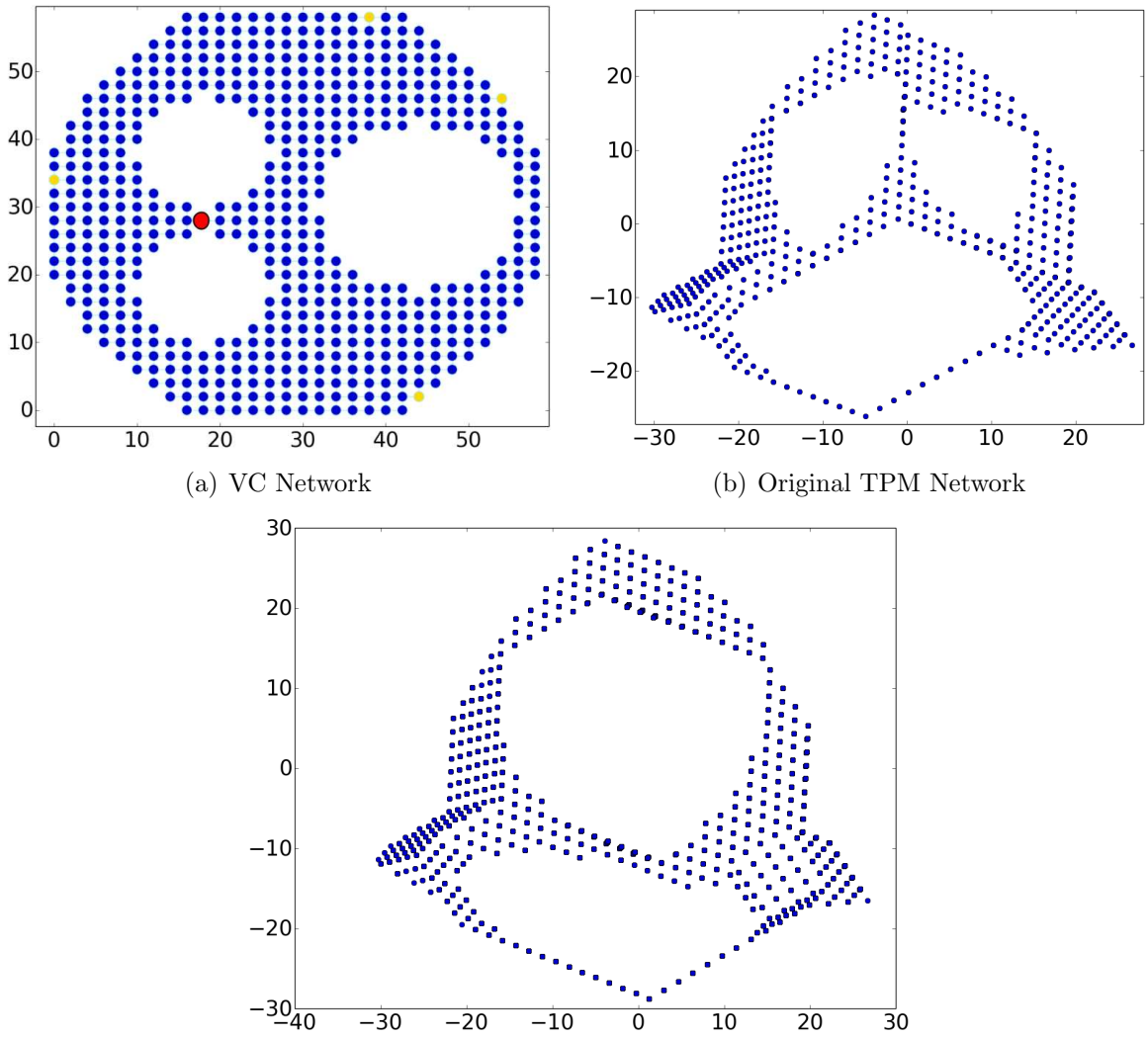


(c) TPM of Attacked Network

FIGURE 4.4. Building Network with Node Death

4.2.1. SPECIAL CASE OF NODE DEATH. Consider the network in Figure 4.5(a). The node shown in red is made to die. This is a critical node as it creates a bridge between two major regions of the network. The original and changed topological maps are shown in Figures 4.5(b) and 4.5(c) respectively.

As we see from these figures, death of a single node brings about a drastic change in the topological map. This is because the two regions which were earlier connected, now have to take a longer path to reach each other. These type of situations can result in false



(a) VC Network

(b) Original TPM Network

(c) TPM of Network after Node Death

FIGURE 4.5. Circular Network with Critical Node Death

positives even in dense networks. This exception should be handled effectively in order to avoid unnecessary false positives.

4.3. Localization of Attack

One may detect changes in the entire topology of the network by looking for changes in the edge node locations of the topological map. However, it may be possible that the attack does not affect the connectivity of edge nodes and the intensity of the attack may be concentrated on the middle nodes. Also maintaining the locations of all the edge nodes in

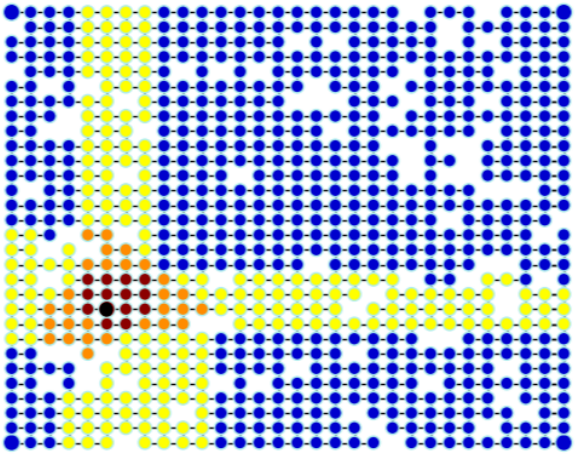


FIGURE 4.6. Intensity of Attack on a Random Network

the network increases memory consumption and computation overhead. Thus, we need to consider the region where the attack affects the most.

A random network with random density is chosen. It is mounted with Coordinate Deflation attack. As shown in Figure 4.6, the most affected nodes are shown in red followed by orange, yellow and blue. The intensity of the attack for each node is defined as the changed number of VC elements. In Figure 4.6, four anchor nodes are chosen. Hence, if all the four VCs are affected, they are shown in red, if three are affected, they are shown in orange and so on. The attacked node, shown in black is attacked with a Coordinate Deflation. It can be noticed in Figure 4.6, that the nodes in the vicinity of the attacked node are affected the most. Detection scheme which can capture this effect of the attack are proposed in Chapter 5. The deflation attracts all traffic towards itself by broadcasting lower hop counts. We can see in this figure, a blackhole is formed towards the attacked node. Thus, this attack causes looping while routing.

In the next chapter we present the detection techniques to detect VC based attacks. We use the topological changes to detect the attacks. We also present a topology distortion measure in order to capture the intensity of the attack.

CHAPTER 5

Coordinate Attack Detection

Attacks which disrupt the Virtual Coordinates of the WSN are very hard to detect since VCs of the the network change on a regular basis due to node failures. Also, Virtual Coordinates have no inbuilt directional information to detect a change in the position of its neighbors. It cannot detect presence of voids and other topological abnormalities. In this chapter, we present two techniques, Centralized Detection Technique (CDT) and Distributed Detection Technique (DDT), to detect coordinate attacks. Both techniques employ the Topological Coordinates (TCs) of the nodes, which are derived as shown in Section 4.1. CDT captures the the shape change in the topological map of the network. The detection computations are done in a base station/server. DDT restricts the attack to the neighborhood of the attack and the detection of an attack is done using the Topology Coordinates of the nodes of the network. The detection of an attack is made by the nodes of the network itself. The assumptions made in both techniques are now explained.

The initial period, when the network is deployed is assumed to be benign. This can be safely assumed since most WSNs are not accessible to the outside world just after deployment. During this time period, the correct VCs of the nodes in the network are generated. We consider only static networks, i.e. new nodes cannot be added in to the network in new positions. We may consider a time period when the network undergoes changes topologically while reconfiguration of the network occurs. This reconfiguration of network should be properly notified to the other nodes in the network, in order not to start the attack detection process due to this change. However, we do consider that nodes may die in the network due to battery drain or power failure. The attackers cannot oscillate between good and bad

behavior. However, changes in the behavior of nodes may be detected by the neighbors of that node. We consider only single attack at a time on the network. Only 2-D networks are in the scope of this thesis.

In this chapter, firstly we define intensity of attacks and node death. Secondly, we present our detection techniques.

5.1. Intensity of Attacks

This section will strengthen our theory of using TPMs for our detection techniques. The Coordinate Deflation and Wormhole attacks and node death affect the network in different ways in the VC domain as well as in the TC domain. We thus, define appropriate intensities for each attack and node death.

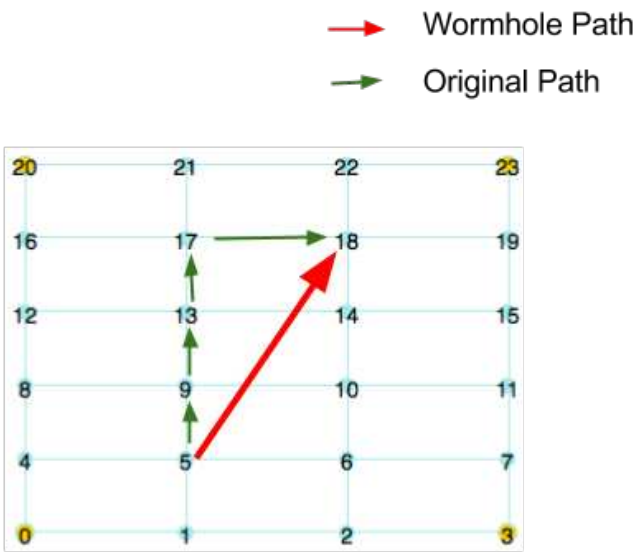


Figure 5.1. Example Network to explain Wormhole Intensity

The intensity of a Wormhole attack is defined as the number of hops overtaken by the Wormhole. Considering the network in Figure 5.1, we mount a Wormhole attack between

nodes 5 and 18. Before the attack, the hop count between these two nodes was 4 (5-9-13-17-18). So, the intensity of the attack is considered as 4.

For Coordinate Deflation attack, the intensity is calculated using the value by which the VC of the attacked node has been changed. Suppose the original VC of a node is [32, 31, 27, 26]. If an attacker generates the VC of the same node as [31, 30, 26, 25], then the intensity of the attack is considered as 4, since the amount by which the VC have changed is 4. The simulations are done upto getting VC as [0,0,0,0] for the attacked node. For the same, the intensity of the attack is 116 when the VC is changed to [0,0,0,0].

In the case of node death, intensity is calculated as the number of nodes which are affected due to the dying nodes. Affected Node is defined as a honest node experiencing a change in its VCs. If a dying node provides shorter hop count to more number of nodes in the network, the intensity of death of such a node will be higher. For a node whose death does not affect many nodes in the network, its failure will have lesser intensity.

Consider a rectangular network with 900 nodes arranged with 30 nodes in a row. In this network, the number of nodes whose VCs have changed due to an attack has been plotted against the intensity of an attack in Figure 5.2. As we see in Figure 5.2, the number of nodes affected by the attack is not proportional to the intensity of the attack. Thus, determining the presence of an attack based on the changed VCs may result in failure of detection. Also VC changes are common in WSNs due to node deaths/failures. However, changes in the network in VC domain is appropriately manifested in the topological domain as we have shown in Chapter 4.

Thus, a detection scheme that just verifies changes in the VC of the nodes may result in a lot of false positives or non detection. Topology maps, on the other hand provide clearer results of differentiating an attack from a node death. We have seen the distortions in TC

domain in Section 4.2. We now present the two techniques, CDT and DDT, to detect attack on a network.

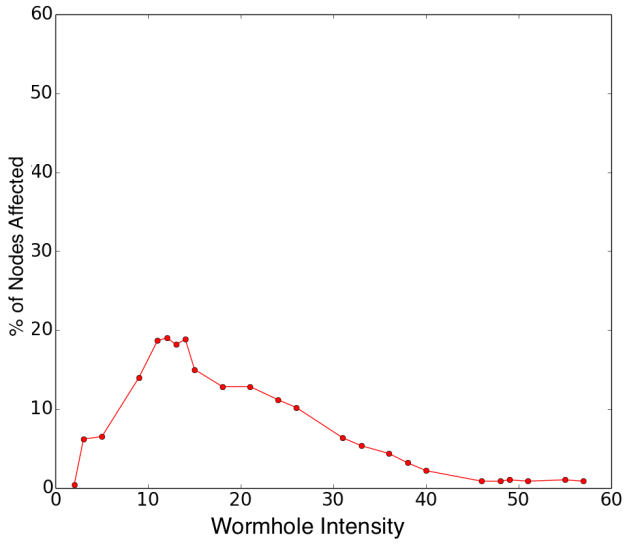


FIGURE 5.2. Intensity of Wormhole Attack vs Number of Node Affected

5.2. CENTRALIZED DETECTION TECHNIQUE (CDT)

In Figure 5.3, the attacked node is shown in black. The most affected nodes are shown in red followed by orange, yellow and blue. As we have seen in Section 4.3 and from Figure 5.3, the intensity of an attack is focused in the neighborhood of the attack. To capture this localization of attack, we divide the network into clusters. Each node in the network belongs to the cluster of that anchor node to which it is closest to. Each cluster is referred to by its corresponding anchor node. Clustering of the network gives the advantage of localizing the attack to a sub region of the network. The clustered representation of a Circular Network with voids is shown in Figure 5.4(a). The corresponding topology map is shown in Figure 5.4(b). The presence of a trusted and powerful base station is assumed for this technique. Also trusted communication between the anchor nodes and the base station is assumed. Each node in the network is able to communicate with its cluster head which in turn can communicate with the base station.

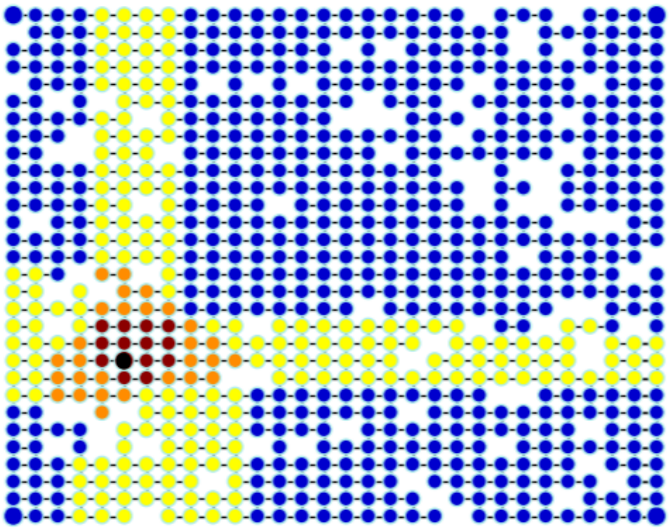
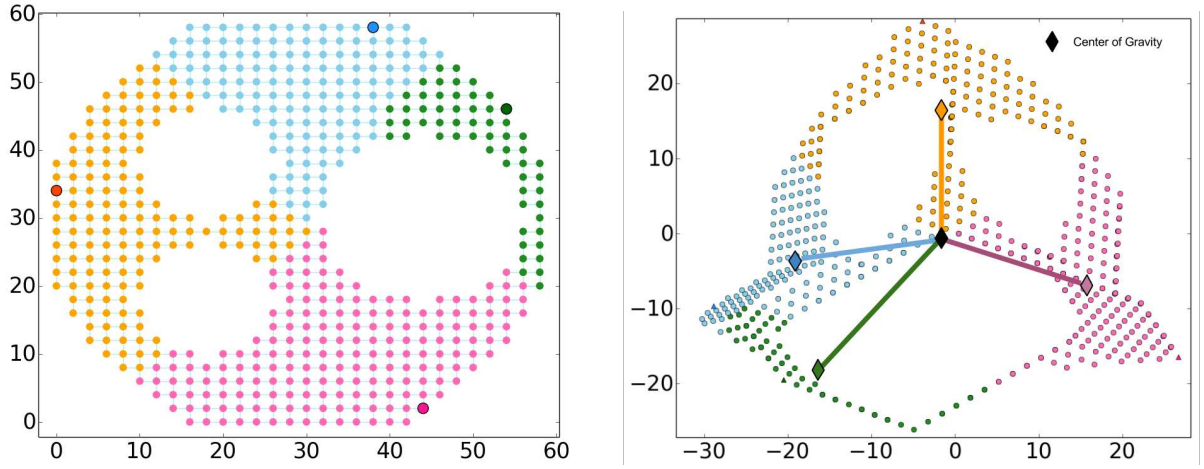


FIGURE 5.3. Intensity of Attack on a Random Network



(a) Clustered VC Network (b) TPM of Clustered Network

FIGURE 5.4. Circular Network with Anchor-based Clusters Identified

The center of gravity(CG) or mean (x,y) position of each cluster and of the entire network in the TC domain is shown in Figure 5.4(b). The distortion in the topology maps of pre-attacked and post-attacked networks are measured using the CG values of each cluster. The CG values are calculated using the Topology Coordinates of the nodes in the cluster/network. The pre-attacked CG values of the clusters and of the entire network is calculated at the initial stage of network development, when the network is assumed to be in benign conditions.

Since we are considering static networks, we keep the CG of the entire network constant for the life of the network. The CG values of the clusters are referred to as Cluster CGs and the CG of the entire network is referred to as the Network CG. Cluster Vector is defined as the connecting line from Network CG to the Cluster CGs.

For the detection technique, we need to maintain the adjacent two cluster CGs, so that the angle between them can be measured. Visually, the adjacent Cluster CGs for any single cluster can be easily determined. However, to find the adjacent topological neighbors for a cluster systematically, we use the angle of the cluster vectors to the X-axis. Consider the Cluster Vectors for the Circular Network in Figure 5.5.

The angles ψ_2 and ψ_3 are calculated between $0 - 2\pi$ anti-clockwise from the X-axis. Similarly ψ_1 and ψ_4 are calculated. The closest two cluster CGs are chosen as the adjacent neighbors. In Figure 5.5, $\psi_2 < \psi_3 < \psi_4$. Thus anchor A_3 chooses A_2 and A_4 as its adjacent anchors. Similarly, A_1 chooses A_4 and A_2, A_2 chooses A_1 and A_3 and A_4 chooses A_3 and A_1.

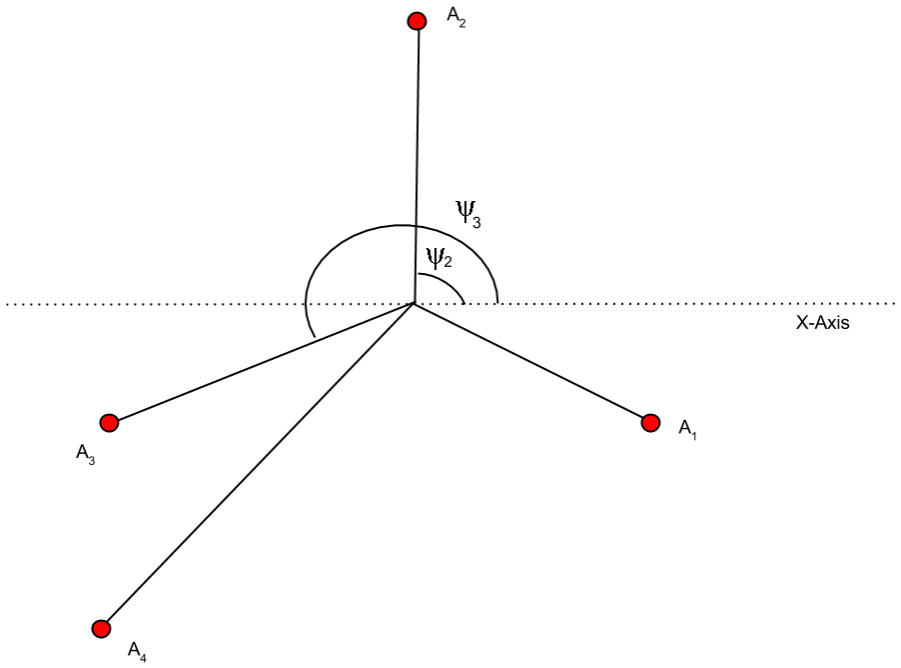


FIGURE 5.5. Adjacent Angle Calculation

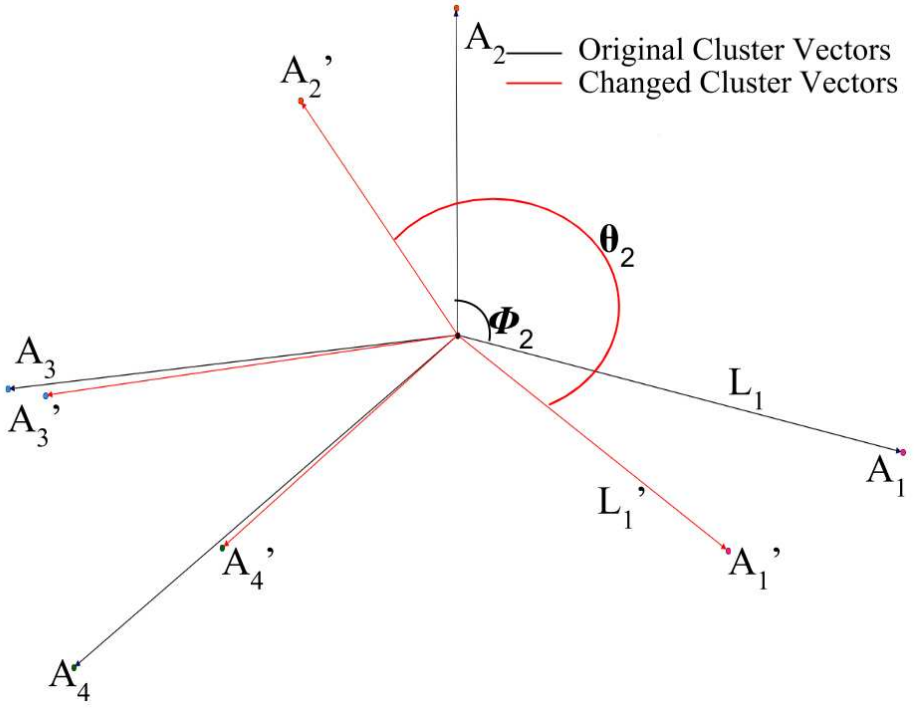


FIGURE 5.6. Original vs Changed CGs

The network in Figure 5.4(a) is mounted with a Wormhole attack. The pre-attacked cluster vectors and the post-attacked cluster vectors are shown in Figure 5.6. The original vectors are represented in black and the changed vectors in red. As clearly seen from the Figure 5.6, there is a large difference between the angles ϕ and θ. Also difference in the lengths of the original and changed cluster vectors are noticeable. For detecting an attack, the change in position of the Cluster CGs is needed with respect to self Cluster CG and with respect to the adjacent Cluster CGs. To achieve this, we use the angle between two adjacent cluster CGs and the length of the self Cluster Vectors.

TABLE 5.1. Mnemonics for CDT Algorithm

Symbol	**Explanation**
A_i	i^{th} Anchor Node Cluster Vector
ϕ	Angle between Original Cluster Vectors with Adjacent Anchors
θ	Angle between Changed Cluster Vectors with Adjacent Anchors
L_i	Length of line from the Network's CG to the i^{th} cluster's original CG
L'_i	Length of line from the Network's CG to the i^{th} cluster's changed CG

Table 5.1 gives the mnemonics which will be helpful in further explanation. ϕ of each cluster vector with its two adjacent Cluster Vectors are calculated at the time of network setup. Also, the length of each Cluster Vector is recorded. In Figure 5.6, Cluster Vector A_1 maintains ϕ_2 with Cluster Vector A_2 and ϕ_4 with Cluster Vector A_4. Also the length L_1, which is the distance between the Network CG to the cluster A_1's CG, is calculated. After changes in the Virtual Coordinates of the network, the CG of the clusters are recalculated to find the changed position as per the changed VC of the nodes. A'_1 then calculates θ value to A'_2 and A'_4 and also L'_1. The algorithm to detect whether this change in network is due to node death or a malicious attack is explained in Algorithm 3.

