TinyTermite: A Secure Routing Algorithm

A Thesis Presented for the

Master of Science Degree

The University of Tennessee at Chattanooga

Joshua L. Patterson

December 2008

To the Graduate Council:

I am submitting a thesis written by Joshua L. Patterson entitled “TinyTermite: A Secure Routing Algorithm”. I have examined the final copy of this thesis and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science.

____

Dr. Mina Sartipi, Chairperson

We have read this thesis and recommend its acceptance:

____

Dr. Billy Harris

____

Dr. Li Yang

Accepted for the Graduate Council:

____

Interim Dean of the Graduate School

Dedication

I would like to dedicate this master’s thesis and my master’s degree to my parents. Without their support neither would be possible.

Abstract

In this thesis, we introduce TinyTermite. TinyTermite is a novel probabilistic routing algorithm that is secure against selective forwarding and replay attacks. We use suspicion pheromone to build a flexible map of possible compromised neighbors. As suspicion builds up and decays for each neighbor, TinyTermite is able to deal with uncertain stimulus and react properly. TinyTermite is fully implemented on TinyOS based Intel Mote 2 platform and the experiments were done to compare its performance with that of the traditional Termite algorithm. The experimental results show that TinyTermite is significantly more secure against replay and sinkhole attacks by lowering the packet loss from 88.5% to 32.9% with 12.7% normal packet loss. The experimental results also demonstrate that the TinyTermite provides high throughput and low latency.

Suspicion pheromone is added to Termite that defends against the selective forwarding and replay attacks. In our experiments we implemented a detection scheme for the replay attack and show how countermeasures can be employed in TinyTermite. We found the suspicious pheromone technique to be quite effective against a replay attack coupled with a selective forwarding attack. The suspicion defense mechanism in TinyTermite provided a significantly better defense against the attacker than Termite alone.

The implementation of TinyTermite on the Imote2 platform was measured with respect to throughput, latency, and packet loss. We examine and measure the basic packet transmission mechanics between nodes on the Imote2 platform and suggest settings for better quality operations.

TABLE OF CONTENTS

Abstract ii
TABLE OF CONTENTS iii
LIST OF TABLES iv
LIST OF FIGURES v
LIST OF FIGURES v
CHAPTER 1 6
CHAPTER 2 10
2.1 Introduction To Wireless Ad Hoc Networks 10
2.2 Routing Algorithms for MANETs 14
2.3.1 DSR 20
2.2.1 DSDV 23
2.2.2 AODV 26
2.2.3 OLSR 29
2.3 Summary 31
CHAPTER 3 34
3.1 Introduction 34
3.2 Principles of Self Organization 36
3.2.1 Feedback 37
3.2.2 Stigmergy 38
3.3 Pheromone 38
3.4 Packet Types and Layout 43
3.5 Comparison to AODV 44
CHAPTER 4 48
4.1 Introduction 48
4.2 Contribution 48
4.3 Attacks On MANETs 49
4.3.1 Replay Attack 50
4.3.2 Selective Forwarding 55
4.5 Implementation 56
4.6 Testing Procedures 59
4.7 Results 63
CHAPTER 5 79
5.1 TinyTermite Implementation 79
5.2 Suspicious Pheromone 79
5.3 Suggestions for Future Work 80
APPENDIX A 81
REFERENCES 85

LIST OF TABLES

Table 1: Strengths and Weaknesses of the four routing algorithms we review................ 33
Table 2: Comparison of Termite and AODV with respect to eight key measures. 47
Table 3: Results of exploratory pass on radio backoff window parameters with radio power setting of 7... 66
Table 4: Comparison of throughput between two nodes (raw throughput), a line network with a preset path, and then multipath routing on a 6 node graph (Figure 9) with TinyTermite. 75
Table 5: Comparison of TinyTermite with Termite with regards to security. Experiments performed on network in Figure 9 where baseline packet loss for network is assumed to be 12.7%... 77

LIST OF FIGURES

Figure 1: The Bellman Ford Algorithm 16
Figure 2: Packet movement through Termite network.. 39
Figure 3: Pheromone table update.. 39
Figure 4: Routing in Termite. 42
Figure 5: Termite packet layout. 43
Figure 6: Replay attack diagram.. 51
Figure 7: TinyTermite packet layout. 56
Figure 8: Single line test scenario. 60
Figure 9: Six node graph for experimental setup to test throughput and latency. 61
Figure 10: Showing radio range of nodes at power 1 in a single line setup. 64
Figure 11: Show radio range of nodes at power 2 in a single line setup. 65
Figure 12: Packet drop percentage versus radio power, single line setup, 1 hop, Preset path with no routing. 68
Figure 13: Throughput versus radio power, single line setup, 1 hop, preset path with no routing. 69
Figure 14: Packet drop rate versus radio power; Each test uses nodes in a line formation with a preset path and no routing. 70
Figure 15: Throughput versus radio power; Each test uses nodes in a line formation with a preset path and no routing. 71
Figure 16: Packet drop rate versus radio power; Each test uses nodes in a line formation with active Termite routing. 73
Figure 17: Throughput versus radio power; Each test uses nodes in a line formation with active Termite routing. 74

CHAPTER 1

Introduction

Mobile ad hoc networks (MANETs) use a group of nodes to move information from a source node to a destination node without having a central authority directing the operations. MANETs can be quickly and inexpensively set up as needed and require no centralized administration or fixed network infrastructure such as base stations or access points. In recent years there has been tremendous growth in the laptop, small internet capable device, and smartphone markets [1]. Lately interest in MANETs has risen due to availability of license free wireless communication platforms. MANETs have begun to move into commercial industry as a way to track inventory, route traffic, and move information. For example, field technicians may need to work in a cluster to take readings, a class of students with laptops may need to interact in a peer to peer manner, or a group of business people may need to share files in an airport [2]. Secure and reliable communication is a necessary prerequisite for applications such as military exercises, disaster relief, and mine site operation.

With MANETs, we do not assume every node can hear every other node and that this characteristic is typical with these types of networks in terms of lack of complete connectivity. Wireless networks are inherently less reliable than wired networks. Routing is a key problem in any information network. In most networks such as the internet the topology is relatively static, with new changes taking a certain amount of time to propagate across the network. With an ad-hoc network we need to be able to route information even as the network topology changes, adapting on the fly. Routing protocols

for existing networks are not designed for the conditions demanded of MANETs. In addition, it is critical that ad-hoc networking be secure against many types of attacks before they can be employed in the field. Since nodes themselves communicate via the wireless medium which is open to most attackers with a Wi-Fi unit, attacks on wireless MANETs are relatively easy.

Termite [3] is a biologically inspired routing algorithm for MANETs that is based on previous work in adapting the effects of stigmergy [4]. Stigmergy is defined as information gathered from work in progress [5] and is a principle mechanic of self organizing systems. Termite models how termite colonies lay pheromone to communicate in a decentralized manner. The concept of a pheromone value is taken from the social insect world where insects such as termites lay small deposits of pheromone as they move to indirectly communicate with one another. The Termite algorithm [3] is very interesting from the perspective of its "on-line" training of the network via routing information "piggybacking" on top of normal network traffic, as opposed to the more common technique of sending packets for the purpose of purely updating routing information.

Self organizing systems exhibit emergent properties, where emergence refers to a process by which a system of interacting agents or nodes acquires qualitatively new properties that cannot be understood as the simple addition of their individual contributions [5]. Termites, just like all other social insects, have no centralized control, blueprints, or coordinators. All of their work coordination is done by examining their localized environment and using that as input into a rule base which tells them which task

to perform (sometimes a repeated encounter of a stimulus is necessary to perform a task switch [5]).

We secured the existing Termite algorithm against selective forwarding and replay attacks [6]. The new algorithm is called TinyTermite. Our contributions are the addition of suspicion and the implementation of Termite on the Imote2 platform. Suspicion is a technique inspired by task response thresholds in ant colonies which allows a MANET to detect certain classes of attacks even given inconclusive information and alter its behavior in a proportional response. In our experiments we implemented a detection scheme for the replay attack. We also show how countermeasures can be employed in TinyTermite. We propose an alternative method to deal with selective forwarding attacks based around disjointed paths taken by pairs of packets. We describe these attacks in sections 4.3.1 and 4.3.2.

We also implemented Termite on the Imote2 platform [7] under the TinyOS operating system [8] and with the nesC programming language [9]. We investigated the properties of the Imote2 platform by measuring its one hop performance, radio subsystem backoff windows, theoretical throughput across many hops, and how radio power affects performance. We also measured Termite with respect to throughput, latency, and packet loss on the Imote2 platform and found it to be an effective routing algorithm for the platform. We found the suspicion pheromone defense to be highly effective against the replay and selective forwarding attacks coupled together and the system successfully routed significantly more packets to the intended destination than it did without the defense mechanism.

We review existing ad-hoc network routing protocols in Chapter 2, comparing four of the major ad-hoc routing protocols. In Chapter 3 we examine Termite and the principles of self organization. We also review how feedback and stigmergy drive emergent systems at the local levels. To conclude the chapter we compare AODV and Termite. We introduce TinyTermite in Chapter 4 and discuss the attacks we are defending against. We also introduce the disjoint path mechanism and the suspicion defense mechanism to combat the attacks. We then discuss our experiments concluding with our results. In Chapter 5 we list our conclusions and future work.

CHAPTER 2

Routing in Wireless Ad-Hoc Networks

2.1 Introduction To Wireless Ad Hoc Networks

Wireless ad-hoc networking uses a group of nodes to move information from a source node to a destination node without having a central authority direct the operations. In certain situations users will want to communicate in situations where no fixed wired infrastructure exists, either because it may not be economically practical or physically possible. Wireless ad-hoc networks provide capabilities that service these type conditions where infrastructure is unavailable for more traditional wired approaches.

Ad hoc networks can be quickly and inexpensively set up as needed and require no centralized administration or fixed network infrastructure such as base stations or access points. In recent years there has been tremendous growth in the laptop, small internet capable device, and smartphone markets [1]. These smaller machines are equipped with sufficient amounts of memory, storage, high resolution displays, and wireless communication adapters. Given their increasing battery life, they are free to roam for hours without the constraint and tether or a wired infrastructure. These devices are quickly becoming a pervasive part of everyday computing infrastructure.