We define two terms, the *Detection Rating* D and the *Threshold Value* T. The value of D, provides a rating for the entire network using both the difference in ϕ and θ, and the difference in length of the Cluster Vectors of each cluster. T is the threshold value that is set, above which the network is declared under attack and below which the change in topology is considered legitimate. A clearly defined threshold is needed for this algorithm to make a

Algorithm 3 Algorithm to calculate Detection Rating of CDT

Let number of clusters be M

Consider the cluster of Anchor A_i

Adjacent topological neighbors : A_j and A_k

Corresponding original adjacent angles with neighbors : ϕ_j and ϕ_k

Corresponding changed adjacent angles with neighbors : θ_j and θ_k

Length of original Cluster Vector : L_i

Length of changed Cluster Vector : L'_i

Angle rating for cluster i $\quad \gamma_i = \frac{\frac{|\phi_j - \theta_j|}{\phi_j} + \frac{|\phi_k - \theta_k|}{\phi_k}}{2}$

Length rating for cluster i $\quad \lambda_i = \frac{|L_i - L'_i|}{L_i}$

Cluster rating $C_i = \gamma_i + \lambda_i$

Detection Rating $D = \frac{\sum_{i=1}^{M} C_i}{M}$

if $D > T$ **then** $Result = True$

else $Result = False$

end if

return Result

decision. This can be measured using simulation networks or by the base station learning the value D over a period of time. In the first case, we can determine the model of the network using some dimensions of the location where the network is going to be deployed. In the latter case an initial period of time is needed where there are no attacks on the network. However, we see in further sections, that for a particular density of networks, the threshold can be clearly defined.

5.3. Distributed Detection Technique (DDT)

On a rectangular network of 900 nodes, the number nodes whose VCs are changed due to an increasing intensity of Coordinate Deflation attack on a single node is plotted in

Figure 5.7. From this figure, we see that an attack on a single node can affect 50% of the nodes in the network. For the detection method proposed in Section 5.2, the base station needs to gather information about the changed VC of all the affected nodes. This gives a worst case complexity of $\mathcal{O}(N)$, where N is the number of nodes. Moreover, the presence of a trusted base station may not be always assured.

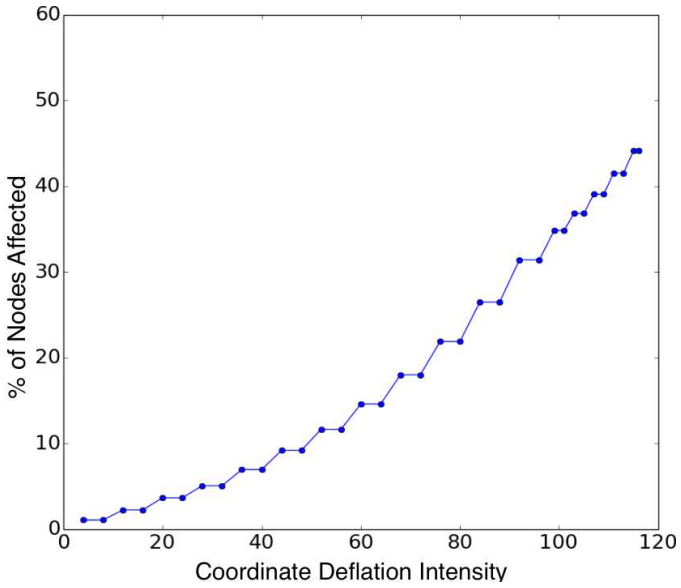
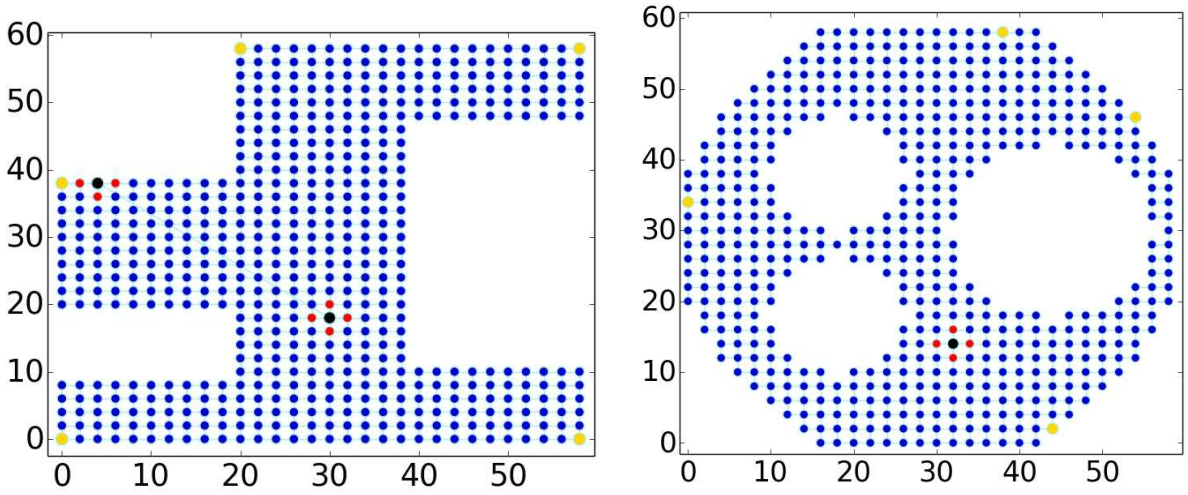


FIGURE 5.7. Intensity of Coordinate Deflation Attack vs Number of Nodes Affected

A detection technique where the nodes of the network themselves can make an attack detection is very beneficial when it is not possible to monitor a network continuously by a server. Also this makes the detection process much faster and restricts the attack from causing further damage. In this section, we present such a detection technique in which the neighbors of the node whose VCs have changed, together make a decision whether the change is a legitimate or illegitimate change. The Topological Coordinates (TCs) manifest the connectivity of the nodes in the network. Thus, directionality information, which is lost in VC domain is regained in TC domain. The changes in TCs of a node due to an attack on

a neighbor will be vastly different from that due to a death of a neighbor node. We capture this difference in this detection algorithm.

Consider the attacks on networks shown in Figure 5.8. Here we make an addition to the regular functionality of VC updating. Any legitimate node which senses a change in its VC due to a neighbor, neither accept nor forwards the change until it is sure that the change is a legitimate change. This prevents the attack from being spread all over the network. In Figure 5.8, the attacked nodes are shown in black and the affected neighbors are shown in red. Further, we present the algorithm and flow of DDT.



(a) Wormhole (b) Coordinate Deflation

FIGURE 5.8. Neighbor Nodes affected by Attacks

A random number of monitor nodes are chosen at initial setup of network. The honest nodes whose VC have changed, check the distance by which they themselves move due to the VC change by using Algorithm 4. The Distance Measurement is done using a Euclidean distance between the TCs generated using original VCs and the TCs generated using the changed VCs. The TCs are generated using Equation 4.3.

Algorithm 4 Algorithm for Distance Measurement

1: Let the number of Anchor Nodes in the network be M

2: Original VC matrix of node P_{i_o}= $[o_0\ o_1\ o_2\ .....\ o_M]$

3: Changed VC matrix of node P_{i_c}= $[c_0\ c_1\ c_2\ .....\ c_M]$

4: $V^{(2)}$, $V^{(3)}$ = 2^{nd} and 3^{rd} columns of matrix V returned by the SVD

5: Original (x_o, y_o) coordinates for node = $(P_{i_o}.V^{(2)},\ P_{i_o}.V^{(3)})$

6: Changed (x_c, y_c) coordinates for node = $(P_{i_c}.V^{(2)},\ P_{i_c}.V^{(3)})$

7: Distance shifted by node due to any change = $d = \sqrt{(x_o - x_c)^2 + (y_o - y_c)^2}$

8: *return* d

The honest nodes on calculating the value of d for respective selves, notify the monitor nodes about the same. The monitor nodes on receiving one Distance Measurement d, waits for a specific period of time for any other nodes trying to report their d values. When that time period is finished, the monitor nodes then follow Algorithm 5 to make a decision of whether the network is under attack or the change in VC is due to an environmental change.

Algorithm 5 Algorithm for Detection Rating of DDT

1: Let the number of honest nodes affected be k

2: Set of changed d values received by monitor nodes : $[d_0, d_1, ...d_{k-1}]$

3: Let the threshold value be T

4: Average d = Detection Rating = $D = \frac{d_0+d_1+.....+d_{k-1}}{k}$

5: **if** $D > T$ **then** Network is under Attack

6: **else** Environmental Change

7: **end if**

This algorithm needs additional memory consumption in each node to store the values of $V^{(2)}$ and $V^{(3)}$. Each node also has additional computation of calculating the Topological Coordinates using the previous and new VCs. However, the attack affects only a minor number of nodes which is restricted to the neighborhood of the attack. Also the need for a base station is removed. A clear threshold to differentiate between an attack and node deaths using this algorithm is seen from the results in the next chapter.

CHAPTER 6

Results

In this chapter, we present results to show the effectiveness of the two detection techniques, Centralized Detection Technique (CDT) and Distributed Detection Technique (DDT), proposed in the previous chapter. The simulations are performed on various networks given in [27]. To show scalability of the algorithms, simulations for a huge network are also presented. Coordinate Deflation, Wormhole attacks and node deaths of random nature, with varying intensity are simulated on these networks. Our results demonstrate the performance of the detection algorithms with respect to network scale and intensity of attacks or node deaths. We also compare the two algorithms in terms of cost, memory and computational overheads.

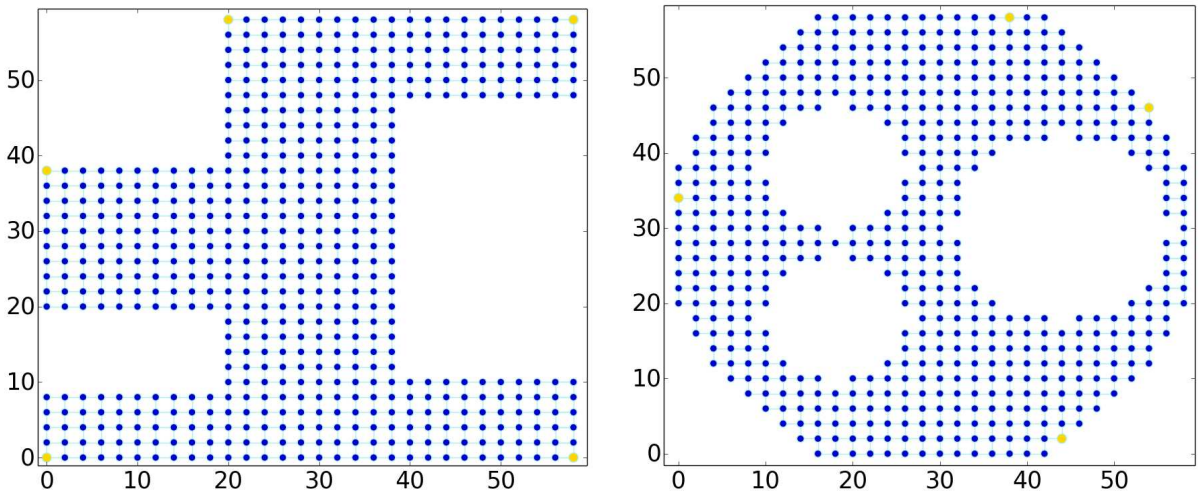
In following sections, firstly we present the simulation networks that we have used for our experiments. Secondly, we define Correct Detection, No Detection and False Positive terms. These terms will be used for demonstrating the effectiveness of the techniques. Further, the simulation results for CDT and DDT are presented. Then, we compare both the techniques in terms of number of messages, memory cost and computational overheads. And finally, we summarize our results.

6.1. Simulation Networks

The networks used for simulations are divided into three categories of networks namely 1) dense networks, 2) sparse network and 3) large network.

6.1.1. Dense Networks. Benchmark networks as shown in Figures 6.1(a) and 6.1(b) [27] are used as simulation networks in this category. We are using the Block network and Circular Network with voids as dense networks. These are networks with 750-900 nodes in the

network. We consider these networks because they provide simulations of physical voids and the environments in which WSNs are deployed. The anchor nodes in these networks are chosen using the ENS Algorithm [26].



(a) Block Network (b) Circular Network with Voids

FIGURE 6.1. Dense Networks

6.1.2. SPARSE NETWORK. We use the Building network [27] to verify the performance of the algorithms in sparse network. This network has 344 nodes, arranged as shown in Figure 6.2. This network represents a WSN being deployed over a building or a house. Such WSNs are used for applications like theft security and fire detection in a building.

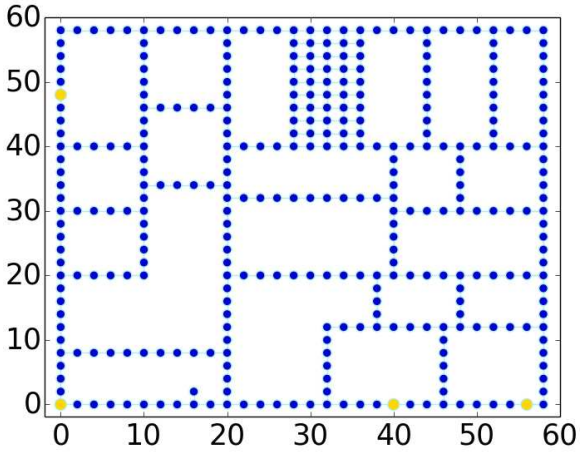


FIGURE 6.2. Sparse Building Network

6.1.3. Large Network. For this category, we choose a network with 10,000 nodes deployed in a rectangular area as shown in Figure 6.3. In the same network, 50 nodes are randomely removed for random deployment nature. As WSNs are predicted to be scaled to huge node densities, it is necessary to design scalable algorithms for WSNs. We show the scalability of our detection techniques using this network. Eight anchor nodes are chosen in this network.

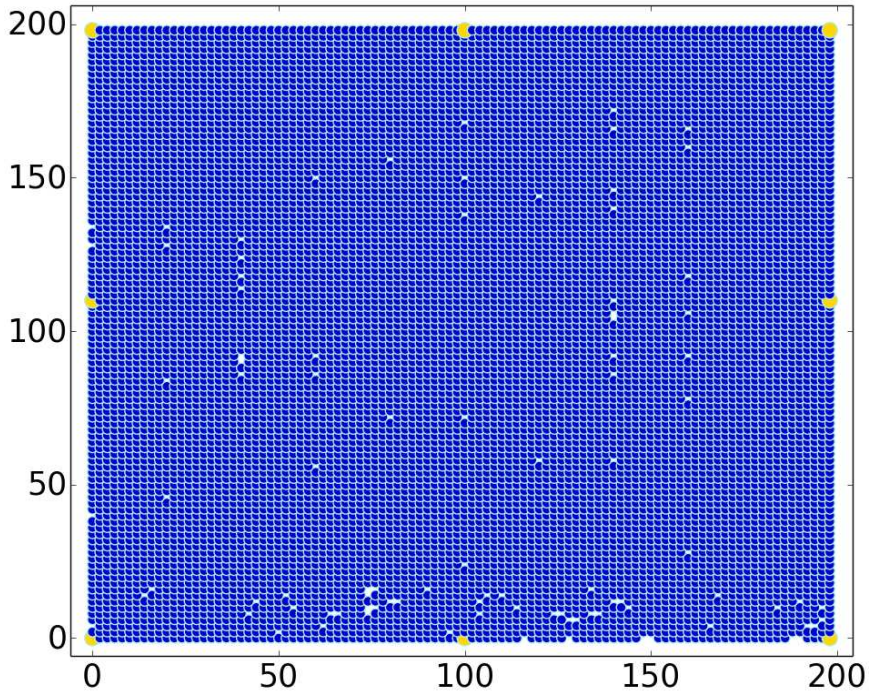
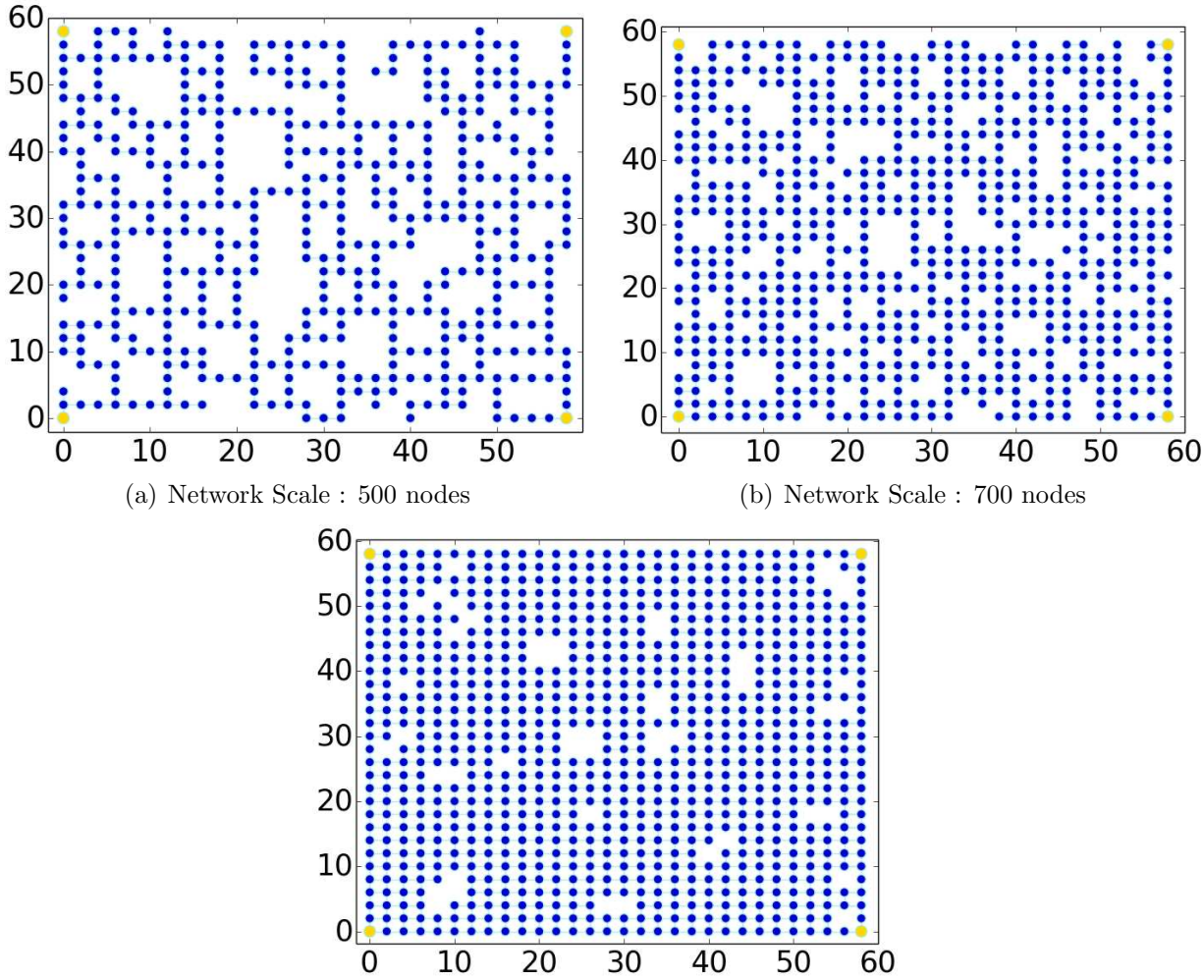


Figure 6.3. Large Network

6.1.4. Random Networks. To simulate the performance of the detection techniques with respect to network scale, we choose networks of randomly deployed nodes over a rectangular area/grid. Some examples of random networks are shown in Figure 6.4. These networks vary in the range of 300 to 850 nodes in the network.



(a) Network Scale : 500 nodes

(b) Network Scale : 700 nodes

(c) Network Scale : 850 nodes

FIGURE 6.4. Random Networks

6.2. DEFINITIONS

In CDT and DDT, we define two terms, Detection Rating, D and Threshold T. D is the value returned by the detection techniques to determine the presence of an attack. T is the threshold value set after a number of simulation runs to differentiate between an attack and a node death.

6.2.1. Correct Detection. Correct Detection corresponds to the situations where during an attack, the Detection Rating, D, is a value greater than the threshold value, T. Correct detection also corresponds to T being less than D in case of a node death.

6.2.2. False Positive. False Positive is generated in the situation where there is a node death, but D comes out to be greater than T, i.e. node death is considered as an attack on the network. When a node death causes a change in VCs of large percentage of nodes in the network, false positives are generated. Existence of a node whose removal or death causes separation of two regions of the network, may cause drastic topological changes in the network. Such a change may be detected as an attack and results in a false positive. One can argue that this is a drastic change that needs to be detected for the proper functioning of the deployed network even though it is not due to an attack. If such a false positive need to be avoided, it is possible to implement a mechanism in which a node provides a warning to its neighbors when its power is critically low and is about to die, and propagating this information for remedial action such as issuing a warning or re-configuring the network.

6.2.3. No Detection. No Detection is considered when D is less than T in case of an attack, i.e. even when there is an attack, the algorithm fails to make a detection. No Detection can be seen when the attack causes very few VCs to change in the network. For our simulation networks, they are the attacks, whose intensity is very less i.e. it propagates to very few nodes. Thus, these attacks are not captured by the algorithm. However, these attacks have in effect, minor or negligible impact. Thus, it is safe to over-look this drawback.

6.3. Results for Centralized Detection Technique (CDT)

In this section, we present the results for the evaluation of CDT introduced in Section 5.2. For each category of networks, the simulation networks are subjected to Coordinate Deflation

and Wormhole attacks one at a time. To evaluate the robustness of detection scheme against node deaths, which routinely happen in networks, we also subject the networks to random node deaths. 30 simulations of each of Coordinate Deflation, Wormhole and node deaths have been simulated for these results. We also present the values of CDT Detection Rating, D for varying values of the intensity of attacks and node death.

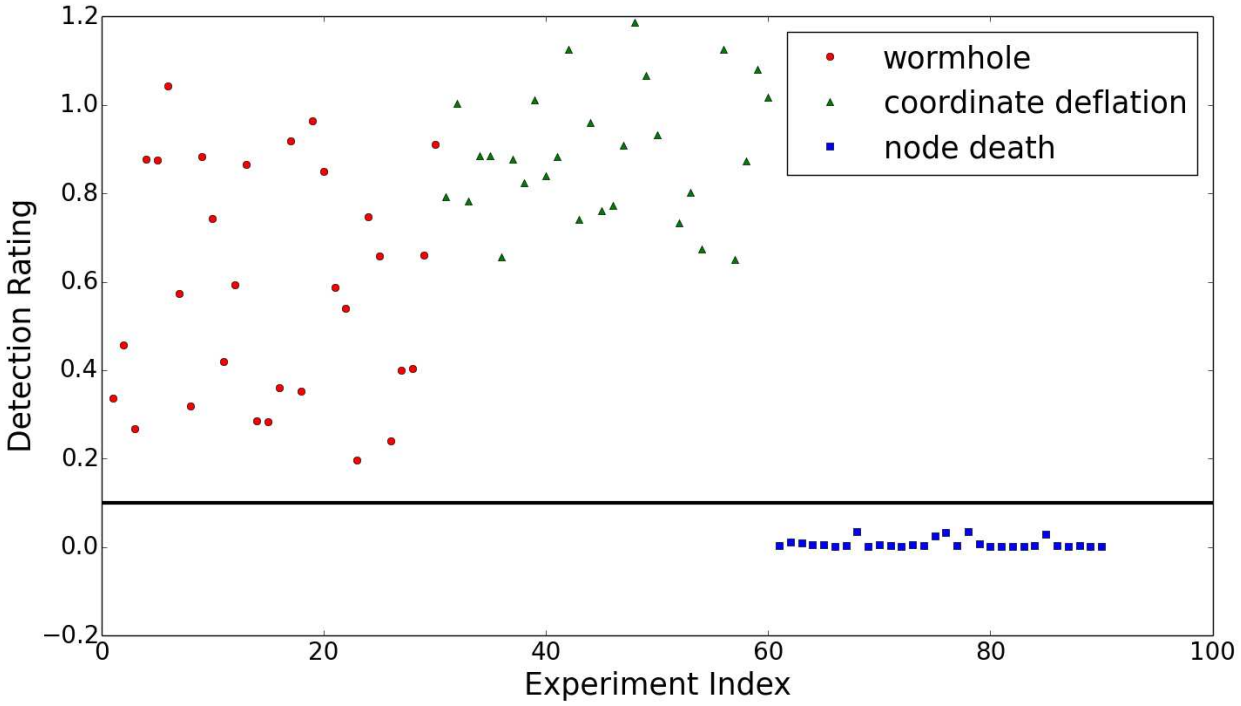


FIGURE 6.5. CDT Results for Block (Dense) Network

6.3.1. DENSE NETWORKS. The results for random attacks on the Block network and the Circular network are shown in Figure 6.5 and Figure 6.6. The X-axis shows the experiment index and the Y-Axis shows the value D. As we see from the results, the value of Detection Rating is higher than 0.1 for Deflation and Wormhole attacks and lower than 0.1 for node deaths. Thus, a clear threshold value can be deduced to differentiate between an attack and a node death.