A mesh network is a type of ad hoc network where all nodes are connected via multiple hops. As mesh networks tend to be stationary, another development in this technology has been Mobile Ad-hoc Networks (MANETs) which allow for the individual nodes to move and self organize on the fly. The idea of forming a mobile network of ad

hoc nodes on the fly dates back the DARPA packet network radio days [10, 11]. MANETs share few properties with centrally oriented networks. First and foremost, the topology can be quite dynamic and change frequently, something traditional wired networks aren't focused on. Second, with an ad hoc network users do not want to perform administration tasks to set up such a network. Lately interest in MANETs has risen due to availability of license free wireless communication platforms. Growing interest within the Internet Engineering Task Force (IETF) has amounted to the formation of a new working group [12, 13] whose charter is to develop a framework for routing in ad-hoc networks.

With MANETs, it is not assumed that every node can hear every other node. This characteristic is typical with these types of networks in terms of lack of complete connectivity and because the wireless physical medium is limited and variable in range, in distinction to existing wired media. With wireless ad-hoc networks if only two nodes or hosts are in close proximity to one another in an ad-hoc network, no real routing is required. However, if multiple nodes are spread out in an ad-hoc network, and some of them can only hear a subset of the network, then routing is required and their neighbors nodes will have to route their packets for them. Wireless networks are inherently less reliable than wired networks. In a wireless environment, network connectivity between two nodes does not always work consistently in both directions. This is due to differing propagation or interference patterns around the two hosts [14, 15]. Also with the wireless medium, battery, computational power, and communication bandwidth are scarce.

Routing is a key problem in any information network. However with the addition of a topology in flux and wireless medium obstacles to overcome, routing becomes an

even more complex problem to deal with. In most networks such as the internet the topology is relatively static, with topology changes taking a certain amount of time to propagate across the network. With an ad-hoc network we need to be able to route information even as the network topology changes, adapting on the fly. Routing protocols for existing networks are not designed for the conditions demanded of MANETs. Ad-hoc networks demand a routing protocol that can provide the kind of dynamic, self starting behavior needed. Most conventional routing protocols provide their worst performance when faced with a highly dynamic interconnection topology since conventional routing protocols are not designed to endure frequent topology changes that tend to occur in ad-hoc networks.

Research on packet routing in wireless networks of mobile hosts dates back to at least 1973 when the DARPA Packet Radio Network (PRNET) was started [16]. Its successor was the Survivable Adaptive Networks (SURAN) project [15]. Originally designed for military applications, mesh networks have begun to move into commercial industry as a way to track inventory, route traffic, and move information, among other uses. For example, a class of students with laptops may need to interact in a peer to peer manner, field technicians may need to work in a cluster to take readings, or a group of business people may need to share files in an airport [2]. In all of these situations a collection of hosts with wireless interfaces may form a temporary network without any sort of pre-existing infrastructure of administration.

Although mobile computers can be modeled as routers, it is very clear that conventional routing protocols coupled with this concept alone places too heavy a computational burden on each mobile device. With conventional networks, links between

routers occasionally go down or come up and the cost of a link may change due to congestion, but the routers do not move around. Also, the convergence mechanics of existing routing protocols are generally not acceptable when used with MANETs. In networks with mobile nodes, however, convergence to new stable routes can take time or be very slow given that the new routing information has to propagate across the network.

MANETs (unless mentioned otherwise, from this point onwards we'll consider this term to mean wireless ad hoc networks) face many challenges as a result of attacks, frequent changes of network topology, and natural physical layer interferences. Furthermore, the techniques of split horizon and poisoned reverse [17] are not useful within the wireless environment due to the broadcast nature of the wireless medium.

Secure and reliable communication is a necessary prerequisite for applications such as military exercises, disaster relief, and mine site operation. While these type operations may benefit from ad hoc networking, it is paramount that ad-hoc networking be secure against many types of attacks before they can be employed in the field [18]. Up until recently researchers in ad hoc networking have mostly studied the routing problem in sandbox type conditions, assuming a trusted environment; very little research has been performed in a more realistic arena where adversaries can deploy and array of attacks in attempts to disrupt the network.

Attacks on wireless MANETs are relatively easy since the nodes themselves communicate via the wireless medium which is open to most attackers with a WI-FI unit. The attacks on ad hoc network routing protocols generally fall into one of two categories; First we have routing-disruption attacks, where the attacker attempts to cause legitimate data packets to be routed in dysfunctional ways. Then we have resource-consumption

attacks, where the attacker injects packets into the network in an attempt to consume valuable network resources such as bandwidth [18]. One of the major type of attacks we will focus on later in this thesis is the replay attack where an attack takes a valid packet and replays it to a neighbor node. This type of attack can not only drain valuable resources but also influence routing patterns in certain types of networks. Other attacks we look at are selective forwarding and sinkhole attacks, where packets are sometimes routing maliciously or not at all, respectively.