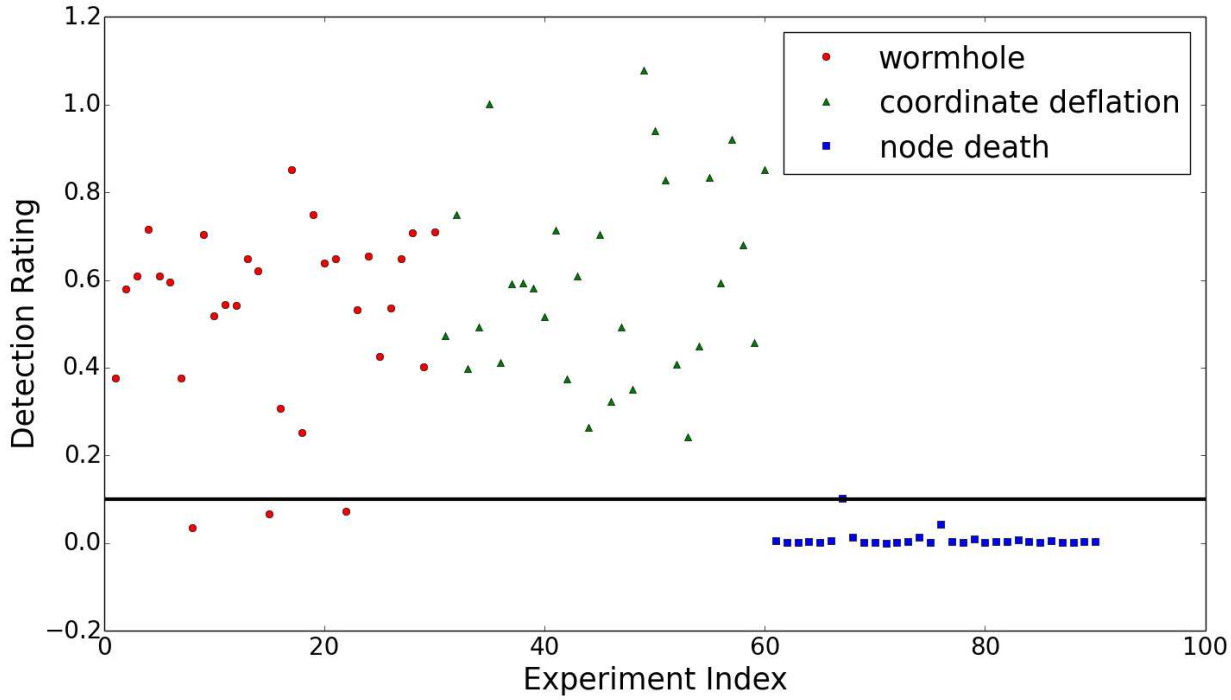
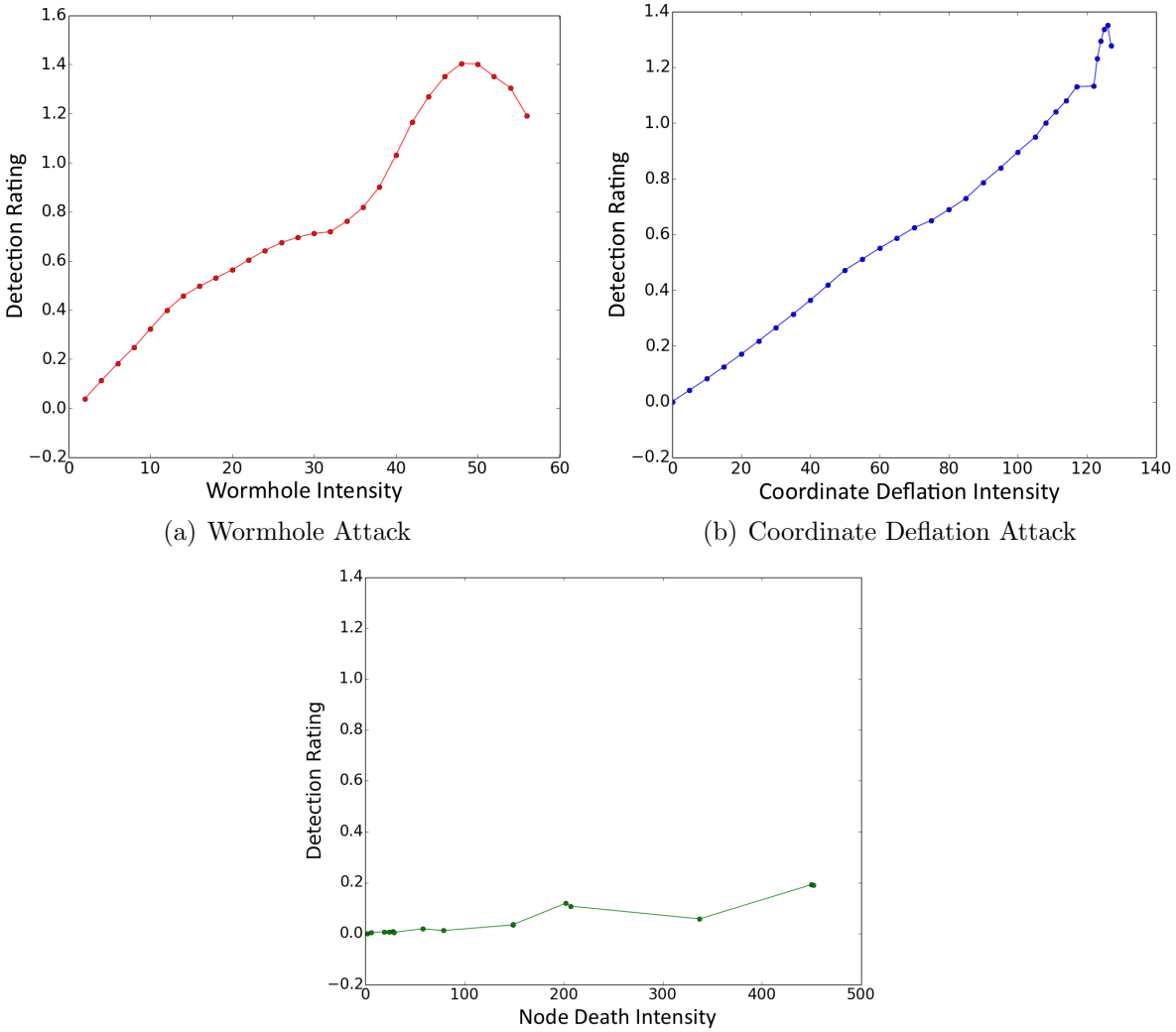


FIGURE 6.6. CDT Results for Circular (Dense) Network with Voids

Figures 6.7(a) and 6.7(b) show the CDT Rating value for different intensities of Coordinate Deflation and Wormhole attack on the Block Network. Wormhole Intensity is defined as the original number of hops between the two attacked nodes. Coordinate Deflation Intensity is the integer value by which the VCs of the attacked node have been modified. We vary the Deflation Intensity till reaching all zero VCs for the attacked node. This is the maximum intensity of Coordinate Deflation attack on a single node. We see that as the intensity of attack increases, the Detection Rating value, D, increases. This is because the topology map is more distorted as the intensity increases. Figure 6.7(c) shows the detection rating for increasing intensity of node death on the Block network. We deduce that even as the number of nodes affected by the node deaths increases, the detection rating value does not vary by a significant value. Hence, a lower threshold value may be set to detect attack

without generating a lot of false positives. Thus, the detection algorithm is very efficient in differentiating an attack from a node death for the Block network.



(a) Wormhole Attack

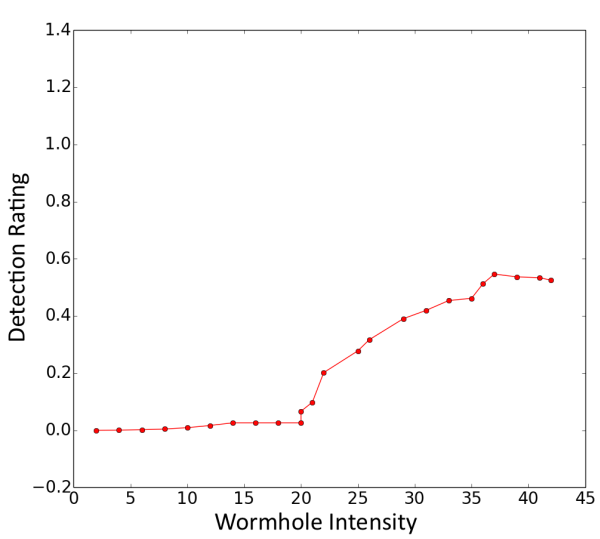
(b) Coordinate Deflation Attack

(c) Node Death

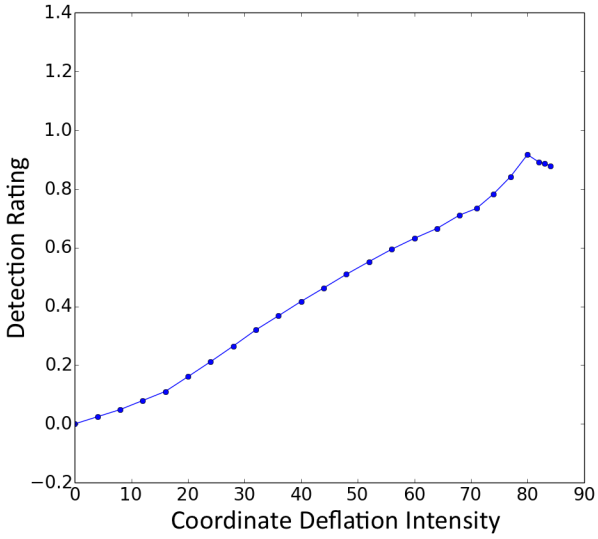
FIGURE 6.7. Attack Intensity vs CDT Results for Block (Dense) Network

Figures 6.8(a) and 6.8(b) plot the CDT Detection Rating value D for varying intensity of Wormhole and Coordinate Deflation attack on the Circular Network with voids. We see similar results of increasing D value with increasing intensity of attack. In Wormhole attack, the attacks which are not detected, are feeble and affect only a minor number of nodes. However, in Figure 6.8(c) we see high detection rating for node death for increased intensity.

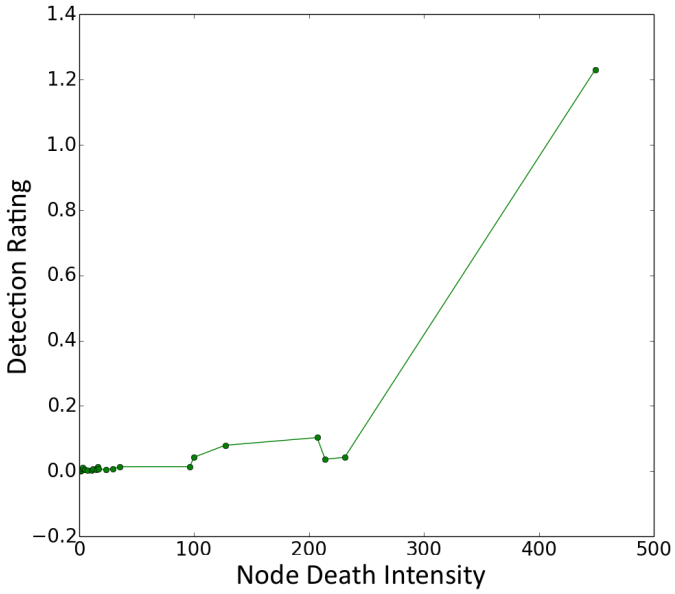
This is the scenario which we have discussed in Section 4.2.1. However, proper notifying mechanisms can be put in place to let the neighbors of critical nodes be known of the critical node's death. Hence, the algorithm may be successfully implemented with high percentage of correct detection.



(a) Wormhole Attack



(b) Coordinate Deflation Attack



(c) Node Death

FIGURE 6.8. Attack Intensity vs CDT Results for Circular (Dense) Network with Voids

Thus, the algorithm performed very efficiently for the Block Network as this network is very well connected and does not have any critically connected nodes. For the Circular Network, the detection rating, D was sufficiently high for both attacks. However, a few critical node deaths resulted in a few false positives.

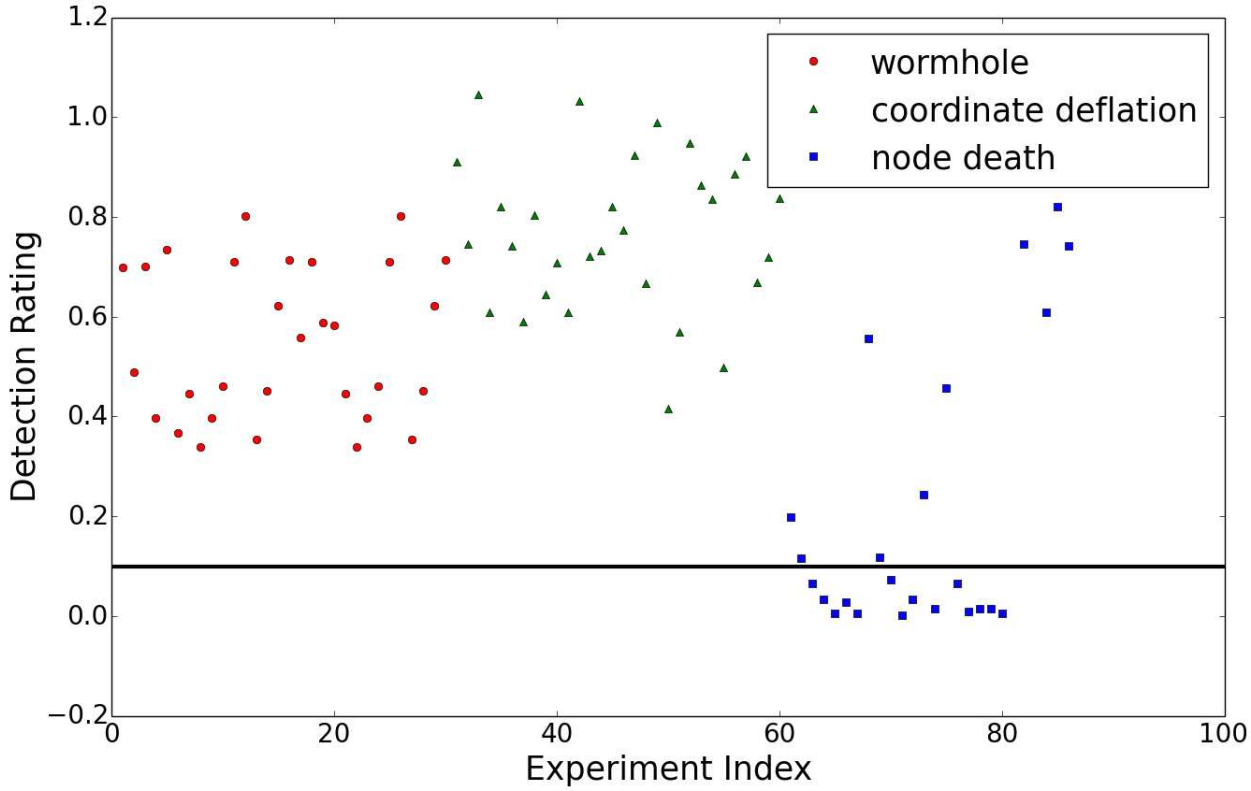
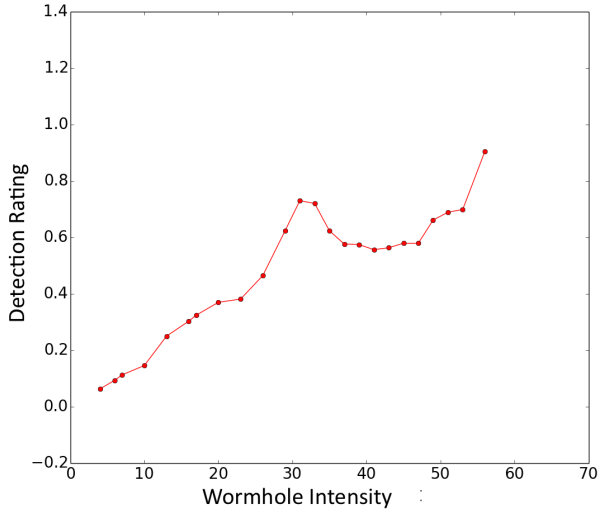


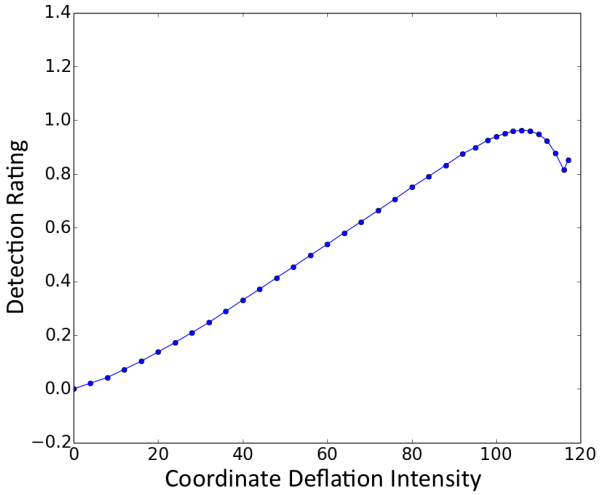
FIGURE 6.9. CDT Results for Building (Sparse) Network

6.3.2. SPARSE NETWORK. The results of the Sparse Building Network for CDT are shown in Figure 6.9. The Detection Rating value for Coordinate Deflation and Wormhole attack lies above a threshold value of 0.2. Thus, we see from this figure that the algorithm efficiently detects all the attacks on this network, if T was set 0.2. However, it also gives high CDT Rating for many node deaths. A clear threshold below which all node death ratings may lie, is hard to define. This is because the node density is very less in this network. Thus,

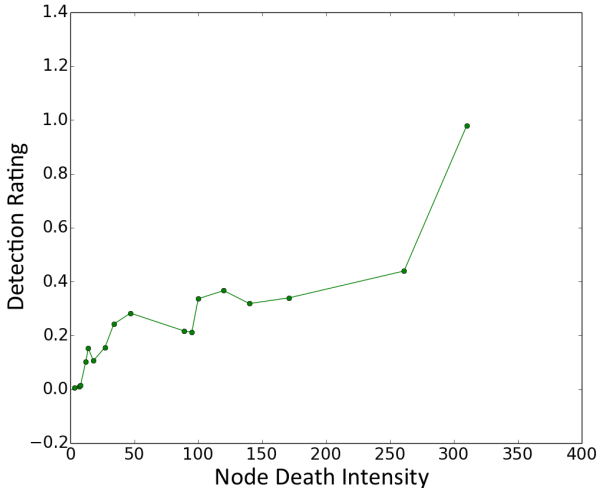
as explained in Section 4.1, even a single node death can cause major topological changes. This results in higher value of D.



(a) Wormhole Attack



(b) Coordinate Deflation Attack



(c) Node Death

FIGURE 6.10. Attack Intensity vs CDT Results for Building (Sparse) Network

We further plot the value of D for different intensities of attack. Figures 6.10(a) and 6.10(b) show the CDT Detection Rating for Coordinate Deflation and Wormhole attack on the Building network. The algorithm successfully increases the value of D with increasing intensity of attack. However, increasing intensity of node deaths also increases the value of

D. This can be attributed to the lower density of the network and lower connectivity of the nodes. Thus, a defining threshold is hard to set for Sparse Networks.

6.3.3. Large Network. In CDT, one of the approaches to achieve scalabilty would be to divide the network into further clusters to reduce the memory requirement in each anchor node. In the case of huge networks, the network may be divided such that two base stations may monitor two regions of the network and follow the detection algorithm separately.

For the large network with 10,000 nodes, we have chosen 8 anchor nodes and thus divided the network into eight clusters. The CDT Rating for 90 random simulations of attacks and node deaths on this network are shown in Figure 6.11. We see that the rating values are largely different for attacks and death of nodes. In the case of dense networks, node deaths do not affect the topology map hugely as the connectivity of the nodes are very high throughout the network.

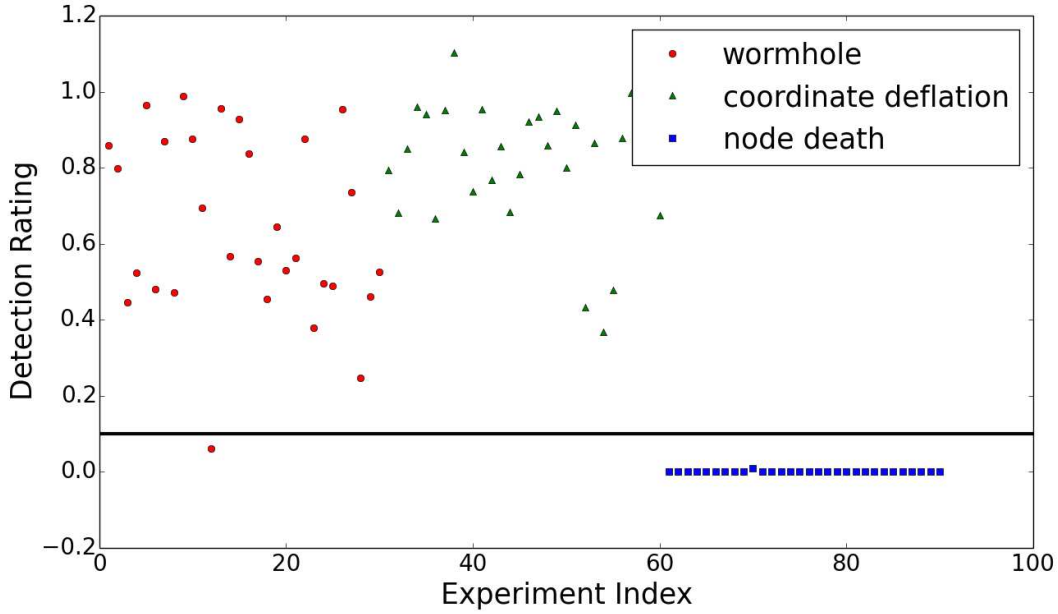
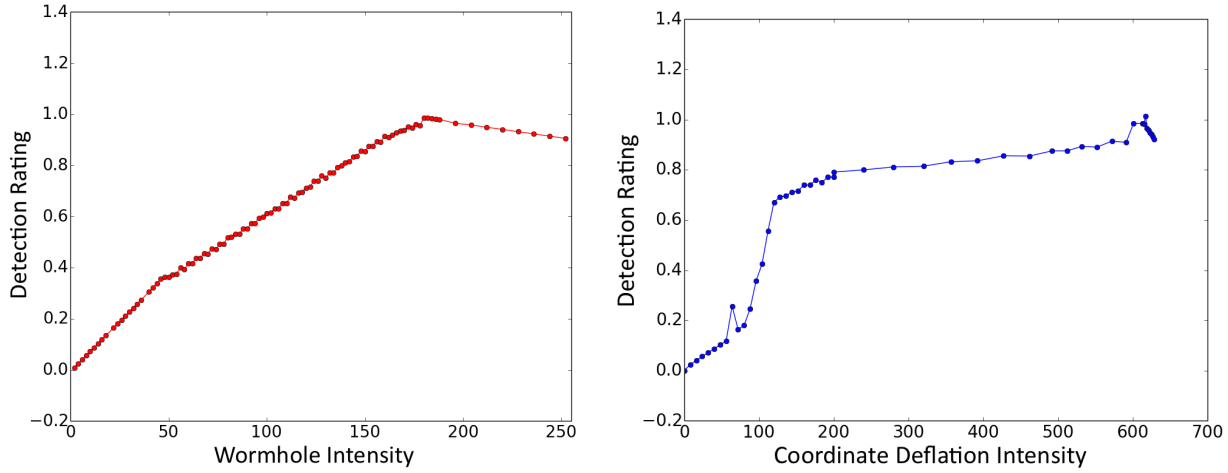


Figure 6.11. CDT Results for Large Network with 10,000 Nodes

The results of CDT with increasing intensity of the attacks are shown in Figure 6.12 for the large network. We see similar results of increasing CDT Detection Rating with increase in intensity of attack. The death of many nodes at a time, resulted in a negligible value of D as shown in Figure 6.11. Thus, CDT algorithm can be implemented on huge networks successfully.

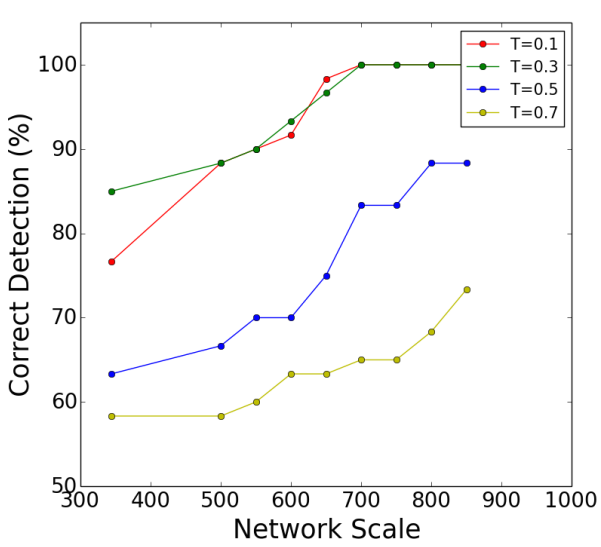


(a) Wormhole Attack (b) Coordinate Deflation Attack

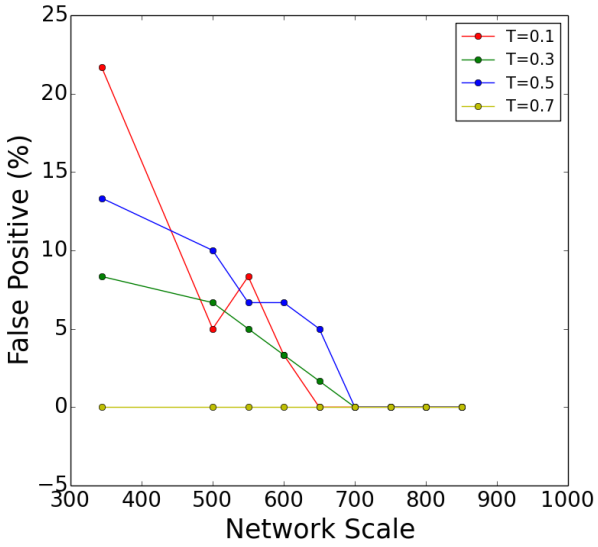
FIGURE 6.12. CDT Results vs Attack Intensity for Large Network with 10,000 Nodes

6.3.4. RESULTS ON SCALING THE NETWORK. Now we compare the value of the CDT Detection Rating with respect to the network scale. Figure 6.13 shows the performance of the algorithm as the network scale increases from 350 to 850 nodes. The performance of the algorithm with respect to the network scale is compared to correct detection, false positive and no detection. These results are shown in Figures 6.13(a), 6.13(b) and 6.13(c).

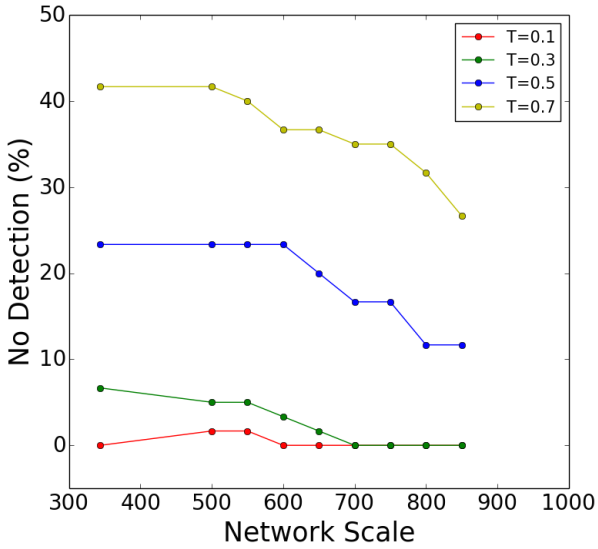
The experimental setup is as follows: 20 simulations of each of Coordinate Deflation, Wormhole and node deaths have been mounted on network of each scale. Each attack or death was random for a network with random intensities. However, the same attacks were mounted on networks of each scale and results were taken for T values of 0.1, 0.3, 0.5 and 0.7.



(a) Network Scale vs Correct Detection Results



(b) Network Scale vs False Positive Results



(c) Network Scale vs No Detection Results

FIGURE 6.13. Network Scale vs CDT Results

Figure 6.13(a) shows the plot, comparing the network scale with correct detection. The performance of the algorithm degrades as the value of T increases. We can see that a value between 0.1 to 0.3 is an ideal value for T. Also, we can deduce that the performance of the algorithm is poorer for lower network density. We only have a correct detection of 75 - 80% for network scale ranging from 350 to 650. However, for denser networks, the algorithm gives 100% correct detection for T value being 0.1 and 0.3.

Figure 6.13(b) shows the plot of network scale vs false positives. As the threshold value increases the number of false positive decreases. This is an obvious deduction, since as T increases, all D values for node death will come lesser than T. However, we see that for T=0.1 or T=0.3 the number of false positive detections are 0 for network scale more than 700 nodes.

Figure 6.13(c) shows the plot of network scale vs no detection. In this figure, we see that T=0.7 gives very bad results for all degrees of network scale. The value T being so high, the algorithm fails to detect most of the attacks. There are almost negligible failure of detections for T=0.1 and 0.3. Also, the network scale does not affect the no detection for lower values of T.

Thus, we can deduce that the algorithm performs exceptionally well for higher network density and lower threshold value. The percentage of false positives is high for the optimal T values for lower network scale. The algorithm cannot differentiate between an attack and a node death for lower network density. Thus, the algorithm is more suitable for networks with higher density.

We tabulate the results of random attacks and node deaths for CDT on benchmark networks presented in [27] in Table 6.1. The algorithm is very efficient for all networks except for the building network. Even in the case of building network, the percentage of correct detection is low only due to the fact that false positives are higher. This can be trustfully deduced since the percentage of no detection is zero for building network. Thus if a separate mechanism is implemented for trustworthy notification of node death, then CDT can be applied to Building Network as well.

TABLE 6.1. Results of CDT for various networks

Type of Network	T=0.1		
	Correct Detection	False Positive	Non Detection
Block	100%	0%	0%
Circular with Voids	94.44%	1.11%	4.44%
M-Shaped	100%	0%	0%
Concave Void	95.98%	3.26%	0.76%
Building	71.4%	28.6%	0.0%

6.4. Results for Distributed Detection Technique (DDT)

In this section, we present the results for the Distributed Detection Technique (DDT) proposed in Section 5.3. As in the previous section, detailed analysis is presented for the Dense Networks, Sparse Network and Large Network. 30 experiments each of Coordinate Deflation, Wormhole and node deaths have been performed on these networks. We also discuss the performance of DDT with varying intensity of attacks and node deaths. The efficiency of the algorithm with respect to network scale is also discussed later.

6.4.1. Dense Networks. Figure 6.14 shows the DDT Detection Rating for the Blocks network. Figure 6.15 shows the DDT Detection Rating for Circular network. The DDT Rating values, D clearly lies above a threshold value of 2 for all the attack simulations. Also, the node death simulations result in D value lesser than 2. Thus, a clear threshold can be deduced to differentiate between an attack and a node death. Both Wormhole and Deflation attack can be successfully detected in both the networks.

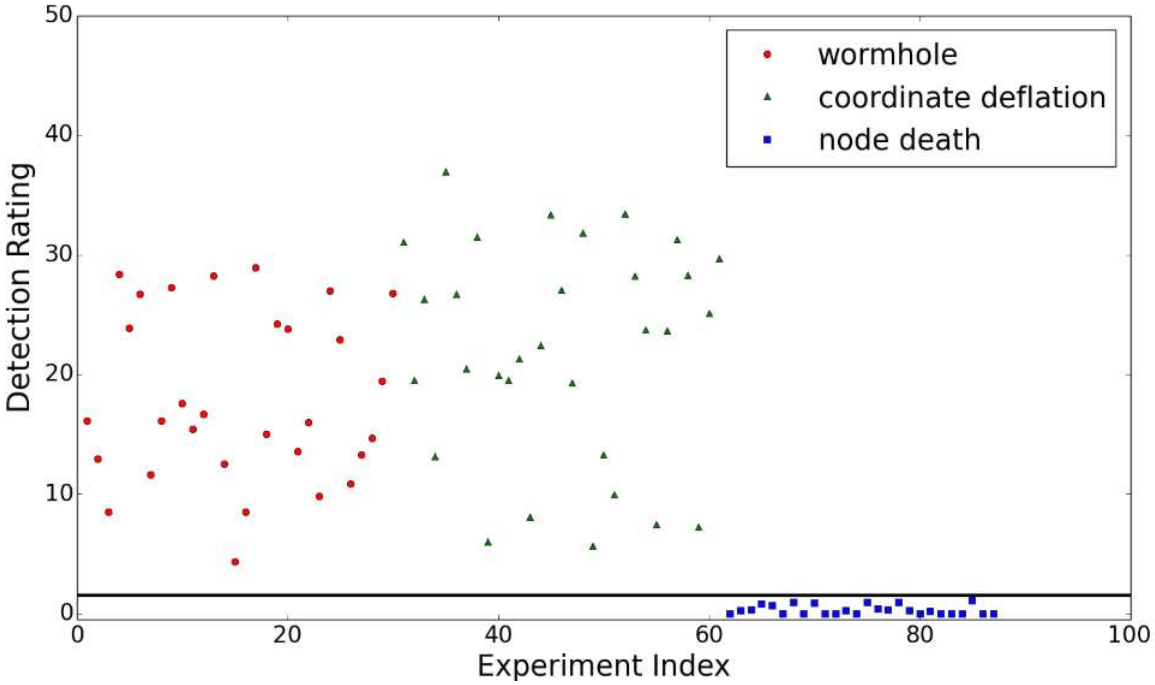


FIGURE 6.14. DDT Results for Block (Dense) Network

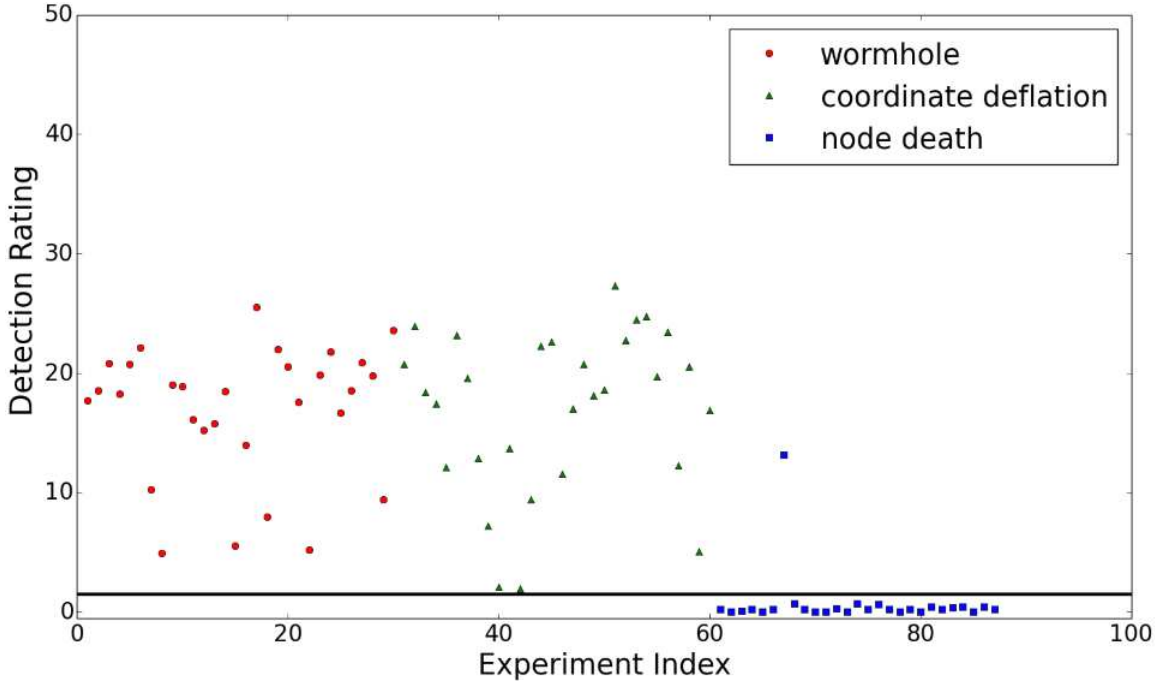
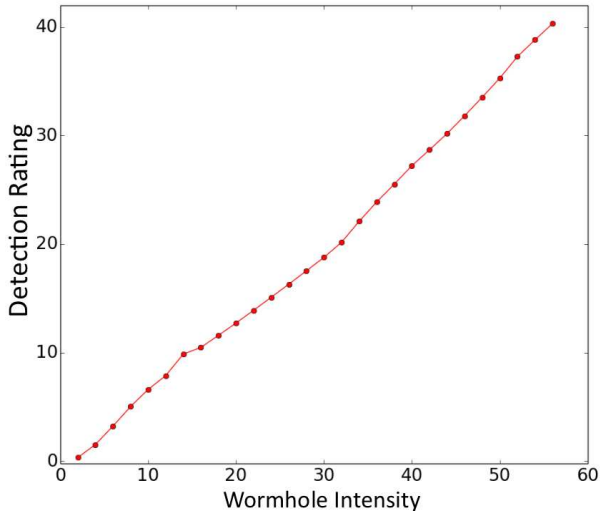


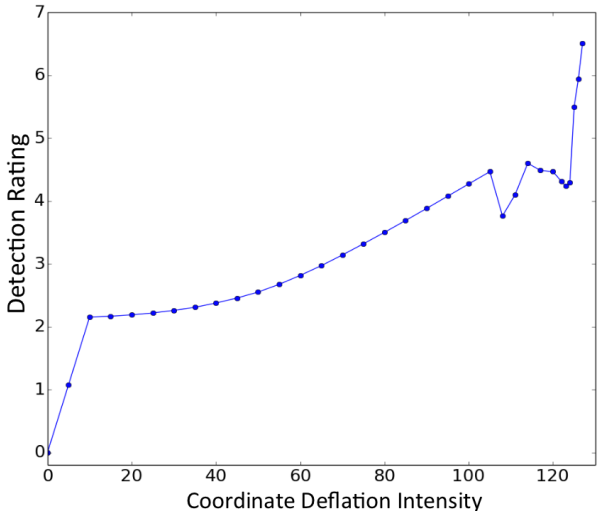
FIGURE 6.15. DDT Results for Circular (Dense) Network with Voids

Lower Intensity of attacks may result in no detection as the effect on Topology Coordinates are lower. We plot the intensity of the Wormhole and Coordination Deflation attack

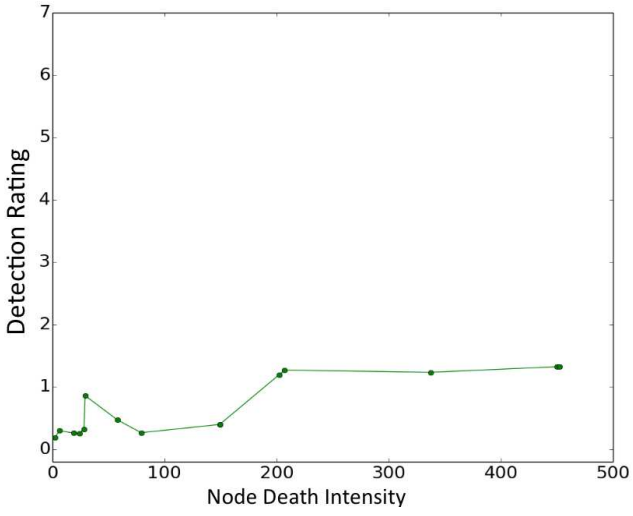
in Figures 6.16 and 6.17 for Block network and Circular network with voids. We see from Figure 6.16(a) and 6.17(a), which are the plot for varying Wormhole Intensity, that as the intensity of the attack increases, the Detection Rating value, D, also increases. This is because, as the intensity of the Wormhole attack increases, the attacked nodes themselves move by a proportional distance. This effect is propagated to its neighbors and they too move by a proportional distance. As we capture this shift in Topological Coordinates in DDT, the value D is effectively increasing with increasing intensity of attack.



(a) Wormhole Attack

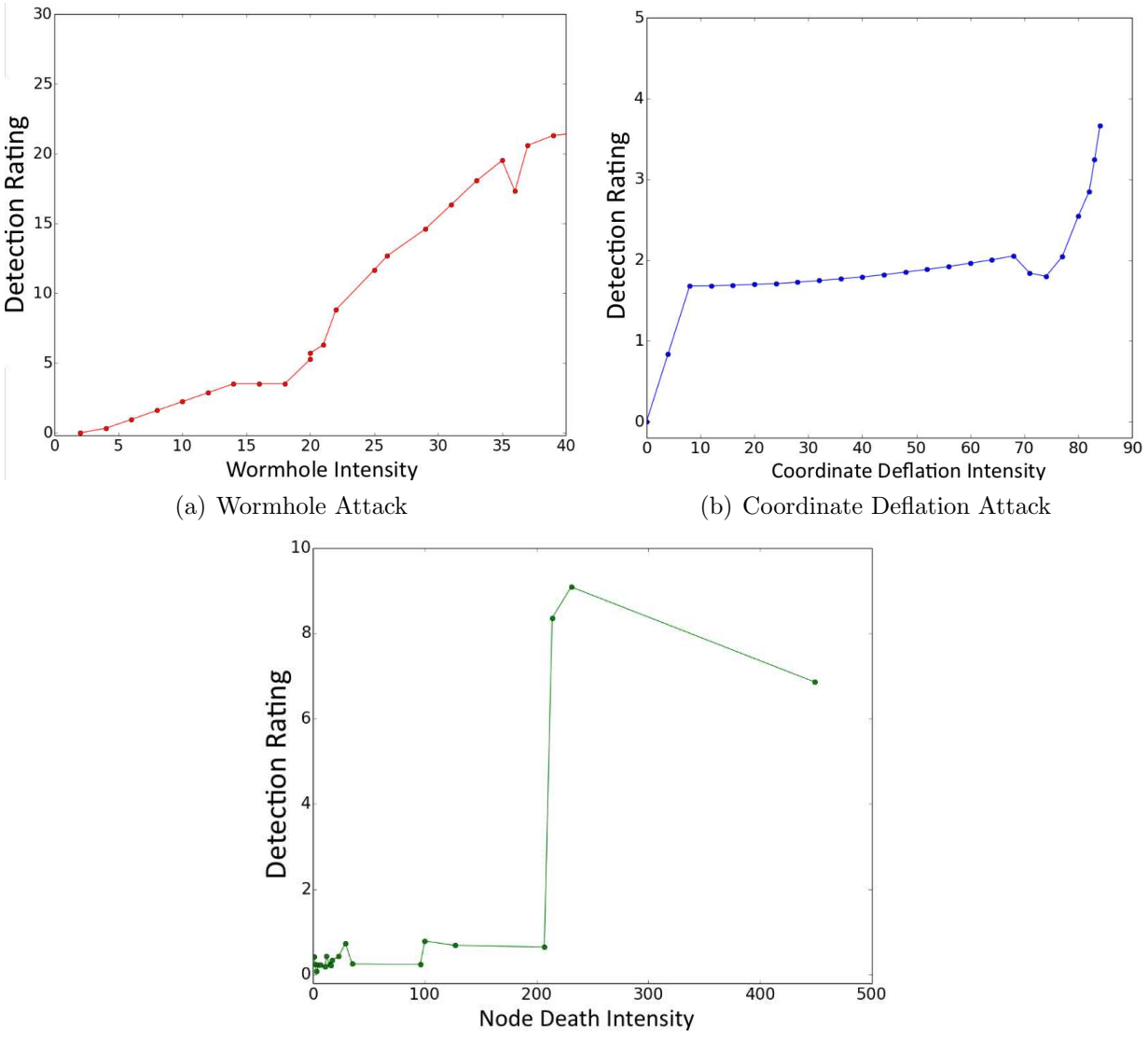


(b) Coordinate Deflation Attack



(c) Node Death

FIGURE 6.16. Intensity vs DDT Detection Rating for Block (Dense) Network



(a) Wormhole Attack

(b) Coordinate Deflation Attack

(c) Node Death

FIGURE 6.17. Intensity vs DDT Detection Rating for Circular (Dense) Network with Voids

However, the same effect is not seen for Coordinate Deflation attack. Figures 6.16(b) and 6.17(b), which are the plot for varying Coordinate Deflation Intensity, show that for even high intensity of Deflation attack, the distance shift seen by the neighbors is not as much as in Wormhole attack. This is because a Deflation attack creates a "blackhole" to attract all the traffic towards itself. However, the relative positions of the nodes to each other in its

neighborhood will not vary by a significant amount. However, as we see from Figures 6.16(c) and 6.17(c), which shows the value of D with increasing intensity of node death, that the DDT Rating for node death is even lower than that of a Deflation attack. Thus, a threshold can be defined to differentiate between node death and a Coordinate Deflation attack as seen in Figure 6.15.

One can notice that the rating for node deaths is quite stable for varying intensities to higher values. In the case of Circular network, we see increased rating due to the node death because of the death a critical bridging node as explained in Section 4.2.1.

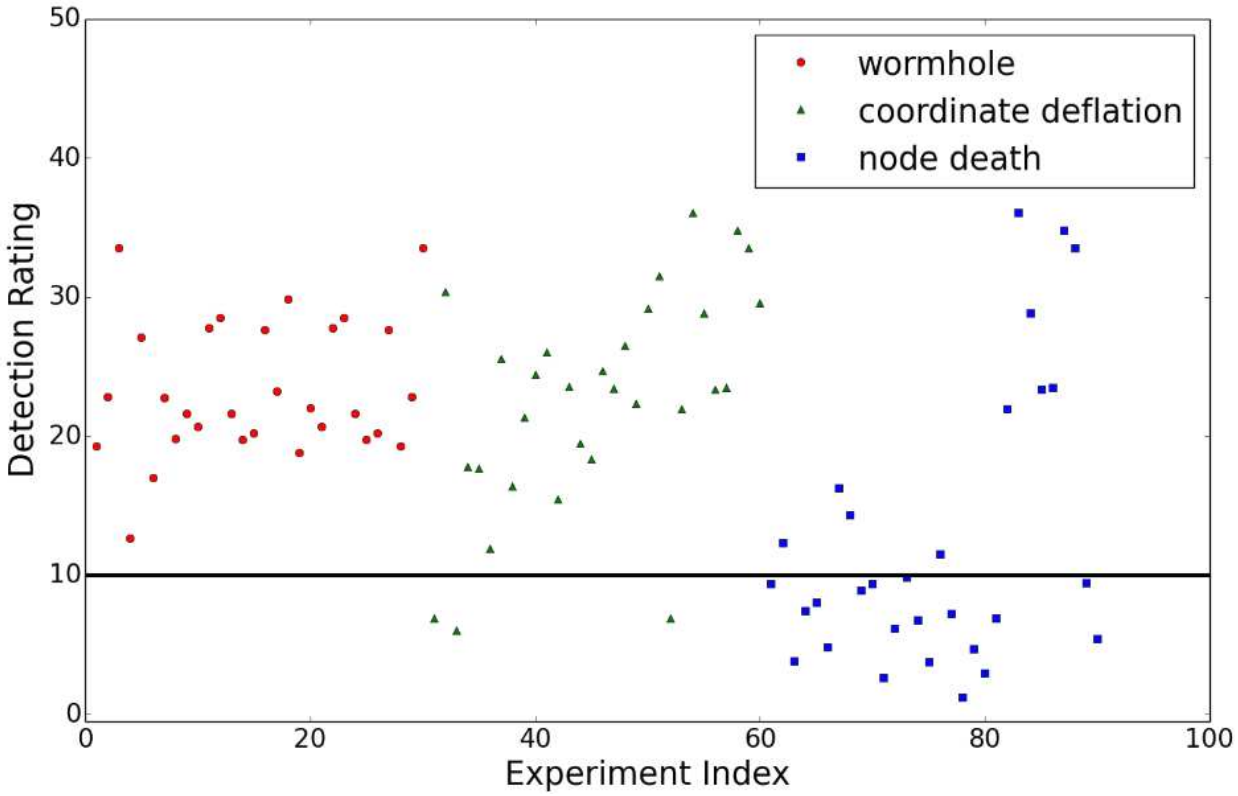
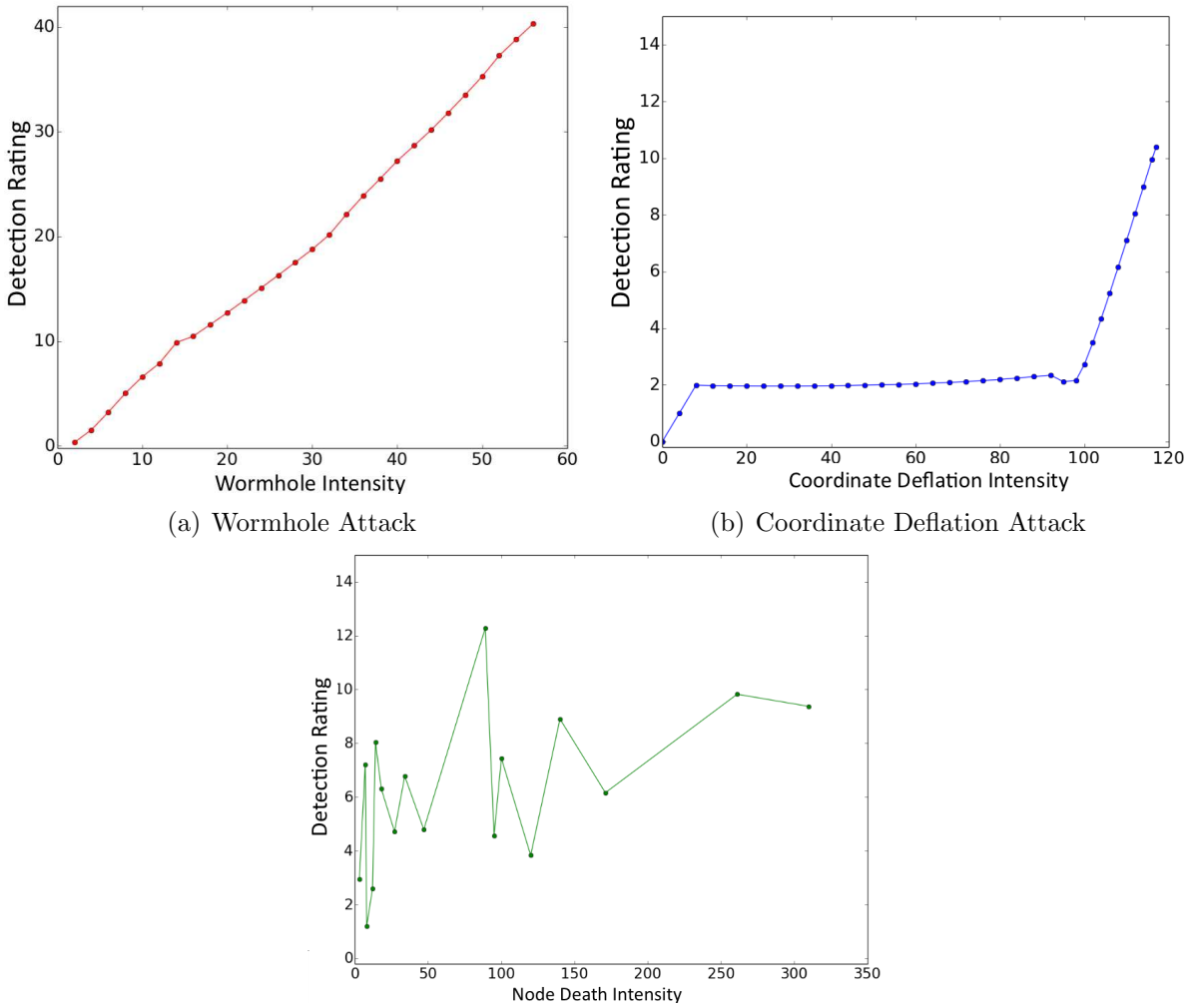


FIGURE 6.18. Average Neighbor Rating for Building (Sparse) Network

6.4.2. SPARSE NETWORK. Figure 6.18 shows the Detection Rating values for the building network. A threshold above which attacks are rated can be determined. The rating due to node deaths are varied over a wide range of values. Thus, it would be difficult to find a

threshold below which all node death ratings would lie. The death of critically connected nodes result in neighbors taking a longer path, which was earlier only 2 hops away. Thus, the shift in Topological Coordinates is higher for node deaths in Sparse Network.