2.2 Routing Algorithms for MANETs

Routing takes place in many types of networks such as telephone, electronic data, and transportation networks. Popular routing methods attempt to achieve the common objective of routing packets along the optimal path. There are two main types of routing algorithms in wireless ad hoc networks. They are Distance Vector [17, 19, 20, 21, 22] and Link State [23, 24, 25] algorithms. Typically conventional wired networks use either distance vector or link state routing algorithms. Ad hoc On Demand Distance Vector (AODV) is a routing algorithm that is of the distance vector class. The researchers behind AODV state that an ad hoc routing protocol needs to have support for multi hop paths, it needs to be self starting, have dynamic topology maintenance, be loop free, have low consumption of memory and bandwidth, be scalable to large node populations, localize the effect of link breakage, have minimal overhead for data transmission, and be able to rapidly converge [26].

The distance vector approach assigns a cost to each of the links between each node in the network. Each node will send information from point A to point B along the

determined set of nodes, or the path, that will result in the lowest total cost. Nodes transmit the distance metric to each node to all of their neighbors. Nodes then compute the shortest path to all other nodes based on the information advertised by its neighbors. In addition to being used in wired networks, distance vector algorithms have been adapted for use in wireless ad hoc networks where each node is essentially a router in the network [16, 27, 28]. Most distance vector algorithms use the Bellman-Ford algorithm [29]. The Bellman-Ford algorithm solves the single source shortest paths problem in the general case in which edge weights may be negative. As explained in the algorithm is as follows:

Given a weighted, directed graph $G = (V, E)$, We have a set of vertices V and a set of edges E, with source s and weight function

$$w : E \rightarrow R$$

the Bellman-Ford algorithm [30] returns a boolean value indicating whether or not there is a negative weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

The algorithm also uses relaxation, progressively decreasing an estimate $d[v]$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$. For each vertex $v \in V$, we maintain an attribute $d[v]$, which is an upper bound on the weight of a shortest path from source s to v. We call $d[v]$ a shortest-path estimate. The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

After Initialize-Single-Source (G, s) , $\pi[v] = NIL$ for all $v \in V$, $d[s] = 0$, and $d[v] = \infty$ for $v \in V - \{s\}$, (where $\pi[v]$ is a predecessor vertex). The algorithm calls Initialize-Single-Source (G, s) and then repeatedly relaxes the edges of the graph G with the RELAX(u, v, w) function. The RELAX(u, v, w) function [30] performs the technique of relaxation. The process of relaxing an edge (u, v) consist of testing whether we can improve the shortest path to v found so far by going though u, and, if so, updating $d[v]$ and $\pi[v]$. The algorithm is listed in Figure 1 below.

RELAX(u, v, w)

1 **if** $d[v] > d[u] + w(u, v)$

2 **then** $d[v] \leftarrow d[u] + w(u, v)$

3 $\pi[v] \leftarrow u$

Bellman-Ford(G, w, s)

1 Initialize-Single-Source(G, s)

2 **for** i $\leftarrow$ 1 **to** $|V[G]|$ - 1

3 **do for** each edge (u, v) $\in E[G]$

4 **do** RELAX(u, v, w)

5 **for** each edge (u, v) $\in E[G]$

6 **do if** $d[v] > d[u] + w(u, v)$

7 **then return** FALSE

8 **Return** TRUE

Figure 1: The Bellman Ford Algorithm

Each node monitors the cost of its outgoing links and periodically broadcasts updates of its current estimates of the shortest distance to every other node in the network to its neighbors in order to keep distance estimates up to date. As the updates are received by each node, the distance estimates are recalculated. This described technique is the classical distributed bellman-ford algorithm [31]. It is computationally more efficient than compared to link state methods, and easier to implement while requiring less amount of storage space.

One shortcoming of the distributed bellman-ford protocol is that it is not loop-free [32]. The paths implied by the routing-tables of all nodes taken together can have loops at any moment. This means that if a path to a destination is traced going from the routing-table of one node to that of another node, a node may be visited more than once before the destination is reached. Looping of data to be routed may occur resulting in considerable overhead if such routing-table loops persist for a long time.

With link-state routing each node broadcasts to all other nodes in the network its view of the status of each of its adjacent network links. Each node then computes the shortest distance to each host based on the complete picture of the network formed from the most recent link information from all nodes. This approach is closer to the centralized version of the shortest path computation method, with each node maintaining a view of the network topology with a cost for each link. To maintain fresh state across the network, each node periodically broadcasts the link costs of its outgoing links to all other

nodes using flooding or a similar technique. As a node receives the information, it updates its own state table to reflect the changes in the network conditions and applies the shortest path algorithm for each destination to choose the preferred next hop neighbor. Open Shortest Path First (OSPF) [25] is an early example of a link state algorithm. It is one of the dynamic routing protocol for use in IP networks.

Some disadvantages of link-state algorithms surface when link costs in a node's view can be incorrect because of long propagation delays or partitioned networks. These causes of inconsistent views of the network topology can lead to routing loops. These routing loops are generally short lived. A short lived routing loop is any loop that disappears in the time it takes a message to traverse the diameter of the network [24]. Link state algorithms also must maintain up to the date routing state for every node in the network, which creates a natural constraint in terms of bounding the network size due to excessive storage and communication overhead in a highly dynamic network.

Simplicity is one of the most preferred attributes of a routing protocol to be implemented in operational networks, with Routing Information Protocol (RIP) [17] being a classic example of this. Ad-hoc routing protocols can also have many classifications in terms of how they discover their routing information. Aspects of ad-hoc networks include:

- Node mobility
- Path maintenance
- Route table management
- Path loop prevention
- Local connectivity management

- Link layer mechanics

Ad-hoc networks generally end up in two main groups: proactive networks and reactive networks.