(a) Wormhole Attack

(b) Coordinate Deflation Attack

(c) Node Death

FIGURE 6.19. Intensity vs DDT Detection Rating for Building (Sparse) Network

Figure 6.19 shows the performance of the DDT algorithm on the Building network with respect to the intensity of attacks. We see that the rating due to node death is very unpredictable. A high intensity Wormhole attack can be differentiated from a node death. But

Coordinate Deflation and node death ratings lie in the same range. Thus, they cannot be differentiated.

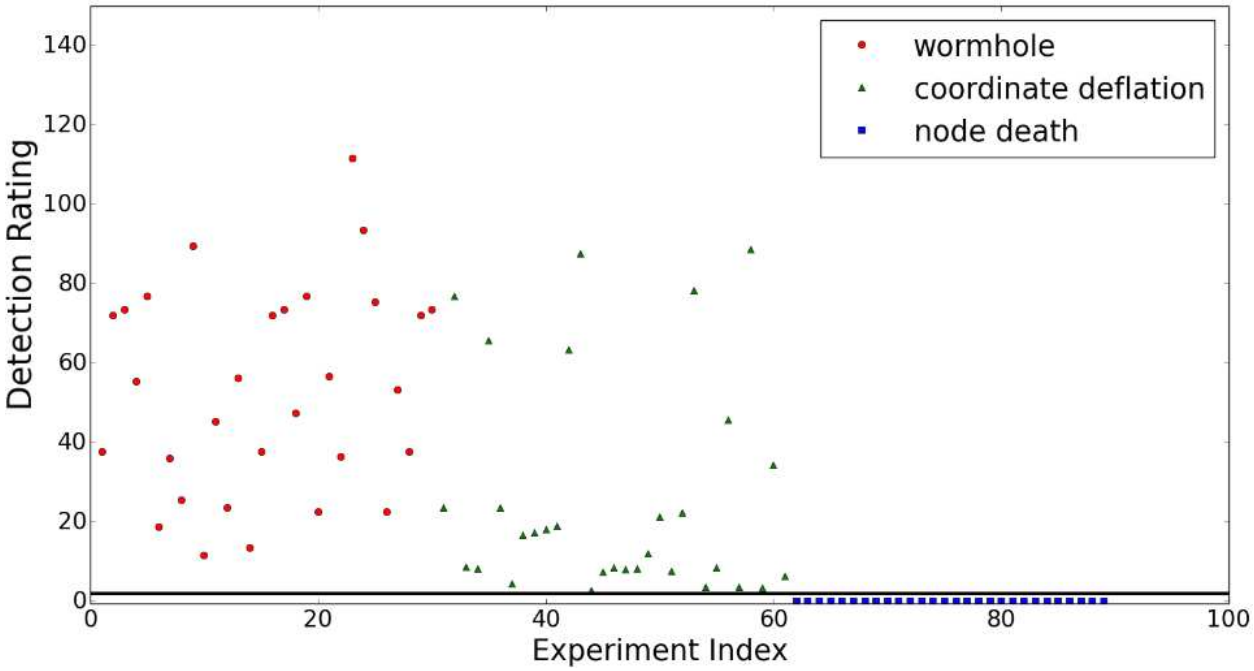
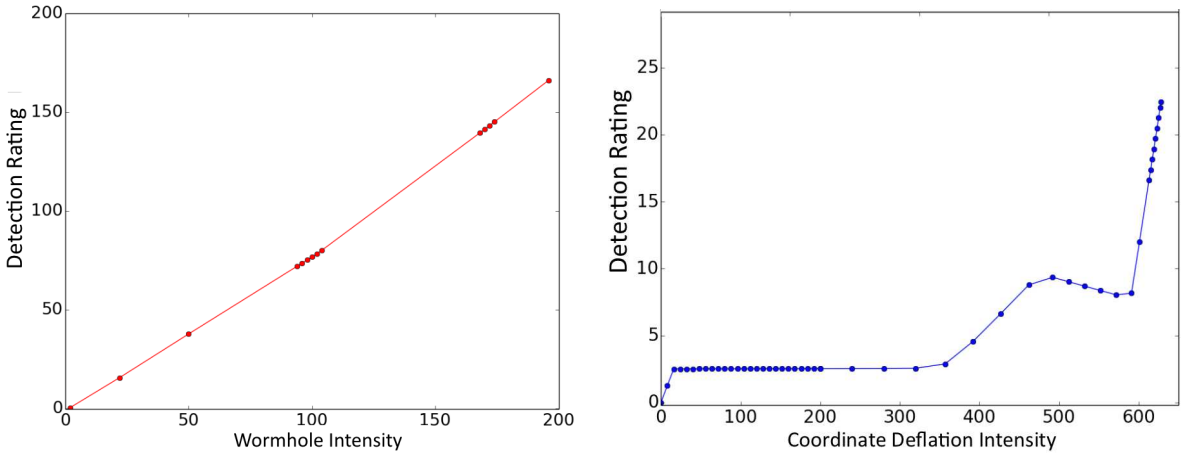


FIGURE 6.20. DDT Detection Rating for Large Network

6.4.3. LARGE NETWORK. The DDT Algorithm detects an attack by detecting change only in the neighbors. The results for random attacks and node deaths on the large network are shown in Figure 6.20. Since the connectivity of nodes is very high, even large number of node deaths, affect very few of the nodes in the network. Thus, the Detection Rating stays constant for node deaths. Thus, a defining threshold can be easily determined for large networks.

The value of the DDT Detection rating with increasing intensity of attacks is shown in Figure 6.21. The results are similar to that of the previous simulation networks. The algorithm works exceptionally well for Wormhole Attack. Although, the value of D is lesser for lower intensity of Coordinate Deflation attack, the value of D for node deaths borders

on being negligible. Thus, we deduce that the scale of network does not affect the detection capability of the algorithm and ca be successfully implemented on huge networks.

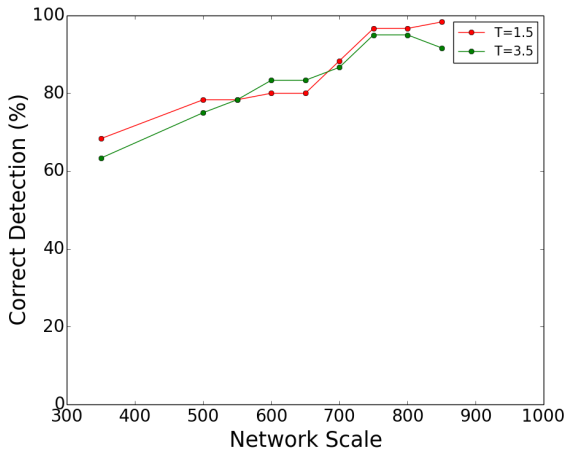


(a) Wormhole Attack (b) Coordinate Deflation Attack

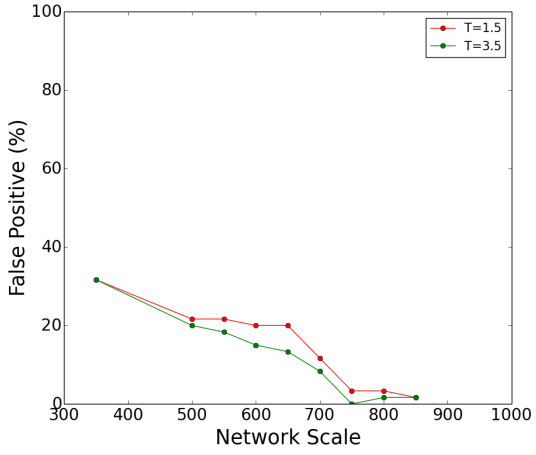
FIGURE 6.21. DDT Detection Rating vs Attack Intensity for Large Network with 10,000 Nodes

We now discuss the performance of the DDT algorithm with respect to network scale. Random networks of scale 350 nodes to 850 nodes have been generated. Percentage of Correct Detection, False Positive and Non Detection have been plotted. 20 simulations of each Coordinate Deflation, Wormhole and node death have been performed on each network of various scales.

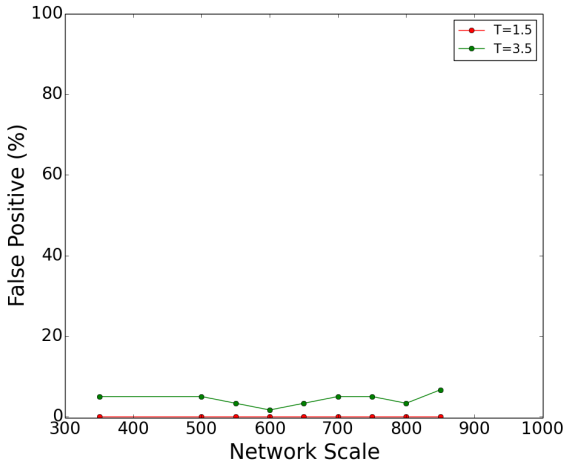
Correct Detection is considered when the DDT rating, D is greater than the threshold value, T for an attack. Also, for a node death, if the rating is below the threshold, then it is considered a Correct Detection. If the Detection rating is greater than the threshold value for a node death, it considered as a case of False Positive. If the rating is lesser than the threshold for an attack, it is termed as No Detection.



(a) Correct Detection



(b) False Positive



(c) No Detection

FIGURE 6.22. Network Scale vs DDT Detection Rating

Figure 6.22(a) shows the percentage of Correct Detection for threshold value of 1.5 and 3.5. Figure 6.22(b) shows the percentage of False Positives. The percentage of No Detection has been shown in Figure 6.22(c). From these figures, it is seen that Correct Detection of attacks and node deaths are not affected by network scale. For lesser network density, the False Positive percentage is higher since more number nodes are critical for the shape of the network. The percentage of No Detection are almost negligible for networks of all scale for both the threshold values.

The performance of the DDT algorithm on the networks presented in [27] is tabulated in Table 6.2. We see that the performance of the algorithm is very good for all dense networks. For Building Network, the percentage of Correct Detection is lower due to increased percentage of False Positives.

TABLE 6.2. Results of DDT for various networks

Type of Network	**T=2**		
	Correct Detection	**False Positive**	**Non Detection**
Block	100%	0%	0%
Circular with Voids	98.89%	1.11%	0%
M-Shaped	100%	0%	0%
Concave Void	95.98%	3.26%	0.76%
Building	68.07%	28.6%	3.33%

6.5. COMPARISON OF TECHNIQUES

In this section, we discuss the pros and cons of both the proposed algorithms.

TABLE 6.3. Nomenclature for Complexity Calculations

Nomenclature	**Definition**
N	Total number of nodes in the network
M	Total Number of anchor/monitor nodes in the network
K	Average number of nodes in each cluster

We see in previous section that both algorithms are similar in terms of efficiency in detecting attacks. However, they both differ in terms of memory requirement, computational

complexity and the number of messages. Let us define the nomenclature for our complexity analysis calculations as shown in Table 6.3.

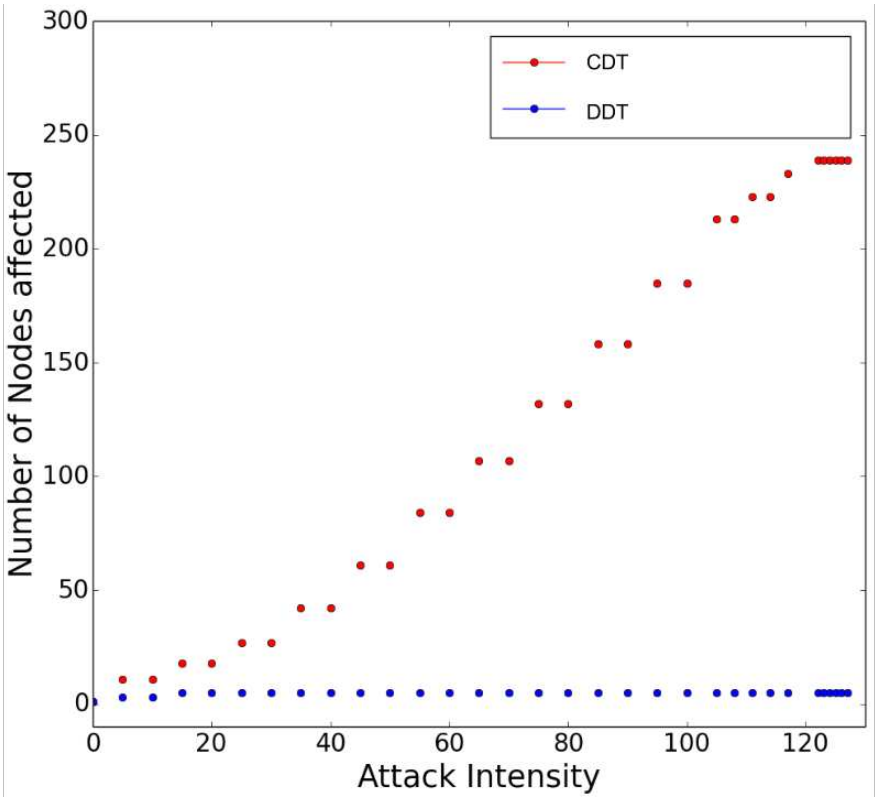


FIGURE 6.23. Number of Nodes Affected due to Coordinate Deflation

6.5.1. NUMBER OF MESSAGES. In the Centralized Detection technique, we need the changes in the VC of all the affected nodes due to an attack. On the other hand, in the Distributed Detection technique, only the VC of the neighbors of the attacked nodes get affected by the attack. Figure 6.23 compares the number of nodes that are affected with increasing intensity of a Coordinate Deflation attack. We compare the number of nodes affected in both the detection techniques.

As we see from Figure 6.23, the number of nodes affected in the CDT is very high. The number nodes affected in the DDT is constant. This is because only the neighbors of the attacked node are getting affected.

In our detection techniques, the nodes which get affected need to report their status to the anchor/monitor nodes to detect the attack. Thus, the number of messages that should be transmitted is proportional to the number of affected nodes in the network. In the worst case, where the attack is so intense, that all the nodes in the network are affected, the complexity of the number of messages in CDT would be $\mathcal{O}(N)$. The number of messages for DD is constant. We can consider this, since to affect the entire network, more than half the nodes need to be under attack. This is an unrealistic scenario, so we consider $\mathcal{O}(1)$ number of messages for the DDT. Thus, detection and mitigation of an attack is much faster in DDT.

6.5.2. Memory Cost. Now, we compare the memory requirements for both the techniques. We only show the extra memory requirements for the detection process. Memory required for storing of updated VCs are not considered as they are required for basic functionalities like routing and no additional memory is needed for detection process in respect to VC. The memory costs are summarized in Table 6.4.

Table 6.4. Comparison of Memory Cost

Type of Node	**CDT**	**DDT**
Regular Node	$\mathcal{O}(M)$	$\mathcal{O}(M) + 2\mathcal{O}(M) = \mathcal{O}(M)$
Anchor Node/Monitor Node	$\mathcal{O}(KM)$	$\mathcal{O}(1)$
Base Station	$\mathcal{O}(MN)$	-

In the Centralized Detection technique, the regular nodes need extra memory for storing the previous VCs. Thus, the memory cost for regular nodes is the number of anchor nodes M. Each anchor node stores updated VCs of its cluster's nodes to report to the base station. Thus, the memory cost for the anchor node is $\mathcal{O}(KM)$ since there will be K number of M-vectors. The base station needs to store the VC of the entire network to calculate the

centre of gravity of the entire network and that of the clusters. Thus, the cost of memory in the base station is $\mathcal{O}(MN)$.

In the Distributed Detection technique, each regular node needs to store the second and third column of the V matrix to calculate the shift in distance as defined in the Equation 4.3. It also needs to store the previous VC. Thus, the memory cost for each regular node is calculated as $\mathcal{O}(M) + \mathcal{O}(M) + \mathcal{O}(M) = \mathcal{O}(M)$. The monitor nodes gather the change in distances of the affected nodes and calculate the average change in distance. As we consider constant number of nodes getting affected in the second algorithm, the memory consumption cost for monitor nodes is considered as $\mathcal{O}(1)$ as well.

6.5.3. Computational Complexity. The computations required in the detection technique is completely done in the base station for the Centralized Detection technique. Thus, there are no additional computational overhead for the regular nodes in the first technique. The computation cost at the base station is calculated as shown in Algorithm 6.

Algorithm 6 Steps to Calculate the Computation Complexity of CDT

1: CG of entire network=$\mathcal{O}(N)$

2: CG of m clusters = $M * \mathcal{O}(K) = \mathcal{O}(KM)$

3: Calculate Detection Rating = Angle Rating+Length Rating = $2 * \mathcal{O}(M)$

4: Total Complexity = $2\mathcal{O}(KM) + \mathcal{O}(M) = \mathcal{O}(N + KM)$

In the Distributed Detection technique, each node needs to store and update the second and third column of the V matrix. It also needs to calculate the shift in distance everytime there is a change in its VCs. Calculation of *(x, y)* coordinates require multiplication of ($[1 \times M]$ and $[M \times 1]$) matrices. The computational overhead of a regular node, that is required in the second method is shown in Algorithm 7.

Algorithm 7 Steps to Calculate the Computation Complexity of DDT

1: Old x and y coordinates $= 2 * (\mathcal{O}([1 \times M] * [M \times 1])) = \mathcal{O}(2M)$

2: New x and y coordinates $= 2 * (\mathcal{O}([1 \times M] * [M \times 1])) = \mathcal{O}(2M)$

3: Distance between Old and New coordinates $= \mathcal{O}(1)$

4: Total Complexity $= \mathcal{O}(M) + \mathcal{O}(M) + \mathcal{O}(1) = \mathcal{O}(M)$

Thus, we deduce that the Distributed Detection technique requires more computation and memory in the regular nodes. Although the asymptotic analysis of memory cost is equal for both the detection techniques, for low memory devices used in WSNs, $\mathcal{O}(3M)$ may become much costlier than $\mathcal{O}(M)$. Also the computation overhead is higher for the Distributed Detection at the regular nodes. But the number of messages in the network are much higher in the Total Detection technique. This results in more transmission of data packets in the network.

6.6. Summary

In this section, we provided results for the proposed algorithms on various types networks. We deduced that the algorithms are more suited for denser networks in terms of differentiating an attack from a node death. For lower density networks, although the algorithms perform well in detecting the attacks, but they generate a lot of false positives due to node deaths. Comparison of both the algorithms was done to see the pros and cons of each algorithm. They are summarized in Table 6.5.

TABLE 6.5. Comparison of Detection Techniques

Parameter	**CDT**	**DDT**
Number of Messages	High	Constant at low value
Memory requirement of regular nodes	Only previous VC needs to be stored	Memory requirements are higher
Memory requirements of Base station	Memory to store details of entire network is needed at base station	Base station is not needed
Computational Overhead	No computations at regular node	Computation Overhead higher at regular nodes
Detection Rate	Slower	Faster

CHAPTER 7

Conclusion and Future Work

7.1. Summary and Conclusion

WSNs have tremendous potential in various environment monitoring applications. In a futuristic approach, WSNs are envisioned to be large networks with millions of nodes. The applications of WSNs are growing by the day. Although, the applications of WSNs are being well researched, the work that has been doe on its security is minimal.

In this thesis, attacks on VC based WSNs have been studied and their effects have been proven to be dangerous for the regular functionalities of WSNs. We discussed Coordinate Deflation and Wormhole attack in detail. There two attacks are chosen since they modify the VC in different patterns. Previous work provided separate algorithms for different kind of attacks. A method that can tackle all types of attack on VCs are not present. Two algorithms are presented to successfully detect VC based attacks. Our algorithms successfully detects any kind of attack that modify the Virtual Coordinates of the network.

Firstly we presented a reputation system for the VC based WSNs. Many attackers modify the detection packets which make security algorithms worthless. This is because correct information to detect a malicious node, is not received by the monitor nodes/server. The proposed reputation system, based on the Beta Reputation System, is applied to the routing of the detection packets to avoid modification of the data. The routing of the packets using the reputation values are explained with a a working example.

Next, we presented two detection techniques, which use the Topology Maps generated using SVD of the VC of the nodes. The first detection technique uses the changes in the VC of all the affected nodes in the network. A topology distortion measure is defined for this

algorithm. The second technique restricts the attack to the neighbors of the attacked node. The neighbors use the topological distance by which they themselves move due to the change in the VCs, to detect the attack. Both algorithms successfully detect Coordinate Deflation and Wormhole attacks. A defining threshold can be deduced from the simulation results which separates the detection rating value for an attack from that of node deaths. Results have been presented to evaluate the performance of the algorithms in terms of scalabilty and intensity of attacks.

We see that these algorithms can be scaled to larger networks with higher node density. For networks with lower node density, the algorithms detect the attacks, but they produce lot of false positives due to node deaths. Thus, the detection techniques are more suited to the networks with high node density.

We compare both the detection techniques. The Centralized Detection technique does not add any additional computation on the regular nodes. Also, the additional memory cost for regular nodes is minimal. However, a trusted base station is needed. A trusted communication between the anchor nodes and the base station is assumed for this technique. The Distributed Detection technique employs the regular nodes for the detection of an attack. For this, additional memory and computation overhead is put on the regular nodes. However, the Distributed Detection algorithm makes faster detection and affects lesser number of nodes in the network. The number of messages that are transmitted for the detection process is much higher in the first algorithm.

Thus, the first algorithm is suited for situations where the nodes in the network are very low power and low memory devices. The second algorithm should be employed in high integrity data networks. Both the algorithms are shown to be very effective in detecting VC based attacks.

7.2. Future Work

As an extension to this work, the application of these algorithm to 3-D networks would be highly beneficial. An algorithm which is applicable to both 2-D and 3-D networks are the need of the hour. A method to combine the reputation values in the detection process using TPM may help in faster detection with lower additional overhead. Many attackers modify the VCs of the attacked node only for a period of time and then change the VCs back to the original value. Such kind of attacks are harmful as they are difficult to detect when they behave correctly. A method to mitigate such attacks may be possible using the reputation scheme presented in this thesis. Our algorithms assume an initial benign period during which the correct VCs are generated. Removal of this assumption shall increase the robustness of the algorithms. As we have shown in previous sections, that death of a critical node brings about drastic changes to the TPM of the network. This functionality may be implemented to identify critical nodes in the network. Additional nodes may then be added to reduce dependency of such nodes.

In conclusion, this thesis proposed two powerful algorithms to capture changes in Virtual Coordinates of the nodes in WSNs using Topology Preserving Maps. These changes were measured to detect an attack on the VCs of the network.

Bibliography

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: A survey," *Comput. Netw.*, vol. 38, pp. 393–422, Mar. 2002.

[2] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon, "Health monitoring of civil infrastructures using wireless sensor networks," in *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pp. 254–263, April 2007.

[3] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, WSNA '02, (New York, NY, USA), pp. 88–97, ACM, 2002.

[4] S. Basagni, M. Conti, S. Giordano, and I. Stojmenovic, *Mobile ad hoc networking*. John Wiley & Sons, 2004.

[5] S. Khan and R. Khan, "A secure authentication and key management scheme for wireless sensor networks," in *Sensor Systems and Software*, pp. 51–60, Springer, 2014.

[6] B. Karp and H. T. Kung, "GPSR: Greedy perimeter stateless routing for wireless networks," in *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, MobiCom '00, (New York, NY, USA), pp. 243–254, ACM, 2000.

[7] A. Caruso, S. Chessa, S. De, and A. Urpi, "GPS free coordinate assignment and routing in wireless sensor networks," in *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, vol. 1, pp. 150–160 vol. 1, March 2005.

[8] Q. Cao and T. Abdelzaher, "Scalable logical coordinates framework for routing in wireless sensor networks," *ACM Trans. Sen. Netw.*, vol. 2, pp. 557–593, Nov. 2006.

[9] D. C. Dhanapala, *Anchor Centric Virtual Coordinate Systems in Wireless Sensor Networks: from Self-Organization to Network Awareness.* PhD thesis, Colorado State University, Fort Collins, Dec 2012.

[10] C. Perkins and E. Royer, "Ad-hoc on-demand distance vector routing," in *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA '99. Second IEEE Workshop on*, pp. 90–100, Feb 1999.

[11] D. Dhanapala and A. Jayasumana, "Geo-logical routing in wireless sensor networks," in *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2011 8th Annual IEEE Communications Society Conference on*, pp. 305–313, June 2011.

[12] R. Fonseca, S. Ratnasamy, J. Zhao, C. T. Ee, D. Culler, S. Shenker, and I. Stoica, "Beacon vector routing: Scalable point-to-point routing in wireless sensornets," in *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, (Berkeley, CA, USA), pp. 329–342, USENIX Association, 2005.

[13] M.-J. Tsai, H.-Y. Yang, B.-H. Liu, and W.-Q. Huang, "Virtual-coordinate-based delivery-guaranteed routing protocol in wireless sensor networks," *IEEE/ACM Transactions on Networking (TON)*, vol. 17, no. 4, pp. 1228–1241, 2009.

[14] S. Chessa, S. Escolar, S. Pelagatti, P. Baronti, and J. Carretero, "Guaranteed-delivery in arbitrary dimensional wireless sensor networks by means of recursive virtual coordinates," in *Computers and Communications (ISCC), 2012 IEEE Symposium on*, pp. 000399–000404, IEEE, 2012.

[15] J. Dong, K. E. Ackermann, B. Bavar, and C. Nita-Rotaru, "Secure and robust virtual coordinate system in wireless sensor networks," *ACM Trans. Sen. Netw.*, vol. 6, pp. 29:1–29:34, July 2010.

[16] A. Jøsang and R. Ismail, "The beta reputation system," in *In Proceedings of the 15th Bled Electronic Commerce Conference*, 2002.

[17] D. Dhanapala and A. Jayasumana, "Topology preserving maps from virtual coordinates for wireless sensor networks," in *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*, pp. 136–143, Oct 2010.

[18] A. Wood and J. Stankovic, "Denial of service in sensor networks," *Computer*, vol. 35, pp. 54–62, Oct 2002.

[19] A. Pathan, H.-W. Lee, and C. S. Hong, "Security in wireless sensor networks: issues and challenges," in *Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference*, vol. 2, pp. 6 pp.–1048, Feb 2006.

[20] J. Newsome, E. Shi, D. Song, and A. Perrig, "The sybil attack in sensor networks: Analysis & defenses," in *Proceedings of the 3rd International Symposium on Information Processing in Sensor Networks*, IPSN '04, (New York, NY, USA), pp. 259–268, ACM, 2004.

[21] Y. Yu, K. Li, W. Zhou, and P. Li, "Trust mechanisms in wireless sensor networks: Attack analysis and countermeasures," *Journal of Network and Computer Applications*, vol. 35, no. 3, pp. 867 – 880, 2012. Special Issue on Trusted Computing and Communications.

[22] Y.-C. Hu, A. Perrig, and D. Johnson, "Packet leashes: a defense against wormhole attacks in wireless networks," in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, vol. 3, pp. 1976–1986 vol.3, March 2003.

[23] W. Du, J. Deng, Y. Han, S. Chen, and P. Varshney, "A key management scheme for wireless sensor networks using deployment knowledge," in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 1, pp. –597, March 2004.

[24] A. P. R. da Silva, M. H. T. Martins, B. P. S. Rocha, A. A. F. Loureiro, L. B. Ruiz, and H. C. Wong, "Decentralized intrusion detection in wireless sensor networks," in *Proceedings of the 1st ACM International Workshop on Quality of Service &Amp; Security in Wireless and Mobile Networks*, Q2SWinet '05, (New York, NY, USA), pp. 16–23, ACM, 2005.

[25] S. Ganeriwal, L. K. Balzano, and M. B. Srivastava, "Reputation-based framework for high integrity sensor networks," *ACM Trans. Sen. Netw.*, vol. 4, pp. 15:1–15:37, June 2008.

[26] D. Dhanapala and A. Jayasumana, "Anchor selection and topology preserving maps in WSNs – 2014; a directional virtual coordinate based approach," in *Local Computer Networks (LCN), 2011 IEEE 36th Conference on*, pp. 571–579, Oct 2011.

[27] "CSU sensor-net benchmarks,." Available: `http://www.cnrl.colostate.edu/Projects/VCS/Sensor-Net.html`.

APPENDIX A

Appendix A:Source Code

Simulations for this thesis are done in Python 2.7. Package used is *networkx*. First the constant variables that are used in the code are presented:

```
NUM_OF_NODES = 0   #Total number of nodes in the network
NUM_OF_ANCHOR_NODES = 0  #Number of Anchor Nodes in the network
ANCHOR_NODES = []  #List of the Anchor nodes in the network
TYPE_OF_ATTACK=None       #This is the type of attack : {1:
   Wormhole,
#2: Coordinate Deflation, 3: Node Death}
GRAPH = None  #For the graph in Networkx
CLUSTERS_OF_NODES = []  #List of nodes in each cluster
ATTACK_INTENSITY=0       #Intensity of the Attack
```

A.1. Network Generation

```
#This function identifies the edge nodes for proper network
   arrangement
def identify_edge_nodes():
    for each_node in range(0, constant.NUM_OF_NODES):
        if(each_node == 0):
            constant.EDGE_NODES.append(each_node)
        elif(each_node%constant.NUM_OF_NODES_IN_A_ROW == 0):
            constant.EDGE_NODES.append(each_node)
        elif((each_node+1)%constant.NUM_OF_NODES_IN_A_ROW == 0)
           :
            constant.EDGE_NODES.append(each_node)
        else:
            constant.MIDDLE_NODES.append(each_node)

def create_per_row_nodes():
    first_edge = 0
    second_edge = constant.NUM_OF_NODES_IN_A_ROW-1
    row=[]
    for i in range(0, constant.NUM_OF_NODES_IN_A_ROW):
        row.append(i)
```

```
    constant.NODES_IN_ROW.append(row)
    for i in range(0, constant.NUM_OF_COLUMNS-1):
        row=[]
        first_value=first_edge+constant.NUM_OF_NODES_IN_A_ROW
        row.append(first_value)
        first_edge=first_value
        for j in range(1, constant.NUM_OF_NODES_IN_A_ROW):
            row.append(first_value+j)
        constant.NODES_IN_ROW.append(row)
    create_x_y_positions()

#This function generates the actual network
def generate_graph():
    constant.GRAPH = nx.MultiGraph()
    for i in range(0, constant.NUM_OF_NODES):
        constant.GRAPH.add_node(i)
    for each in constant.GRAPH.nodes():
        constant.GRAPH.node[each]['type'] = None
    identify_edge_nodes()
    Flag = 0
    for each_node in range(0, constant.NUM_OF_NODES-1):
        Flag = 0
        for each_edge_node in constant.EDGE_NODES:
            if int(each_node)==int(each_edge_node):
                Flag = 1
                break
        if Flag == 0:
            constant.EDGES.append((int(each_node), int(
               each_node)+1))
    odd_edges = []
    even_edges = []
    for each in constant.EDGE_NODES:
        if(each%2 == 0):
            even_edges.append(each)
        else:
            odd_edges.append(each)
    for each in even_edges:
        constant.EDGES.append((each, each+1))
    for i in range(0, len(even_edges)-1):
        constant.EDGES.append((even_edges[i], even_edges[i+1]))
    for each in odd_edges:
        constant.EDGES.append((each, each-1))
    for i in range(0, len(odd_edges)-1):
```

```
            constant.EDGES.append((odd_edges[i], odd_edges[i+1]))
    for each in constant.MIDDLE_NODES:
        if (each + constant.NUM_OF_NODES_IN_A_ROW) in
        constant.MIDDLE_NODES:
            constant.EDGES.append((each, each +
            constant.NUM_OF_NODES_IN_A_ROW))
    for each in constant.EDGES:
        add_edges_to_graph(each)

def add_edges_to_graph(edge_set):
    constant.GRAPH.add_edge(edge_set[0],edge_set[1)

#This function displays the graph using matplotlib package
def draw_display_graph(display_VC = True, save_fig = False):
    plt.figure()
    pos = nx.graphviz_layout(constant.GRAPH, prog='neato')

    color_map = {'Anchor_node':'Gold', 'Regular_node':'
       MediumBlue',
         'Changed_VC_node':'r', 'deflated' : 'Black',
         'Changed_chosen_VC' : 'r', 'Anchor_node_VC_chaged' :
         'Gold', None: 'MediumBlue'}
    nodes = nx.draw_networkx_nodes(constant.GRAPH, constant.
       POSITIONS,
          node_color=[color_map[constant.GRAPH.node[each]['type'
             ]]
          for each in constant.GRAPH.nodes()],
          node_size=[size_map[constant.GRAPH.node[each]['type']]
          for each in constant.GRAPH.nodes()])
    nodes.set_edgecolor('PaleTurquoise')
    nx.draw_networkx_edges(constant.GRAPH, constant.POSITIONS)
    plt.tick_params(axis='x', labelsize=25)
    plt.tick_params(axis='y', labelsize=25)
    labels = {}
    for each in constant.GRAPH.nodes():
        labels[each] = each
    nx.draw_networkx_labels(constant.GRAPH, constant.POSITIONS,
        labels)
    if display_VC:
        print
        for each_node in constant.GRAPH.nodes():
            x,y=constant.POSITIONS.get(each_node)
```

```
                plt.text(x, y+0.5, s=str(constant.GRAPH.node[
                   each_node]
                ['vector_cordinates_original']),
                         fontsize = 8, horizontalalignment='center'
                            )
                if constant.GRAPH.node[each_node]['type'] is
                'Changed_VC_node' or
                constant.GRAPH.node[each_node]['type'] is 'deflated
                   ' or
                constant.GRAPH.node[each_node]['type'] is
                'Changed_chosen_VC' or
                constant.GRAPH.node[each_node]['type'] is
                'Anchor_node_VC_chaged':
                    plt.text(x, y-0.5, s=str(constant.GRAPH.node[
                       each_node]
                    ['vector_cordinates_changed']), fontsize = 8,
                    color = 'red',
                    horizontalalignment='center')
    if save_fig:
        plt.savefig(fig_name)
    plt.show(block = True)

def sort_anchor_VC_tuple(current_node):
    constant.GRAPH.node[current_node]['sorted_anchor_VC'] = []
    x = create_anchor_VC_tuple(current_node)
    sorted_x = sorted(x.items(), key=operator.itemgetter(1))
    constant.GRAPH.node[current_node]['sorted_anchor_VC'] =
       sorted_x
    return sorted_x

def cluster_head_anchor(current_node):
    sorted_x = sort_anchor_VC_tuple(current_node)
    cluster_anchor = sorted_x[0][0]
    return cluster_anchor

def assign_nodes_to_clusters(anchor_node, pointer):
    lista  = []
    for each_node in constant.GRAPH.nodes():
        if constant.GRAPH.node[each_node]['type'] is '
           Regular_node':
            if constant.GRAPH.node[each_node]['
               chosen_anchor_node']
             == anchor_node:
```

```
                lista.append(each_node)
    constant.CLUSTERS_OF_NODES.append(lista)

def draw_clusters_graph():
#     plt.close()
    plt.figure()
    color = 0
    pos = nx.graphviz_layout(constant.GRAPH)
    for each_list in constant.CLUSTERS_OF_NODES:
        nodes1 = nx.draw_networkx_nodes(constant.GRAPH,
        constant.POSITIONS, nodelist=each_list, node_color=
                                constant.COLORS[color],
                                   node_size=60)
        nodes1.set_edgecolor(constant.COLORS[color])
        color = color+1
    color=0
    for each_node in constant.ANCHOR_NODES:
        nodes = nx.draw_networkx_nodes(constant.GRAPH,
        constant.POSITIONS,nodelist=[each_node], node_color=
        constant.ANCHOR_COLORS[color], node_size=150)
        color=color+1
    nx.draw_networkx_edges(constant.GRAPH, constant.POSITIONS,
    edge_color='PaleTurquoise')
     labels = {}
     for each in constant.GRAPH.nodes():
         labels[each] = each
     nx.draw_networkx_labels(constant.GRAPH, constant.POSITIONS
        , labels)
    for each_node in constant.GRAPH.nodes():
         x,y=constant.POSITIONS.get(each_node)
         plt.text(x, y+1.5, s=str(constant.GRAPH.node[each_node
            ]
         ['vector_cordinates_original']),
                    fontsize = 8, horizontalalignment='center')
    plt.savefig('Original_Clusters_Graph.png')
    plt.tick_params(axis='x', labelsize=25)
    plt.tick_params(axis='y', labelsize=25)
    plt.show(block = False)



#This function takes the node to be attacked and the
#attacked VC from the user
def coordinate_deflation_initial_inputs():
```

```
    Affected_node = int(raw_input('Enter the node number to
                create cordinate deflation : '))
    print 'original vector cordinates of ' + str(Affected_node)
        + ' is : '
    print constant.GRAPH.node[Affected_node]
    ['vector_cordinates_original']
    new_VC = []
    i=0
    while i<constant.NUM_OF_ANCHOR_NODES:
        changed_VC = raw_input('enter the changed VC : ')
        new_VC.append(int(changed_VC))
        i = i+1
    return Affected_node, new_VC

#This function simulates the coordinate deflation attack
def cordinate_deflation(Affected_node, attacked_VC):
    constant.GRAPH.node[Affected_node]['type'] = 'deflated'
    constant.GRAPH.node[Affected_node]['
       vector_cordinates_changed']
    = attacked_VC
    VC_number = 0
    while VC_number < constant.NUM_OF_ANCHOR_NODES:
        hops_from_affected_to_self = 0
        current_affected_VC = constant.GRAPH.node[Affected_node
           ]
                        ['vector_cordinates_changed'][VC_number
                           ]
        for each_node in constant.GRAPH.nodes():
            hops_from_affected_to_self = len(nx.shortest_path(
               constant.GRAPH,
                source = each_node, target = Affected_node))-1
            total_deflated_hop_count = current_affected_VC +
                hops_from_affected_to_self
            original_hop_count = constant.GRAPH.node[each_node]
                ['vector_cordinates_original'][VC_number]
            if total_deflated_hop_count < original_hop_count:
                constant.GRAPH.node[each_node]
                ['vector_cordinates_changed']
                [VC_number] = total_deflated_hop_count
            else:
                constant.GRAPH.node[each_node]
                ['vector_cordinates_changed']
                [VC_number] = constant.GRAPH.node[each_node]
```

```
                ['vector_cordinates_original'][VC_number]
            VC_number = VC_number + 1
        utility.check_type_of_nodes_after_change()
        for each_node in constant.GRAPH.nodes():
            utility.sort_anchor_VC_tuple(each_node)
        utility.draw_display_graph(display_VC = False)



#This function is called after a Wormhole has been created.
def Wormhole():
    for each_node in constant.GRAPH.nodes():
        i=0
        for each_anchor in constant.ANCHOR_NODES:
            constant.GRAPH.node[each_node]
            ['vector_cordinates_changed'][i] =
            len(nx.shortest_path(constant.GRAPH,
            source=each_node , target=each_anchor))-1
            i = i+1
    utility.check_type_of_nodes_after_change()
    utility.draw_display_graph(True, prog='dot')



def remove_nodes(node_number):
    constant.GRAPH.remove_node(node_number)
    if node_number in constant.ALL_NODES:
        constant.ALL_NODES.remove(node_number)

# Recreate VCs after node death with choice to
#recreate original VCs or changed VCs
def recreate_VCs(current_node, state):
    if state is 'original':
        del constant.GRAPH.node[current_node]
        ['vector_cordinates_original'][:]
    elif state is 'changed':
        del constant.GRAPH.node[current_node]
        ['vector_cordinates_changed'][:]
    paths = []
    for each in constant.ANCHOR_NODES:
        paths.append(nx.shortest_path(constant.GRAPH,
```

```
            source=current_node, target=each))
    if state is 'original':
        for each_path in paths:
            constant.GRAPH.node[current_node]
            ['vector_cordinates_original'].
            append(len(each_path)-1)
    elif state is 'changed':
        for each_path in paths:
            constant.GRAPH.node[current_node]
            ['vector_cordinates_changed'].
            append(len(each_path)-1)

# Create node death of the selected and change VCs of all
   affected nodes
def node_death(current_node):
    remove_nodes(current_node)
    neighbors = []
    for each_node in constant.GRAPH.nodes():
        if current_node in constant.GRAPH.node[each_node]['
           neighbors']:
            constant.GRAPH.node[each_node]['neighbors'].
            remove(current_node)
            neighbors.append(each_node)
    for each_cluster in constant.CLUSTERS_OF_NODES:
        if current_node in each_cluster:
            each_cluster.remove(current_node)
```

A.2. Generation of Topological Map

```
#Calculate the P Matrix
def calculate_node_matrix():
    list_of_list = []
    for each_node in constant.GRAPH.nodes():
        list_of_list.append(constant.GRAPH.node[each_node]
        ['vector_cordinates_original'])
    P = np.array(list_of_list)
    return P

#Get v Matrix
def calculate_SVD(P, filename='SVD/default.txt'):
```

```
    target = open(filename, 'w')
    u, s, v = np.linalg.svd(P)
    return v
# Get topological coordinates
def calculate_xSVD_ySVD(node_number, V, status='original'):
    constant.GRAPH.node[node_number]['x_y_changed']=[]
    if status is 'original' or status is 'o':
        constant.GRAPH.node[node_number]['x_y_original']=[]
        xsvd = np.dot(np.array(constant.GRAPH.node[node_number]
        ['vector_cordinates_original']), V[1,:])
        ysvd = np.dot(np.array(constant.GRAPH.node[node_number]
        ['vector_cordinates_original']), V[2,:])
        constant.GRAPH.node[node_number]['x_y_original'].append
           (xsvd)
        constant.GRAPH.node[node_number]['x_y_original'].append
           (ysvd)

    elif status is 'changed' or status is 'c':
        xsvd = np.dot(np.array(constant.GRAPH.node[node_number]
        ['vector_cordinates_changed']), V[1,:])
        ysvd = np.dot(np.array(constant.GRAPH.node[node_number]
        ['vector_cordinates_changed']), V[2,:])
        constant.GRAPH.node[node_number]['x_y_changed'].
        append(xsvd)
        constant.GRAPH.node[node_number]['x_y_changed'].
        append(ysvd)
```

A.3. Reputation System

```
# Calculate r and s values from original VC and changed VC
def calculate_r_and_s(old_VC, new_VC, rep_calculate = 'yes'):
    r=0
    s=0
    for i in range(0, len(old_VC)):
        if(old_VC[i]==new_VC[i]):
            r=r+1
        else:
            difference = math.fabs(old_VC[i]-new_VC[i])
            s= s+difference
    if(rep_calculate is 'yes'):
```

```
        return calculate_rep_rating(r, s)
    else:
        return str(r)+" "+str(s)

#Function for Reputation Rating
def calculate_rep_rating(new_r, new_s, old_r=0, old_s=0):
    num = old_r+new_r
    den = old_r+new_r+old_s+new_s
    if num == 0 and den ==0:
        rep_rating =0
    else:
        rep_rating = num/den
    return round(rep_rating,2)

#Calcuate the Reputation Rating for neighbours
def for_all_nodes_neighbor_rep():
    for each_node in constant.GRAPH.nodes():
        constant.GRAPH.node[each_node]
        ['Neighbor_Reputation'] = {}
        constant.GRAPH.node[each_node]
        ['Neighbor_Reputation_discounted'] = {}
        for each_neighbor in constant.GRAPH.node[each_node]
        ['neighbors']:
            constant.GRAPH.node[each_node]['Neighbor_Reputation
               ']
            [each_neighbor]=calculate_r_and_s(constant.GRAPH.
               node
            [each_neighbor]['vector_cordinates_original'],
            constant.GRAPH.node[each_neighbor]
            ['vector_cordinates_changed'],
            sorted_anchor_positions(each_node))

            r_s = calculate_r_and_s_with_weights(constant.GRAPH
               .
                node[each_neighbor]['vector_cordinates_original
                   '],
                constant.GRAPH.node[each_neighbor]
                ['vector_cordinates_changed'],
                sorted_anchor_positions(each_node)')
            reg = r_s.split(' ')
            r = float(reg[0])
            s = float(reg[1])
            constant.GRAPH.node[each_node]
```

```
            ['Neighbor_Reputation_discounted']
                [each_neighbor] =
                calculate_discounted_r_and_s(constant.GRAPH.
                node[each_node]['r_s_self'][0],
                constant.GRAPH.node[each_node]['r_s_self'][1],
                   r, s)
```