- **Proactive:** A proactive ad-hoc network seeks to constantly have the best possible global routing information in its routing tables, updating the routing table as new information comes online. [27]
- **Reactive:** A reactive ad-hoc network only caches routing information in its tables relative to the routes it has seen recently or needs to forward the current packets in its queue [33]. These algorithms are considered to be pure on-demand route acquisition systems. Nodes that reside outside active paths neither maintain routing state nor participate in periodic routing table exchanges. This thesis is focused on reactive networks, or on-demand networks, as they have been shown to give better performance than proactive networks for ad-hoc networks [26, 34, 35, 36].

In the next section, we will discuss four popular wireless mobile ad-hoc routing protocols and take a look at their strengths and weaknesses. The algorithms we study are

- **DSR** – Dynamic Source Routing
- **AODV** – Ad hoc On Demand Distance Vector
- **DSDV** – Destination Sequenced Distance Vector
- **OLSR** – Open Link State Routing

2.3.1 DSR

DSR is a routing protocol intended for wireless mesh networks [33]. It is a reactive protocol as it forms a route on demand when a transmitting computer or packet requests one. DSR's primary difference from distance vector protocols is that it does not rely on the routing table at each hop but rather relies on source routing instead. Source routing refers to having the source node dictate the path that the packet takes through the network with the entire path to the destination included in the data inside the packet. Source routing has been employed in a number of configurations for wired networks, using either statically defined or dynamically constructed source routes [37, 38]. It has also been used to route packets in the Tuscon Amateur Packet Radio (TAPR) work for wireless networks [39].

DSR allows the network to be self organizing and self configuring without existing infrastructure. The protocol adapts quickly to routing changes when changes in network topology is frequent, yet demands little to no overhead during periods of less node movement. Source routing also has the added benefit of producing loop free routes and eliminates the need for up to date routing information to be pushed to intermediate nodes along the route. Even up to the highest levels of simulated host movement, the overhead of DSR is quite low, falling to just 1% of total data packets sent for medium node movement rates in a network of 24 mobile nodes.

DSR has two main mechanisms; route discovery and route maintenance. These two mechanisms work together to allow nodes to discover and maintain source routes to various destinations in the ad-hoc network. There are no periodic broadcasts for this protocol. Instead, when a node needs to route to another node, it dynamically determines

one based on cached information and on the information gathered from the route discovery protocol.

Network nodes cooperate to forward packets for each other to allow for the delivery of packets to nodes not in direct range of wireless transmission for the source node. Given that the path length or hop order may change at any time, the nature of the network topology is highly dynamic. A primary focus in designing DSR was on creating a routing protocol that had very low overhead yet was able to react to changes in network topology quickly.

To send packets or data from a source node to a destination node, DSR constructs a source route giving the address of each hop along the way and the order the hops should be taken in. The packet is transmitted from node to node, and if the packet is not destined for the receiver, then the node reads the next hop in the header and forwards it on. Each node in the ad-hoc network maintains a route cache in which it caches source routes that is has learned. When a node wishes to send a packet to a destination node, the source node first checks its route cache for a source route to the destination node. If no entry is found, then the source node may initiate the route discovery protocol. While waiting for this procedure to complete, the node may continue to operate on other packets in a normal fashion, sending and receiving packets with other nodes. During usage of any source route, a node monitors its correct operation. If any hop in the source route goes offline, the route cannot be used to reach the destination. This path maintenance monitoring is employed to detect when the node needs to initiate discovery again to find a new correct route to the destination.

Route discovery is the mechanism by which DSR allows a node in the ad-hoc network to dynamically discover a route to any other node in the network. This destination can be immediately reachable or through other intermediary hops through other hosts. Route discovery broadcasts (RREQ) are initiated when a new route needs to be found by sending out a route request packet to the neighbor nodes in wireless range. If the route discovery process is successful, the initiating node will receive a route reply packet (RREP) which will list a sequence of nodes in the ad hoc network for the packet to traverse in order to reach its destination. Each RREQ packet contains a list of the nodes that it has traversed so far in order to reach the current receiving node. This is referred to as the route record, and also contains a unique request ID, set by the initiating node from a locally controlled sequence number. In order to detect duplicate entries, each node keeps a list of entries keyed with the node ID and the unique request ID pairs. If the node has seen a RREQ before, it discards it. The node will also discard the packet if its own address is already in the packet route list, as this prevents loops in the route. If the node is the destination node listed in the RREQ packet, then it will copy its own address into the route list record, and send a RREP packet back towards the source node. If it cannot service the request, the RREQ will be rebroadcast out to the node's neighbors with the node's own address appended to the route record in the packet. The RREQ thus propagates through the network until it reaches a node which can service the request.

DSR has an advantage in that it does not employ periodic broadcasting of routing advertisement messages. This saves on bandwidth overhead especially when there is little to no activity in the network and also saves on battery power which is a limited resource in sensor networks and handheld devices, allowing the device to sleep when activity dies

down. Ad-hoc networks inherently have multiple paths in them which a wired network tends not to have. With distance vector and link state algorithms, there is an increase in routing table overhead for these multiple path entries which must be maintained, which in turn requires more CPU overhead and power consumption.