A.4. Total Detection Rating

```
def create_clusterwise_x_y_positions(status='o'):
    i=0
    for each_list in constant.CLUSTERS_OF_NODES:
        x_pos=[]
        y_pos=[]
        if status is 'o':
            x_pos.append(constant.GRAPH.node[constant.
               ANCHOR_NODES[i]]
            ['x_y_original'][0])
            y_pos.append(constant.GRAPH.node[constant.
               ANCHOR_NODES[i]]
            ['x_y_original'][1])
        elif status is 'c':
            x_pos.append(constant.GRAPH.node[constant.
               ANCHOR_NODES[i]]
            ['x_y_changed'][0])
            y_pos.append(constant.GRAPH.node[constant.
               ANCHOR_NODES[i]]
            ['x_y_changed'][1])
        i=i+1
        for each_node in each_list:
            if status is 'o':
                x_pos.append(constant.GRAPH.node[each_node]
                ['x_y_original'][0])
                y_pos.append(constant.GRAPH.node[each_node]
                ['x_y_original'][1])
            elif status is 'c':
                x_pos.append(constant.GRAPH.node[each_node]
                ['x_y_changed'][0])
                y_pos.append(constant.GRAPH.node[each_node]
                ['x_y_changed'][1])
```

```
        if status is 'o':
            constant.X_POSITIONS_OF_CLUSTERS_ORIGINAL.append(
               x_pos)
            constant.Y_POSITIONS_OF_CLUSTERS_ORIGINAL.append(
               y_pos)
        elif status is 'c':
            constant.X_POSITIONS_OF_CLUSTERS_CHANGED.append(
               x_pos)
            constant.Y_POSITIONS_OF_CLUSTERS_CHANGED.append(
               y_pos)

#Centre of Gravity of clusters
def mean_x_y_position_cluster(status='o'):
    if status is 'o':
        for each_list in constant.
           X_POSITIONS_OF_CLUSTERS_ORIGINAL:
            sum_val=0
            for i in range(0, len(each_list)):
                sum_val=sum_val+each_list[i]
            mean_val=sum_val/len(each_list)
            constant.X_CG_CLUSTERS_ORIGINAL.append(mean_val)
        for each_list in constant.
           Y_POSITIONS_OF_CLUSTERS_ORIGINAL:
            sum_val=0
            for i in range(0, len(each_list)):
                sum_val=sum_val+each_list[i]
            mean_val=sum_val/len(each_list)
            constant.Y_CG_CLUSTERS_ORIGINAL.append(mean_val)
    elif status is 'c':
        for each_list in constant.
           X_POSITIONS_OF_CLUSTERS_CHANGED:
            sum_val=0
            for i in range(0, len(each_list)):
                sum_val=sum_val+each_list[i]
            mean_val=sum_val/len(each_list)
            constant.X_CG_CLUSTERS_CHANGED.append(mean_val)
        for each_list in constant.
           Y_POSITIONS_OF_CLUSTERS_CHANGED:
            sum_val=0
            for i in range(0, len(each_list)):
                sum_val=sum_val+each_list[i]
            mean_val=sum_val/len(each_list)
            constant.Y_CG_CLUSTERS_CHANGED.append(mean_val)
```

```
#Centre of Gravity of the network
def mean_x_y_position_of_network(status='o'):
    if status is 'o':
        sum_val_x=0
        sum_val_y=0
        for each_node in constant.GRAPH.nodes():
            sum_val_x=sum_val_x+ constant.GRAPH.node[each_node]
            ['x_y_original'][0]
            sum_val_y=sum_val_y+ constant.GRAPH.node[each_node]
            ['x_y_original'][1]
        constant.X_Y_CG_FULL.append(sum_val_x/constant.
           NUM_OF_NODES)
        constant.X_Y_CG_FULL.append(sum_val_y/constant.
           NUM_OF_NODES)

#Class file for each cluster rating

class CG(object):
    def __init__(self, cluster_id):
        self.cluster_id=cluster_id
        # goes from 0 to number of anchor nodes
        self.anchor_node=0
        self.original_x=0
        self.original_y=0
        self.length_original=0
        self.angle_x_axis=0
        self.slope=0.0
        self.changed_x=0
        self.changed_y=0
        self.is_cluster_under_attack=False
        self.length_changed=0
        self.adjacent_neighbors=[]
        self.adjacent_angles_original=[]
        self.adjacent_angles_changed=[]
        self.angle_rating=0
        self.length_rating=0
        self.total_rating=0
        self.std_dev_original=0
        self.std_dev_changed=0
        self.change_in_std_dev=0

    def angle_from_x_axis(self):
```

```
        self.angle_x_axis = math.degrees(math.atan2(self.
            original_y,
        self.original_x))%360

    def assign_anchor_node(self):
        self.anchor_node=constant.ANCHOR_NODES[self.cluster_id]

    def get_x_y_positions(self, state='o'):
        if state is 'o':
            self.original_x=constant.X_CG_CLUSTERS_ORIGINAL
            [self.cluster_id]
            self.original_y=constant.Y_CG_CLUSTERS_ORIGINAL
            [self.cluster_id]
        elif state is 'c':
            self.changed_x=constant.X_CG_CLUSTERS_CHANGED
            [self.cluster_id]
            self.changed_y=constant.Y_CG_CLUSTERS_CHANGED
            [self.cluster_id]

    def get_self_length(self, state='o'):
        if state is 'o':
            self.length_original=SVD.
                calculate_topological_distance
            (constant.X_Y_CG_FULL, [self.original_x, self.
                original_y])
        if state is 'c':
            self.length_changed=SVD.
                calculate_topological_distance
            (constant.X_Y_CG_FULL, [self.changed_x, self.
                changed_y])

    def get_slope(self):
        self.slope=(self.original_y-constant.X_Y_CG_FULL[1])/
        (self.original_x-constant.X_Y_CG_FULL[0])

    def find_adjacent_CG(self):
        sorted_positions=np.argsort(constant.
            ANGLE_OF_CG_WITH_X_AXIS).
        tolist()
        adj_nighbors=[]
        i=0
        for each in sorted_positions:
            if each is self.cluster_id:
```

```
                if i==0:
                    adj_nighbors.append(sorted_positions[i+1])
                    adj_nighbors.append(sorted_positions
                    [len(sorted_positions)-1])
                elif i==len(sorted_positions)-1:
                    adj_nighbors.append(sorted_positions[i-1])
                    adj_nighbors.append(sorted_positions[0])
                else:
                    adj_nighbors.append(sorted_positions[i-1])
                    adj_nighbors.append(sorted_positions[i+1])
            i=i+1
        for each_id in adj_nighbors:
            for each_cg in constant.CLUSTERS:
                if each_cg.cluster_id == each_id:
                    self.adjacent_neighbors.append(each_cg)
                    break

    def adjacent_angles(self, state='o'):
        if state is 'o':
            for each_neighbor in self.adjacent_neighbors:
                self.adjacent_angles_original.append
                (utility.angle_between_3_points (constant.
                   X_Y_CG_FULL,
                [self.original_x, self.original_y],
                [each_neighbor.original_x, each_neighbor.
                   original_y]))

        elif state is 'c':
            for each_neighbor in self.adjacent_neighbors:
                self.adjacent_angles_changed.append
                (utility.angle_between_3_points(constant.
                   X_Y_CG_FULL,
                 [self.changed_x, self.changed_y],
                 [each_neighbor.changed_x, each_neighbor.
                    changed_y]))

    def angle_length_rating(self):
        rate1=(math.fabs(self.adjacent_angles_original[0]-
        self.adjacent_angles_changed[0]))/self.
           adjacent_angles_original[0]
        rate2=(math.fabs(self.adjacent_angles_original[1]-
        self.adjacent_angles_changed[1]))/self.
           adjacent_angles_original[1]
```

```
        self.angle_rating=(rate1+rate2)/2
        self.length_rating=(math.fabs(self.length_original-
        self.length_changed))/self.length_original
        self.total_rating=self.angle_rating+self.length_rating

def create_cluster(cluster_id):
    cluster=CG(cluster_id)
    cluster.assign_anchor_node()
    cluster.get_x_y_positions()
    cluster.angle_from_x_axis()
    cluster.get_self_length()
    cluster.get_slope()
    return cluster

def total_network_rating():
    network_rating=0
    for each in constant.CLUSTERS:
        network_rating=network_rating+each.total_rating
    constant.NETWORK_RATING=network_rating
    /constant.NUM_OF_ANCHOR_NODES
    print 'total network rating : ' + str(constant.
       NETWORK_RATING)
```

A.5. Neighbor Detection

```
def coordinate_deflation_initial_inputs(tokens=None):
    if tokens is None:
        Affected_node = int(raw_input('Enter the node number to
            create cordinate deflation : '))
        print 'original vector cordinates of ' + str(
           Affected_node) + ' is : '
        print constant.GRAPH.node[Affected_node]['
           vector_cordinates_original']
        new_VC = []
        i=0
        while i<constant.NUM_OF_ANCHOR_NODES:
            changed_VC = raw_input('enter the changed VC : ')
            new_VC.append(int(changed_VC))
```

```
            i = i+1
        return Affected_node, new_VC
    else:
        Affected_node = int(tokens[1])
        print 'original vector cordinates of ' + str(
           Affected_node) + ' is : '
        print constant.GRAPH.node[Affected_node]['
           vector_cordinates_original']
        new_VC = []
        for i in range(2,len(tokens)):
            changed_VC=tokens[i]
            new_VC.append(int(changed_VC))
        return Affected_node, new_VC

# This function changes the VCs of only the neighbors of the
   attacked node
def coordinate_deflation_in_neighbors(tokens=None):
    Affected_node, attacked_VC=
       coordinate_deflation_initial_inputs(tokens)
    constant.GRAPH.node[Affected_node]['type'] = 'deflated'
    constant.GRAPH.node[Affected_node]['
       vector_cordinates_changed'] =
     attacked_VC
    size=0
    for i in range(0, constant.NUM_OF_ANCHOR_NODES):
        size=size+math.fabs(constant.GRAPH.node[Affected_node]
        ['vector_cordinates_original'][i]-constant.GRAPH.node[
           Affected_node]
        ['vector_cordinates_changed'][i])
    constant.ATTACK_INTENSITY=size
    VC_number = 0
    while VC_number < constant.NUM_OF_ANCHOR_NODES:
        hops_from_affected_to_self = 0
        current_affected_VC = constant.GRAPH.node[Affected_node
           ]
        ['vector_cordinates_changed'][VC_number]
        for each in constant.GRAPH.node[Affected_node]['
           neighbors']:
            if constant.GRAPH.node[each]['type'] == '
               Regular_node':
                hops_from_affected_to_self = len(nx.
                   shortest_path
```

```
                (constant.GRAPH, source = each, target =
                   Affected_node))-1
                total_deflated_hop_count = current_affected_VC
                   +
                hops_from_affected_to_self
                original_hop_count = constant.GRAPH.node[each]
                ['vector_cordinates_original'][VC_number]
                if total_deflated_hop_count <
                   original_hop_count:
                    constant.GRAPH.node[each]['
                       vector_cordinates_changed']
                    [VC_number] = total_deflated_hop_count
                else:
                    constant.GRAPH.node[each]['
                       vector_cordinates_changed']
                    [VC_number] = constant.GRAPH.node[each]
                    ['vector_cordinates_original'][VC_number]
            elif constant.GRAPH.node[each]['type'] == '
               Anchor_node':
                constant.GRAPH.node[each]['
                   vector_cordinates_changed'] =
                constant.GRAPH.node[each]['
                   vector_cordinates_original']
        VC_number = VC_number+1
    utility.check_type_of_nodes_after_change()
    for each_node in constant.GRAPH.nodes():
        utility.sort_anchor_VC_tuple(each_node)


def Wormhole_in_neighbors(first_node, second_node):
    i=0
    for each_anchor in constant.ANCHOR_NODES:
            constant.GRAPH.node[first_node]
            ['vector_cordinates_changed'][i] = len(nx.
               shortest_path
            (constant.GRAPH, source=first_node, target=
               each_anchor))-1
            i=i+1
    i=0
    for each_anchor in constant.ANCHOR_NODES:
            constant.GRAPH.node[second_node]
            ['vector_cordinates_changed'][i] = len(nx.
               shortest_path
```

```
            (constant.GRAPH, source=second_node, target=
               each_anchor))-1
            i=i+1
    for each_neighbor in constant.GRAPH.node[first_node]['
       neighbors']:
        i=0
        for each_anchor in constant.ANCHOR_NODES:
            constant.GRAPH.node[each_neighbor]
            ['vector_cordinates_changed'][i] = len(nx.
               shortest_path
            (constant.GRAPH, source=each_neighbor, target=
               each_anchor))-1
            i = i+1
    for each_neighbor in constant.GRAPH.node[second_node]['
       neighbors']:
        i=0
        for each_anchor in constant.ANCHOR_NODES:
            constant.GRAPH.node[each_neighbor]
            ['vector_cordinates_changed'][i] = len(nx.
               shortest_path
            (constant.GRAPH, source=each_neighbor, target=
               each_anchor))-1
            i = i+1
    utility.check_type_of_nodes_after_change()
    constant.GRAPH.node[first_node]['type']='deflated'
    constant.GRAPH.node[second_node]['type']='deflated'



def node_death_and_VC_of_neighbors_affected(current_node):
    neighbors = []
    neighbors = constant.GRAPH.node[current_node]['neighbors']
    for each_neighnor in neighbors:
        constant.NEIGHBORS_OF_DEAD_NODE.append(each_neighnor)
    utility.remove_nodes(current_node)
    if current_node in constant.NEIGHBORS_OF_DEAD_NODE:
        constant.NEIGHBORS_OF_DEAD_NODE.remove(current_node)
    for each_node in constant.GRAPH.nodes():
        if current_node in constant.GRAPH.node[each_node]['
           neighbors']:
            constant.GRAPH.node[each_node]['neighbors'].
            remove(current_node)
```

```
    for each_cluster in constant.CLUSTERS_OF_NODES:
        if current_node in each_cluster:
            each_cluster.remove(current_node)
    for each_node in neighbors:
        utility.recreate_VCs(each_node, 'changed')




def distance_between_nodes(first_node_num, first_node_status,
second_node_num, second_node_status):
    if first_node_status is 'o' and second_node_status is 'o':
        return SVD.calculate_topological_distance(constant.
        GRAPH.node[first_node_num]['x_y_original'],
        constant.GRAPH.node[second_node_num]['x_y_original'])
    elif first_node_status is 'o' and second_node_status is 'c'
       :
        return SVD.calculate_topological_distance(constant.
        GRAPH.node[first_node_num]['x_y_original'],
        constant.GRAPH.node[second_node_num]['x_y_changed'])
    elif first_node_status is 'c' and second_node_status is 'o'
       :
        return SVD.calculate_topological_distance(constant.
        GRAPH.node[first_node_num]['x_y_changed'],
        constant.GRAPH.node[second_node_num]['x_y_original'])
    elif first_node_status is 'c' and second_node_status is 'c'
       :
        return SVD.calculate_topological_distance(constant.
        GRAPH.node[first_node_num]['x_y_changed'],
        constant.GRAPH.node[second_node_num]['x_y_changed'])

def get_list_neighbor_changed_position():
    changed_neighbors=[]
    changed_distances_of_neighbors=[]
    if constant.TYPE_OF_ATTACK == 3:
            for each_neighbor in constant.
               NEIGHBORS_OF_DEAD_NODE:
                changed_neighbors.append(each_neighbor)
    else:
        for each in constant.GRAPH.nodes():
            if constant.GRAPH.node[each]['type'] is 'deflated':
                for each_neighbor in constant.GRAPH.node[each][
                   'neighbors']:
```

```
                    changed_neighbors.append(each_neighbor)
    for each_n in changed_neighbors:
        change_in_position=distance_between_nodes(each_n, 'o',
           each_n, 'c')
        changed_distances_of_neighbors.append(
           change_in_position)
    return changed_distances_of_neighbors

def get_average_rating():
    changed_distances=get_list_neighbor_changed_position()
    avg=0
    for each in changed_distances:
        avg=avg+each
    avg=avg/len(changed_distances)
    return avg




def take_inital_inputs_from_file(input_file):
    fo=open(input_file, "rw+")
    constant.NUM_OF_NODES = int(fo.readline())
    constant.NUM_OF_NODES_IN_A_ROW = int(fo.readline())
    constant.NUM_OF_COLUMNS = int(constant.NUM_OF_NODES/
       constant.NUM_OF_NODES_IN_A_ROW)
    utility.create_per_row_nodes()
    utility.generate_graph()
    num_of_nodes_to_remove = int(fo.readline())
    nodes_to_be_removed = []
    for i in range(0, num_of_nodes_to_remove):
        nodes_to_be_removed.append(int(fo.readline()))
    constant.NUM_OF_NODES = constant.NUM_OF_NODES -
    len(nodes_to_be_removed)
    for each in nodes_to_be_removed:
        utility.remove_nodes(each)
#     utility.draw_display_graph(display_VC = False, save_fig =
    False)
    constant.NUM_OF_ANCHOR_NODES = int(fo.readline())
    selection = fo.readline()
    if selection is 'y':
        utility.random_anchor_selection()
    else:
        for i in range(0, constant.NUM_OF_ANCHOR_NODES):
```

```
            constant.ANCHOR_NODES.append(int(fo.readline()))

    constant.ANCHOR_NODES.sort()
    constant.ALL_NODES = [None] * constant.NUM_OF_NODES
    constant.WEIGHT_FACTOR = float(fo.readline())

def take_initial_inputs_by_terminal():
    constant.NUM_OF_NODES = int(raw_input("Enter the number of
       nodes : "))
    constant.NUM_OF_NODES_IN_A_ROW = int(raw_input("Enter
       number
     of nodes in row : "))
    constant.NUM_OF_COLUMNS = int(constant.NUM_OF_NODES/
       constant.NUM_OF_NODES_IN_A_ROW)
    utility.create_per_row_nodes()
    utility.generate_graph()
    utility.draw_display_graph(False)
    num_of_nodes_to_remove = int(raw_input("Enter the
    number of nodes you want to remove : "))
    nodes_to_be_removed = []
    for i in range(0, num_of_nodes_to_remove):
        nodes_to_be_removed.append(int(raw_input("Node to be
           removed : ")))
    constant.NUM_OF_NODES = constant.NUM_OF_NODES -
    len(nodes_to_be_removed)
    for each in nodes_to_be_removed:
        utility.remove_nodes(each)
    utility.draw_display_graph(False)
    constant.NUM_OF_ANCHOR_NODES = int(raw_input("Enter number
       of Anchor Nodes : "))
    selection = raw_input('Random Anchor selection ? (y/n : )')
    if selection is 'y':
        utility.random_anchor_selection()
    else:
        for i in range(0, constant.NUM_OF_ANCHOR_NODES):
            constant.ANCHOR_NODES.append(int(raw_input('Anchor
               Node : ')))
    constant.ANCHOR_NODES.sort()
    constant.ALL_NODES = [None] * constant.NUM_OF_NODES
    constant.WEIGHT_FACTOR = float(raw_input('Enter the
       weightage factor : '))

def assign_values_to_nodes():
```

```
    i = 0
    for each in constant.GRAPH.nodes():
        constant.ALL_NODES[i] = node.make_node(int(each))
        constant.GRAPH.node[each]['vector_cordinates_original']
            = constant.ALL_NODES[i].vector_cordinates_original
        constant.GRAPH.node[each]['vector_cordinates_changed']
           = []
        for j in range(0, constant.NUM_OF_ANCHOR_NODES):
            constant.GRAPH.node[each]['
               vector_cordinates_changed'].
            append(constant.GRAPH.node[each]['
               vector_cordinates_original'][j])
        constant.GRAPH.node[each]['node_id'] = constant.
           ALL_NODES[i].node_id
        constant.GRAPH.node[each]['type'] = constant.ALL_NODES[
           i].type
        constant.GRAPH.node[each]['paths'] = constant.ALL_NODES
           [i].paths
        chosen_anchor_node = constant.ANCHOR_NODES[utility.
           cluster_head_anchor
        (each)]
        constant.GRAPH.node[each]['chosen_anchor_node'] =
           chosen_anchor_node
        i = i+1
    for each in constant.GRAPH.nodes():
        constant.GRAPH.node[each]['neighbors'] = []
        constant.GRAPH.node[each]['neighbors'] = constant.GRAPH
           .neighbors(each)
        constant.SELF_REPUTATION_DATASET.update({each:[]})
    for each in constant.GRAPH.nodes():
        utility.diff_and_add_2nd_degree_VC(each)
#    beta_reputation.create_sheet()
#    beta_reputation.for_all_nodes_self_rep(constant.
   WEIGHT_FACTOR)

#    beta_reputation.neighbor_rep('all')
    #beta_reputation.write_to_sheet()
    #beta_reputation.write_to_sheet('with_discounting')

def assign_nodes_to_clusters():
    pointer = 0
    for each in constant.ANCHOR_NODES:
        utility.assign_nodes_to_clusters(each, pointer)
```

```
        pointer = pointer+1
    for anchor_node in constant.ANCHOR_NODES:
        dictionary_for_anchor_node_details(anchor_node)
#     utility.draw_clusters_graph()
    for each in constant.ANCHOR_NODES:
        constant.AVG_SELF_REP_DATASET.update({each:[]})
    constant.X_VALUES.append('original')
#    beta_reputation.avg_self_rep_cluster()

def dictionary_for_anchor_node_details(anchor_node):
    constant.GRAPH.node[anchor_node]['anchor_node_details'] =
       {}
    position = utility.find_anchor_node_position(anchor_node)
    constant.GRAPH.node[anchor_node]['anchor_node_details'].
    update({'position' : position})
    constant.GRAPH.node[anchor_node]['anchor_node_details'].
    update({'nodes_in_cluster' : constant.CLUSTERS_OF_NODES[
       position]})

def process_request(request):
    if request is 'print':
        node_number = raw_input('Enter the node number')
        print 'vector_cordinates_original'
        print 'vector_cordinates_changed'
        print 'type'
        print 'chosen_anchor_node'
        print 'neighbors'
        print 'Self_Reputation'
        print 'r_s_self'
        print 'Neighbor_Reputation'
        print 'Neighbor_Reputation_discounted'
        category = raw_input('Enter category:')
        if(node_number is 'all'):
            for each in constant.GRAPH.nodes():
                print each
                print ""
                print constant.GRAPH.node[each][category]
        else:
            print constant.GRAPH.node[int(node_number)][
               category]

def print_input_values():
    print 'Type "print" for details'
```

```
    print 'The kind of attacks:'
    print '1: Wormhole'
    print '2: Cordinate Deflation'
    print '3: Make a node die'
    print '4: Plot Graph'
    print '5: Calculate Area'
    print '6: Calculate Shortest Distance'
    print '7: Calculate topological xy coordinates'
    print '8: Plot XY coordinates'
    print '9: Calculate topological distance'
    print '10: Write SVDs to file'

#The main function
def main():
    initial_inputs=sys.argv[1]
    threshold_value=0.3
    if initial_inputs is 'f':
        file = sys.argv[2]
        take_inital_inputs_from_file(file)
    else:
        take_initial_inputs_by_terminal()
    file_name='network_rating.csv'
    assign_values_to_nodes()
    utility.draw_display_graph(display_VC=True)
    assign_nodes_to_clusters()
    P=SVD.calculate_node_matrix()
    V = SVD.calculate_SVD(P)
    for each_node in constant.GRAPH.nodes():
        SVD.calculate_xSVD_ySVD(each_node, V)
    for i in range(0,constant.NUM_OF_ANCHOR_NODES):
        constant.MEAN_OF_CLUSTERS.append(SVD.
           calculate_cluster_mean(i))
    utilities_cg.create_clusterwise_x_y_positions('o')
    utilities_cg.mean_x_y_position_cluster('o')
    utilities_cg.mean_x_y_position_of_network()
    utilities_cg.create_CG_details()
    constant.LENGTH_OF_CLUSTERS_FOR_DETECTION_2_CHANGED=[[] for
        i in range(constant.NUM_OF_ANCHOR_NODES)]
    constant.LENGTH_OF_CLUSTERS_FOR_DETECTION_2_ORIGINAL=[[]
       for i in range(constant.NUM_OF_ANCHOR_NODES)]
    distributed_cluster_heads.calculate_std_dev_of_clusters()
    f=open('simulations/simulation.txt','r')
    curr_string=f.readline()
```

```
list_of_lines=curr_string.split(":")
utilities_cg.delete_first_line_from_file('simulations/
   simulation.txt')
while(1):
    size=0
    print_input_values()
    kind_of_attack = raw_input('Enter the kind of attack to
        mount: ')
    kind_of_attack=list_of_lines[0]
    if kind_of_attack =='print':
        process_request('print')

    elif kind_of_attack == 'Wormhole' or kind_of_attack ==
       '1':
         first_node=8
         first_node = int(raw_input('Enter the first node
            to make
         a new connection : '))
        first_node=list_of_lines[1]
        print 'first node : '+ str(first_node)
        constant.GRAPH.node[int(first_node)]['type']='
           deflated'
         second_node = int(raw_input('Enter the second node
             : '))
        second_node=list_of_lines[2]
        print 'second node : '+ str(second_node)
        constant.GRAPH.node[int(second_node)]['type']='
           deflated'
        constant.TYPE_OF_ATTACK=1
        path=nx.shortest_path(constant.GRAPH, source=int(
           first_node),
        target=int(second_node))
        size=len(path)-1
        constant.ATTACK_INTENSITY=size
        utility.add_edges_to_graph([int(first_node), int(
           second_node)])
        attack.Wormhole()
        n_attack.Wormhole_in_neighbors(int(first_node), int
           (second_node))
        constant.INTENSITY_OF_ATTACK= utility.
        calculate_percentage_of_nodes_afected()
        print '% of nodes affected : ' + str(constant.
           INTENSITY_OF_ATTACK)
```

```
            beta_reputation.for_all_nodes_self_rep(constant.
                WEIGHT_FACTOR)
            for each_node in constant.GRAPH.nodes():
                if constant.GRAPH.node[each_node]['type'] is '
                    Changed_chosen_VC' or constant.GRAPH.node[
                    each_node]['type'] is 'Changed_VC_node':
                                print each_node
                     print constant.GRAPH.node[each_node]['
                         Self_Reputation']
            constant.X_VALUES.append('Wormhole')
            beta_reputation.neighbor_rep('all')
            beta_reputation.write_to_sheet()
            for each in constant.GRAPH.nodes():
                SVD.calculate_xSVD_ySVD(each, V, 'c')
            n_detection.write_neighbor_distance_changed()
            utilities_cg.create_clusterwise_x_y_positions('c')
            utilities_cg.mean_x_y_position_cluster('c')
            utilities_cg.CG_details_changed()
            utilities_cg.total_network_rating()
            distributed_cluster_heads.
                calculate_std_dev_of_clusters('c')
            distributed_cluster_heads.change_in_self_dev()
            utilities_cg.append_csv_file(file_name)
            utilities_cg.
                append_csv_file_with_intensity_of_attack(
                file_name)
            utilities_cg.write_to_file_cluster_wise()
            utilities_cg.write_to_file_std_dev()
            utilities_cg.write_to_file_avg_std_dev()

        elif kind_of_attack == 'Cordinate Deflation' or
            kind_of_attack == '2':
            constant.TYPE_OF_ATTACK=2
            Affected_node, new_VC = attack.
                coordinate_deflation_initial_inputs(
                list_of_lines)
            Affected_node, new_VC = attack.
                coordinate_deflation_initial_inputs()
            attack.cordinate_deflation(Affected_node, new_VC)
            for i in range(0, constant.NUM_OF_ANCHOR_NODES):
                size=size+math.fabs(constant.GRAPH.node[
                    Affected_node]
```

```
            ['vector_cordinates_original'][i]-constant.
               GRAPH.node[Affected_node]['
               vector_cordinates_changed'][i])
        constant.ATTACK_INTENSITY=size
        n_attack.coordinate_deflation_in_neighbors()
      beta_reputation.for_all_nodes_self_rep(constant.
         WEIGHT_FACTOR)
      for each_node in constant.GRAPH.nodes():
           if constant.GRAPH.node[each_node]['type'] is '
              Changed_chosen_VC' or constant.GRAPH.node[
              each_node]['type'] is 'Changed_VC_node':
        constant.X_VALUES.append('coordinate_deflation')
       beta_reputation.neighbor_rep('all')
       beta_reputation.write_to_sheet()
       beta_reputation.write_to_sheet('with_discounting')
        for each in constant.GRAPH.nodes():
            SVD.calculate_xSVD_ySVD(each, V, 'c')
       n_detection.write_neighbor_distance_changed()
       utilities_cg.create_clusterwise_x_y_positions('c')
       utilities_cg.mean_x_y_position_cluster('c')
       utilities_cg.CG_details_changed()
       utilities_cg.total_network_rating()
       distributed_cluster_heads.
          calculate_std_dev_of_clusters('c')
       distributed_cluster_heads.change_in_self_dev()
       utilities_cg.append_csv_file(file_name)
       utilities_cg.
          append_csv_file_with_intensity_of_attack(
          file_name)
       utilities_cg.write_to_file_cluster_wise()
       utilities_cg.write_to_file_std_dev()
       utilities_cg.write_to_file_avg_std_dev()

  elif kind_of_attack == 'Node dies' or kind_of_attack ==
      '3':
       constant.TYPE_OF_ATTACK=3
       num_of_nodes_to_die=int(raw_input('Number of nodes
          to die : '))
       num_of_nodes=int(list_of_lines[1])
       for i in range(0, num_of_nodes_to_die):
       for i in range(2,len(list_of_lines)):
           constant.DEAD_NODES.append(int(list_of_lines[i
              ]))
```

```
        current_node = int(raw_input('Enter the node
            number to make it die : '))
        constant.DEAD_NODES.append(current_node)
        current_node=int(list_of_lines[i])
        utility.node_death(current_node)
        n_attack.
            node_death_and_VC_of_neighbors_affected(
            current_node)
    utility.check_type_of_nodes_after_change()
    results=open('results/current.txt', 'a')
    sdf=utility.num_of_nodes_changed()
    results.write(str(sdf)+" ")
    results.write(str(len(constant.DEAD_NODES)))
    for each in constant.DEAD_NODES:
        results.write(":"+str(each))
    results.write('\n')
    results.close()
    utility.draw_display_graph(display_VC = False,
       save_fig = False,
    fig_name = 'node_death.png')
beta_reputation.for_all_nodes_self_rep(constant.
   WEIGHT_FACTOR)
for each_node in constant.GRAPH.nodes():
    if constant.GRAPH.node[each_node]['type'] is '
       Changed_chosen_VC' or constant.GRAPH.node[
       each_node]['type'] is 'Changed_VC_node':
        print each_node
        print constant.GRAPH.node[each_node]['
           Self_Reputation']
    constant.X_VALUES.append('node_death')
beta_reputation.neighbor_rep('all')
beta_reputation.write_to_sheet()
beta_reputation.write_to_sheet('with_discounting')
    for each in constant.GRAPH.nodes():
        SVD.calculate_xSVD_ySVD(each, V, 'c')
    n_detection.write_neighbor_distance_changed()
    utilities_cg.create_clusterwise_x_y_positions('c')
    utilities_cg.mean_x_y_position_cluster('c')
    utilities_cg.CG_details_changed()
    utilities_cg.total_network_rating()
    distributed_cluster_heads.
       calculate_std_dev_of_clusters('c')
    distributed_cluster_heads.change_in_self_dev()
```

```
        utilities_cg.append_csv_file(file_name)
        utilities_cg.
           append_csv_file_with_intensity_of_attack(
           file_name)
        utilities_cg.write_to_file_std_dev()
        utilities_cg.write_to_file_avg_std_dev()

    elif kind_of_attack == 'Plot Graph' or kind_of_attack
       == '4':
        print 'In elif'
        graph_plot.attack_vs_avg_self_rep()

    elif kind_of_attack == 'Calculate Area' or
       kind_of_attack == '5':
        a1 = int(raw_input('1st anchor node : '))
        a2 = int(raw_input('2nd anchor node : '))
        a3 = int(raw_input('3rd anchor node : '))
        print "Area of triangle is : " + str(CSR_utilities.
           area_anchor_nodes
        ([a1, a2, a3]))
        type_area = raw_input('original(o) or changed(c)?')
        if type_area is 'original' or type_area is 'o':

            for each in constant.GRAPH.nodes():
                if(constant.GRAPH.node[each]['type'] is '
                   Regular_node'):
                print "Node : " + str(each)
                CSR_utilities.addition_of_areas(each, [a1,
                   a2, a3])

        elif type_area is 'changed' or type_area is 'c':
            for each in constant.GRAPH.nodes():
                if(constant.GRAPH.node[each]['type'] is '
                   Changed_VC_node' or
                   constant.GRAPH.node[each]['type'] is '
                      Changed_chosen_VC' or
                   constant.GRAPH.node[each]['type'] is '
                      deflated'):
                    print "Node : " + str(each)
                    print 'Original Area : '
                    CSR_utilities.addition_of_areas(each, [
                       a1, a2, a3])
                    print 'Changed Area'
```

```
                    CSR_utilities.addition_of_areas(each, [
                        a1, a2, a3], 'changed')

    elif kind_of_attack == 'Calculate Shortest Distance' or
        kind_of_attack == '6':
        source_node=0
        destination_node=0
        source_VC_status= None
        source_VC_status=None
        all_or_one = raw_input('all(1) or single node(2) :'
            )
        if(all_or_one =='1'):
            destination_node=int(raw_input('Enter second
                node : '))
            source_VC_status=str(raw_input('status of
                source VC : '))
            destination_VC_status=str(raw_input('status of
                destination VC : '))
            for each_node in constant.GRAPH.nodes():
                source_node=each_node
                min_max_value = CSR_utilities.
                    shortest_hop_distance_bound
                (source_node, destination_node,
                    source_VC_status,
                destination_VC_status)
                min_hop_distance = len(nx.shortest_path
                (constant.GRAPH, source=each_node,target=
                destination_node))-1
        else:
            source_node=int(raw_input('Enter first node : '
                ))
            destination_node=int(raw_input('Enter second
                node : '))
            source_VC_status=str(raw_input('status of
                source VC : '))
            destination_VC_status=str(raw_input('status of
                destination VC : '))
            min_max_value = CSR_utilities.
                shortest_hop_distance_bound
            (source_node, destination_node,
                source_VC_status,
            destination_VC_status)
```

```
elif kind_of_attack == 'Calculate topological xy
   coordinates' or kind_of_attack == '7':
    status = raw_input('original (o) or changed(c) : ')
    node_number = int(raw_input('Enter the node number
       : '))
    if status is 'o' or status is 'original':
        print constant.GRAPH.node[node_number]['
           x_y_original']
    elif status is 'c' or status is 'changed':
        Pchanged = P
        Pchanged[node_number,:]=constant.GRAPH.node[
           node_number]['vector_cordinates_changed']
        print Pchanged
        Vchanged = SVD.calculate_SVD(Pchanged)
        SVD.calculate_xSVD_ySVD(node_number, Vchanged,
           status)
        SVD.calculate_xSVD_ySVD(node_number, V, status)
        print 'Original X Y : ' + str(constant.GRAPH.
           node[node_number]
        ['x_y_original'])
        print 'Changed X Y : ' + str(constant.GRAPH.
           node[node_number]
        ['x_y_changed'])

elif kind_of_attack == '8' or kind_of_attack == 'Plot
   XY coordinates':
    pairs = []
    anc = []
    plt.figure()
    color=0
    for each in constant.GRAPH.nodes():
        if constant.GRAPH.node[each]['type'] is '
           Anchor_node' or
        constant.GRAPH.node[each]['type'] is '
           Anchor_node_VC_chaged':
            anc.append((constant.GRAPH.node[each]['
               x_y_original'][0], constant.GRAPH.node[
               each]['x_y_original'][1]))
    for each_list in constant.CLUSTERS_OF_NODES:
        pairs=[]

        for each_node in each_list:
```

```
        if constant.GRAPH.node[each_node]['type'] is
           'deflated':
             plt.plot(constant.GRAPH.node[each_node
                ]['x_y_original'][0],
                    constant.GRAPH.node[each_node
                       ]['x_y_original'][1], 's',
                       c='r')
         else:
         pairs.append((constant.GRAPH.node[each_node
            ]['x_y_original'][0], constant.GRAPH.
            node[each_node]['x_y_original'][1]))
    plt.plot([p[0] for p in pairs], [p[1] for p in
       pairs], 'o',
     c=constant.COLORS[color])
    plt.plot([p[0] for p in pairs], [p[1] for p in
       pairs], 'o',
    c='MediumBlue')
    color=color+1
color=0
for each in anc:
    plt.plot([each[0]], [each[1]], '^', c=constant.
       ANCHOR_COLORS[color])
    plt.plot([each[0]], [each[1]], 'o', c='
       MediumBlue')
    color=color+1
plt.plot(constant.X_Y_CG_FULL[0], constant.
   X_Y_CG_FULL[1], 'D', c='Black')
color=0
for i in range(0, constant.NUM_OF_ANCHOR_NODES):
  plt.plot(constant.X_CG_CLUSTERS_ORIGINAL[i],
  constant.Y_CG_CLUSTERS_ORIGINAL[i], 'D',
  c=constant.ANCHOR_COLORS[color])
    color=color+1
plt.tick_params(axis='x', labelsize=25)
plt.tick_params(axis='y', labelsize=25)
plt.show(block=False)
graph_plot.original_CG()
changed = raw_input('Plot changed XY coordinates ?(
   y/n) : ')
if changed is 'y':
    plt.figure()
    color=0
    for each_list in constant.CLUSTERS_OF_NODES:
```

```
        pairs=[]
        for each_node in each_list:
            if (constant.GRAPH.node[each_node]['
               type'] is 'Changed_VC_node' or
               constant.GRAPH.node[each_node]['type
               '] is 'Changed_chosen_VC') or
               constant.GRAPH.node[each_node]['type
               '] is'deflated':
                pairs.append((constant.GRAPH.node[
                   each_node]['x_y_changed'][0],
                   constant.GRAPH.node[each_node]['
                   x_y_changed'][1]))
            elif constant.GRAPH.node[each_node]['
               type'] is 'Regular_node':
                pairs.append((constant.GRAPH.node[
                   each_node]['x_y_original'][0],
                   constant.GRAPH.node[each_node]['
                   x_y_original'][1]))

            plt.plot([p[0] for p in pairs], [p[1]
               for p in pairs], 'o', c=constant.
               COLORS[color])
            plt.plot([p[0] for p in pairs], [p[1]
               for p in pairs], 'o', c='MediumBlue'
               )
        color=color+1
    for each_node in constant.GRAPH.nodes():
        if constant.GRAPH.node[each_node]['type'] is
           'deflated':
            plt.plot(constant.GRAPH.node[each_node][
               'x_y_changed'][0],
                     constant.GRAPH.node[each_node][
                        'x_y_changed'][1], 's', c='r
                        ')
    for each in constant.GRAPH.nodes():
        if (constant.GRAPH.node[each]['type'] is '
           deflated' or
            constant.GRAPH.node[each]['type'] is '
               Changed_VC_node' or
```

```
            constant.GRAPH.node[each]['type'] is '
               Changed_chosen_VC'):
                                        pairs1.append
               ((constant.GRAPH.node[each]['
               x_y_changed'][0], constant.GRAPH.
               node[each]['x_y_changed'][1]))
        else:
            pairs1.append((constant.GRAPH.node[each
               ]['x_y_original'][0], constant.GRAPH
               .node[each]['x_y_original'][1]))
plt.figure()
plt.plot([p[0] for p in pairs1], [p[1] for p in
    pairs1], 'o', c='DarkBlue')
color=0
anc=[]
for each in constant.GRAPH.nodes():
    if constant.GRAPH.node[each]['type'] is '
       Anchor_node' or
     constant.GRAPH.node[each]['type'] is '
       Anchor_node_VC_chaged':
        anc.append((constant.GRAPH.node[each]['
           x_y_changed'][0], constant.GRAPH.
           node[each]['x_y_changed'][1]))
for each in anc:
 plt.plot([each[0]], [each[1]], '^', c=constant
    .ANCHOR_COLORS[color])
    plt.plot([each[0]], [each[1]], 'o', c='
       MediumBlue')
    color=color+1
plt.plot(constant.X_Y_CG_FULL[0], constant.
   X_Y_CG_FULL[1], 'D', c='Black')
color=0
for i in range(0, constant.NUM_OF_ANCHOR_NODES)
   :
    plt.plot(constant.X_CG_CLUSTERS_CHANGED[i],
        constant.Y_CG_CLUSTERS_CHANGED[i], 'D',
        c=constant.ANCHOR_COLORS[color])
    color=color+1
plt.tick_params(axis='x', labelsize=25)
plt.tick_params(axis='y', labelsize=25)
plt.show(block=False)
graph_plot.compare_CG()
CG.calculate_angle_rating()
```

```
            graph_plot.x_y_changed_vs_original()

    elif kind_of_attack == '10':
        SVD.calculate_SVD(P, 'SVD/original.txt')
        list_of_list = []
        for each_node in constant.GRAPH.nodes():
            list_of_list.append(constant.GRAPH.node[
               each_node]['vector_cordinates_changed'])
        P_full_changed = np.array(list_of_list)
        SVD.calculate_SVD(P_full_changed, 'SVD/
           full_matrix_changed.txt')
        list_of_list = []
        for each_node in constant.GRAPH.nodes():
            if (constant.GRAPH.node[each_node]['type'] is '
               deflated' or
                  constant.GRAPH.node[each_node]['type'] is
                     'Changed_VC_node' or
                  constant.GRAPH.node[each_node]['type'] is
                     'Changed_chosen_VC'):
                list_of_list.append(constant.GRAPH.node[
                   each_node]['vector_cordinates_changed'])
        P_subset_changed = np.array(list_of_list)
        SVD.calculate_SVD(P_subset_changed, 'SVD/
           subset_matrix_changed.txt')
        list_of_list = []
        for each_node in constant.GRAPH.nodes():
            if (constant.GRAPH.node[each_node]['type'] is '
               deflated' or
                  constant.GRAPH.node[each_node]['type'] is
                     'Changed_VC_node' or
                  constant.GRAPH.node[each_node]['type'] is
                     'Changed_chosen_VC'):
                list_of_list.append(constant.GRAPH.node[
                   each_node]['vector_cordinates_original'
                   ])
        P_subset_original = np.array(list_of_list)
        SVD.calculate_SVD(P_subset_original, 'SVD/
           subset_matrix_original.txt')
     n_detection.write_avg_value()
     n_detection.write_neighbor_distance_changed()
     n_detection.correct_false_non(T=5)
    results=open('results/current.txt', 'a')
    sdf=utility.num_of_nodes_changed()
```

```
            results.write(str(sdf)+","+str(constant.NETWORK_RATING)
               +"\n")
            false_positive=open('results/false_positive.txt', 'a')
            correct_detection=open('results/correct_detection.txt',
                'a')
            no_detection=open('results/no_detection.txt', 'a')
            print
            print 'Type of Attack : ' + str(constant.TYPE_OF_ATTACK
               )
            print 'Network rating : ' + str(constant.
               AVERAGE_STD_DEV)
            if constant.TYPE_OF_ATTACK == 3:
                if constant.AVERAGE_STD_DEV >= threshold_value:
                    print 'false positive'
                    false_positive.write('1\n')
                else:
                    print 'correct detection of node death'
                    correct_detection.write('1\n')
            else:
                if constant.AVERAGE_STD_DEV > threshold_value:
                    print 'correct detection of attack'
                    correct_detection.write('1\n')
                else:
                    print 'no detection'
                    no_detection.write('1\n')
            false_positive.close()
            correct_detection.close()
            no_detection.close()
            decision_to_quit = raw_input("Do u want to exit?")
            if(decision_to_quit is 'y'):
                break
    if __name__ == '__main__':
        main()

    def check_type_of_nodes_after_change():
        for each in constant.GRAPH.nodes():
            if constant.GRAPH.node[each]['type'] == 'Regular_node':
                if not compare_vector_cordinates(each):
                    constant.GRAPH.node[each]['type'] = '
                       Changed_VC_node'
            if constant.GRAPH.node[each]['type'] == '
               Changed_VC_node':
                if detect_if_chosen_VC_is_changed(each):
```

```
                constant.GRAPH.node[each]['type'] = '
                    Changed_chosen_VC'
                constant.NODES_WITH_CHANGED_CHOSEN_VC.append(
                    each)
        elif constant.GRAPH.node[each]['type'] == 'Anchor_node'
            :
            if not compare_vector_cordinates(each):
                constant.GRAPH.node[each]['type'] = '
                    Anchor_node_VC_chaged'

def num_of_changed_vectors(node):
    constant.GRAPH.node[node]['position_of_VC_changed'] = []
    comparator = []
    i=0
    while i < len(constant.GRAPH.node[node]['
        vector_cordinates_changed']):
        if constant.GRAPH.node[node]['vector_cordinates_changed
            '][i] !=
        constant.GRAPH.node[node]['vector_cordinates_original'
            ][i]:
            comparator.append(constant.GRAPH.node[node]
            ['vector_cordinates_changed'][i])
            constant.GRAPH.node[node]['position_of_VC_changed'
                ].append(i)
        i = i+1
    return len(comparator)

def compare_vector_cordinates(node):
    constant.GRAPH.node[node]['position_of_VC_changed'] = []
    comparator = []
    i=0
    while i < len(constant.GRAPH.node[node]['
        vector_cordinates_changed']):
        if constant.GRAPH.node[node]['vector_cordinates_changed
            '][i]
        != constant.GRAPH.node[node]['
            vector_cordinates_original'][i]:
            comparator.append(constant.GRAPH.node[node]
            ['vector_cordinates_changed'][i])
            constant.GRAPH.node[node]['position_of_VC_changed'
                ].append(i)
        i = i+1
    if len(comparator) > 0:
```

```
        return False
    else:
        return True

def sort_anchor_VC_tuple(current_node):
    constant.GRAPH.node[current_node]['sorted_anchor_VC'] = []
    x = create_anchor_VC_tuple(current_node)
    sorted_x = sorted(x.items(), key=operator.itemgetter(1))
    constant.GRAPH.node[current_node]['sorted_anchor_VC'] =
       sorted_x
    return sorted_x

def draw_clusters_graph():
    plt.figure()
    color = 0
    pos = nx.graphviz_layout(constant.GRAPH)
    for each_list in constant.CLUSTERS_OF_NODES:
        nodes1 = nx.draw_networkx_nodes(constant.GRAPH,
        constant.POSITIONS, nodelist=each_list, node_color=
                                constant.COLORS[color],
                                   node_size=60)
        nodes1.set_edgecolor(constant.COLORS[color])
        color = color+1
    color=0
    for each_node in constant.ANCHOR_NODES:
        nodes = nx.draw_networkx_nodes(constant.GRAPH,
        constant.POSITIONS,nodelist=[each_node], node_color=
        constant.ANCHOR_COLORS[color], node_size=150)
        color=color+1
    nx.draw_networkx_edges(constant.GRAPH, constant.POSITIONS,
    edge_color='PaleTurquoise')
    plt.tick_params(axis='x', labelsize=25)
    plt.tick_params(axis='y', labelsize=25)
    plt.show(block = False)

def num_of_nodes_changed():
    count = 0;
    for each in constant.GRAPH.nodes():
        if constant.GRAPH.node[each]['type'] is 'deflated' or
        constant.GRAPH.node[each]['type'] is 'Changed_chosen_VC
           ' or
        constant.GRAPH.node[each]['type'] is '
           Anchor_node_VC_chaged':
```

```
            count=count+1
    return count
```

APPENDIX B

Appendix B:List of Abbreviations

ENS: Extreme Node Search

GC: Geographical Coordinates

GCS: Geographical Coordinate System

GPS: Global Positioning System

GPSR: Greedy Perimeter Stateless Routing

TC: Topological Coordinates

TPM: Topology Preserving Maps

VC: Virtual Coordinates

VCS: Virtual Coordinate System

WSN: Wireless Sensor Networks

APPENDIX C

Appendix C:Other Simulation Networks

Networks are referred from [27].

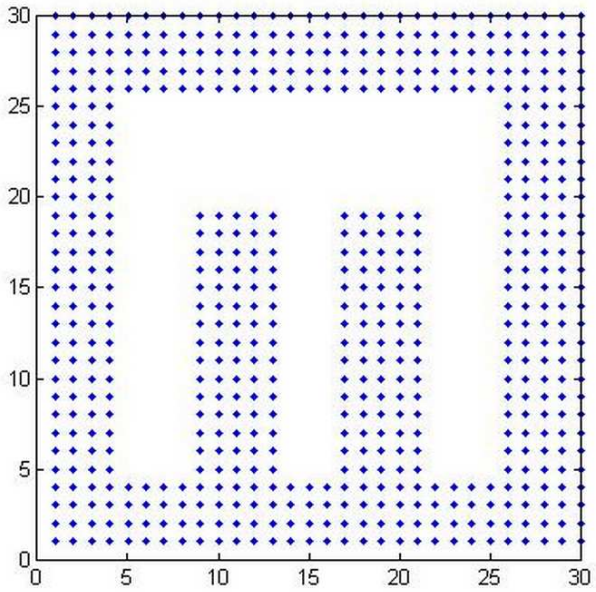


Figure C.1. M-Shaped Network

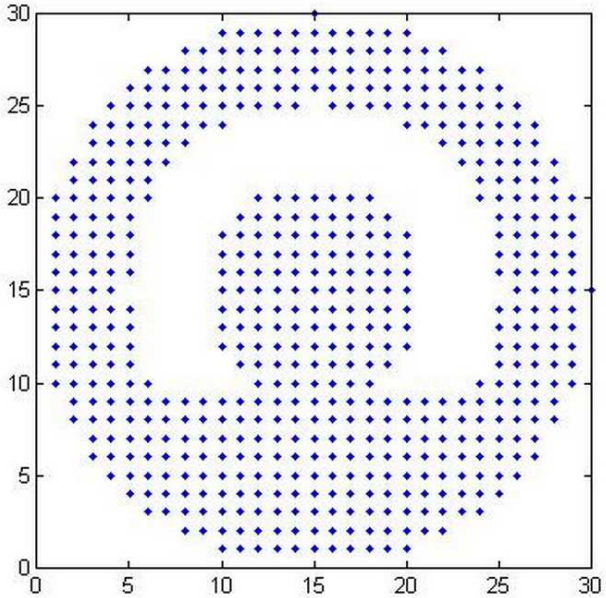


Figure C.2. Concave Void Network