One drawback with DSR is that a long enough path would require an arbitrarily long enough packet header. That is, as our path length approaches a large number, our header size will exceed the maximum allowed size of the physical medium in some cases.

2.2.1 DSDV

Destination-Sequenced Distance-Vector Routing (DSDV) is a table-driven routing scheme for MANETs based on the Bellman-Ford algorithm [27]. The main contribution of the algorithm, developed by C. Perkins and P.Bhagwat in 1994, was to solve the Routing Loop problem. It is considered proactive as well as a distance vector class algorithm that actively pushes out its routing table state periodically as changes occur.

DSDV uses a distributed version of the shortest path problem [40] where each node in the graph maintains a table which has an entry indicating a preferred neighbor for a destination. When a packet is routed to a destination node, the destination node's address is place in the header of the packet. As each node receives a packet, if the packet is addressed to the node, then the packet is passed on up to the application layer. If the packet is addressed to some other node, then the packet is forwarded to the neighbor which is preferred for that destination, with the forwarding process continuing until the packet reaches the destination.

DSDV is a variant of the distance vector routing method and is effective for creating ad-hoc networks for small populations of mobile nodes. DSDV depends on the correct operation of the periodic adjustment and global dissemination of routing table information. Being dependent on this, DSDV is generally considered a brute force approach. This limits the effective size of ad-hoc network that can employ DSDV when using frequent system wide broadcasts of routing information as the control message overhead grows $O(n^2)$. DSDV is also required to maintain at least one path for each possible destination in the network at all times which is quite costly in terms of node resources and wastes resources given that a lot of this information is never used. However, keeping complete routing table information does reduce the acquisition latency before the first packet transmission to a destination node.

The routing table contains entries, each of which has a sequence number which is generally even; else, an odd number is used. The destination generates this sequence number and the emitter needs to send out the next update with this number. Nodes send distribute routing information between themselves by sending full dumps occasionally and smaller incremental updates more often. The DSDV routing protocol requires that each node advertise its own routing table to each of its current neighbors. The advertisements must be made frequently enough as the entries in the list may change fairly dynamically over time. Each node also agrees to relay packets to other nodes upon request. The data broadcast by each node will contain its new sequence number, the destination address, the number of hops required to reach the destination, and the sequence number of the information received regarding that destination, as originally

stamped by the destination. The transmitted routing tables contain similar information with some additional fields like hardware address.

Routes received in broadcasts are also advertised by the receiver, propagating information across the network. The receiver increments the metric before advertising the route, since packets coming in via this route will require one more hop than before to reach the destination. A broken link is described by a metric of infinity, or any value greater than the max allowed for the metric. When the next hop for a destination has been broken, any route through that next hop is automatically assigned the value of infinity and an updated sequence number. This results in the broadcast of a routing information packet since it qualifies as a substantial route change. Some of these broadcasts are full dumps of a node's table with others augmenting them serving as partial incremental updates.

When a node receives new routing information, generally in an incremental update, the information is compared with existing routing information already available. Any route with a more recent sequence number is employed, and routes with older sequence numbers are discarded. Routes with equivalent sequence numbers are distinguished between by their metric in terms of which has fewer hops. The metrics for any routes selected out of the update are incremented by one hop, and the new entries are scheduled for advertisement to local neighbors.

Being available early on has given DSDV an inherent advantage in the market. By bounding the number of nodes in the network we can achieve good performance from the algorithm. Although many improved versions of this algorithm have been proposed, there is no commercial implementation of this algorithm. This is because no formal

specification of this algorithm exists so far. A strength of DSDV is that at all instants the protocol guarantees loop free paths to each destination

The major downside to DSDV is that it requires a regular update of its routing tables, which drains a small amount of bandwidth and uses up battery power even when the network is idle. Also, whenever the topology of the network changes, the network needs to re-converge which requires new sequence numbers to be generated for each node. Given these facts, DSDV is not suitable for large or highly dynamic networks. Another critical constraint of DSDV is the storage requirements for the routing table, as it can grow to be quite large.

2.2.2 AODV

AODV is a reactive routing protocol for MANETs. It falls into the class of distance vector algorithms, as its name suggests. Although not specifically dependent on the wireless physical medium, AODV was designed with limited range broadcast media in mind. Employing such media, a mobile node can have neighbors which hear its transmissions and yet may not hear one another. AODV uses symmetric links between neighboring nodes and does not attempt to use links that are not symmetric. This is due to the fact that the flow of packets needs to take the same path towards the source that it took towards the destination, but in reverse. Any link that was not bi-directional would break this basic mechanic of AODV.

AODV uses a system of route request (RREQ) packets and route reply (RREP) packets to setup on demand paths through its network. A RREQ packet hops from neighbor to neighbor actively looking for a node with path information towards a

destination. A RREP packet is sent from the node with information about the destination back along the path the RREQ packet has taken.

AODV employs a broadcast route discovery system that is similar in nature to the DSR algorithm. The primary difference is that instead of source routing AODV sets up a reverse path pointer at each hop on the way towards the destination node with a RREQ packet. Once the RREQ packet arrives at the destination, a RREP packet then begins to hop back towards the source node setting up the forward path pointer records. Each table entry at each node has a sequence number similar to how DSDV employs sequence numbers in order to keep up with the latest path information. However, unlike DSDV, each ad hoc node has a monotonically increasing sequence number counter which is used to supersede stale cached routes. The combination of these two techniques allows AODV to use its bandwidth efficiently, be responsive to changes in network topology, and maintain loop free routing. Each soft entry in a node's routing table has a timeout value that indicates when it can be removed. This ensures we only keep relevant routing information and allow the unused information to expire quickly. Routing entries that are used have their timeout value refreshed as data moves across the path and paths that no longer exist or are not being used are purged. A routing entry is considered active if a packet has used it inside the timeout period. With AODV all routes in the route table are marked with destination sequence numbers which ensure that no routing loops can be formed which is similar to DSDV.

Path maintenance is key to consistent operation in AODV. Movement of nodes not on an active path do not affect routing to that path's destination. If a source node moves during a data transmission operation, it can repeat the setup sequence and rebuild

the path to the destination which would be a reactive measure. Intermediate nodes and the destination node on the path send out a special RREQ when they move in order to perform path maintenance. Every so often each node sends out hello messages to main symmetric links to its neighbors which provides for timely and consistent operation of the protocol as new requests come in.

Once a node or the next hop becomes unreachable, the previous node on the path sends an unsolicited RREP back towards the source with a hop count value of infinity and an incremented sequence number. This table entry is updated at each hop along the way, signifying that the destination is no longer accessible. Once a source node receives this notification, it can restart the path discovery process if it chooses.

Nodes manage connections in their local "neighborhood", or within the distance that their physical medium can effectively communicate. Nodes find their neighbors in one of two ways. Periodically a node can emit a special RREP packet with a time to live (TTL) of 1 that all of its neighbors will hear and update their routing tables to reflect the existence of the neighbors node. Also, a node updates its routing table when it hears or overhears a broadcast from a node within its range. Only node links which can be traverses or used in a bidirectional manner are employed by AODV and considered to be neighbors of the node.

On-demand routing protocols such as AODV and DSDV have been shown to perform better with significantly lower overheads than proactive routing protocols in many situations [26, 34, 35, 36]. They are able to react quickly to the many changes that may occur in the network topology while reducing routing overhead in regions of the network where change occurring less frequently.

2.2.3 OLSR

The Optimized Link State Routing Protocol (OLSR) [41] is a proactive IP routing protocol meant for ad hoc wireless networks. OLSR performs topology flooding using a reliable algorithm. Topology flooding is a technique based on periodically sending out a node's link state information to its neighbors. This algorithm is not aimed at reliability, yet simply floods the network often enough to make sure each node's graph state does not stay unsynchronized for long periods of time. As each node notices its neighbor nodes, it distributes the links via flooding and then optimizes its graph state.

OLSR maintains graph state by having each node send out hello messages to specific nodes in the network to exchange neighborhood information. This hello packet includes the node's IP, sequence number, and a list of distance information for its neighbors. Each node rebuilds its own routing information once it receives an update this hello packet and recalculates the shortest path to every node in the graph. This routing information is only updated when:

- A change in the node's local neighborhood is detected
- A route to another node has expired
- A better or shorter route has been detected for a destination

OLSR employs hello messages to find one hop neighbors, as well as its two hop neighbors via their responses. It generates a constant overhead of control traffic. Being that OLSR is a proactive protocol it actively finds routes to all destinations. It maintains this graph topology as well as routing information actively at all times. With this style technique there is no route discovery delay, with routes to all destinations are created and

maintained before used. The routing overhead generated does not increase with the number of routes used, although it is generally greater than that of a reactive protocol.

In order to flood topology information OLSR designates a router on every link for this function. Using hello messages to discover the two hop neighbors, OLSR performs a distributed election of a set of multipoint relays (MPRs). The usage of MPRs helps to minimize the flooding of broadcast messages in the network and reduce the size of hello messages. Topology control (TC) messages which contain the MPR selectors are forwarded by the MPR nodes. This gives OLSR some unique attributes as only a subset of nodes source link state information is shared among all nodes and not all links of a node are advertised but only those which represent MPR selections.

A common criticism of OLSR is the fact that there is no way for link quality to be sensed. It was assumed that if hello messages had been received recently over a link that the link was considered to be up. Given that OLSR is a proactive routing protocol it uses relatively more power and computation than reactive protocols, thus making it less fit for sensor networks or meshes of small battery powered devices. OLSR also uses a relatively large amount of bandwidth and computation to construct its optimal paths through the network topology. Transient loops can also result from packet loss coupled with inconsistent state held for the network link state database. Given that link state routing requires the topology database to be synchronized across the network and reliable flooding is very difficult in an ad hoc network, OLSR doesn’t bother with reliability; It simply floods the necessary data for the topology database often enough to make sure the database does not remain unsynchronized for extended periods of time. Some primary advantages of OLSR are:

- Minimal latency
- Ideal in high density and large networks
- OLSR achieves more efficiency than classic link state algorithms when networks are dense
- OLSR avoids the extra work of “finding” the destination by retaining a routing entry for each destination all the time, providing for low single-packet transmission latency

OLSR’s disadvantages include:

- When the network is sparse, every neighbor of a node becomes a multipoint replay, reducing OLSR to a pure link state protocol
- High control overhead (reduced by MPR usage)
- Higher computation
- Storage
- Implementation complexity

2.3 Summary

We’ve taken a look at four algorithms for routing in MANETs, each with its own design tradeoffs, strengths, and weaknesses. Table 1 summarizes the strengths and weaknesses of the four routing algorithms we reviewed.

In the next section we will take a look at a routing algorithm, Termite, which improves on some of the weaknesses of the referenced routing algorithms. MANET networks are generally based on distance vector or link state routing algorithms created around fifty years ago [29, 42, 43]. AODV is of particular interest when discussing

Termite as Termite can be viewed as a probabilistic version of AODV in many ways. The AODV routing protocol is one of the most popular on demand routing protocols for ad-hoc networks. It was originally based on DSDV and was originally introduced in 1999 by Perkins and Royer [26]. AODV is in the continuing stages of being standardized by the IETF MANET working group [44].

Networks such as AODV and OLSR aim to find good routing solutions through the network topology. This approach, while effective in stable conditions, can see its performance degrade significantly in high dynamic environments where many routing updates are needed due to a changing network topology. Termite has the ability to self organize in an inherent fashion due to its use of pheromone and stigmergy. As packets move through the network each node updates its pheromone table which serves as a continual source of online feedback. This allows Termite to be more effective as a routing algorithm in chaotic environments.

Routing Algorithm	Notes	Drawbacks
DSR	Reactive, Forms route on demand, path in header of packet. Low overhead during periods of low node movement.	Path size limited by size of packet.
DSDV	Proactive, Table driven, distance vector, distributed version of the shortest path problem.	Requires a regular update of its routing tables, creates power drain.
AODV	Reactive, distance vector, Similar to DSR in that it builds "paths". Routes packet to next hop, which holds pointer to next hop on pre setup path.	Multiple RREPs for a single RREQ can lead to heavy control overhead. Periodic beaconing leads to unnecessary bandwidth consumption.
OLSR	Proactive, Performs topology flooding with a reliable algorithm. Ideal in high density networks.	Excessive flooding can use a large amount of bandwidth and CPU resources.

Table 1: Strengths and weaknesses of the four routing algorithms we review.

CHAPTER 3

Termite

3.1 Introduction

Termite [3] is a biologically inspired routing algorithm that was based on previous work in adapting the effects of stigmergy [4]. Stigmergy is defined as information gathered from work in progress [5]. Termite models how termite colonies lay pheromone to communicate in a decentralized manner. The concept of a pheromone value is taken from the social insect world where insects such as termites lay small deposits of pheromone as they move to indirectly communicate with one another. The reliance on pheromone as a routing metric estimator creates a mechanic by which the network can continuously adjust and balance itself to best meet the chaotic nature found in MANETs. Termite has an inherent stochastic nature while employing a distance vector strategy to route its packets while adapting to changing conditions in the network. Termite is a reactive routing algorithm that uses a routing metric based on a pheromone values carried by each packet across the network.

The Termite algorithm [3] is very interesting from the perspective of its "on-line" training of the network via routing information "piggybacking" on top of normal network traffic, as opposed to the more common technique of sending packets for the purpose of purely updating routing information. All routing information is carried on each normal packet as a pheromone factor, and this factor degrades each time the packet hops. At each hop, the amount of pheromone on the packet is used to update the node's internal routing or pheromone table. As packets move through the network, at each hop they are routed

independently based on a table of pheromone values. This table is generated by a set of probabilistic equations that govern the buildup and decay of destination links and their associated neighbors. Termite's internal routing tables are very tuned towards how packets move as observed locally as opposed to relying on the explicit routing information received from other neighbor nodes. This also provides for a much lower overhead in terms of control packets. The pheromone for each link is constantly decaying towards a value of zero, yet as traffic comes in across that link, information is built up by the pheromone value on the packet itself. The pheromone table is updated in a manner such that the link updated is where the packet came from, not where it is headed. The update to the pheromone table constructs a local map of the network traffic in reverse.

As each packet moves from node to node, it deposits its pheromone in each node's pheromone table for the previous hop's link. High levels of pheromone in an area will attract more termites to the area while the continuous evaporation of pheromone eases the attraction towards the area for the termites. The pheromone is built up leading back towards the source for a packet as it moves through the network. As packets move through the network, pheromone continuously builds up and decays with high quality paths emerging through the network. Each node periodically updates its pheromone table to decay the pheromone on each destination via neighbor link. The pheromone buildup and decay mechanics allow the network to not become locked in on any one good solution. These mechanics of attraction and repulsion are used in Termite to drive the flow of network traffic across the network. Exploring the network in a continuous manner also helps adaptation of the network in chaotic and uncertain routing conditions.

Termite minimizes control overhead as it distributes routing information with pheromone carried on data packets. Routing algorithms such as DSDV and OLSR send out control packets specifically meant to distribute state information. Since control overhead is minimized, this adaptive approach to routing has an increased ability to maintain network performance in a distributed manner. To understand what pheromone is and how it drives self organization within MANETs, we have to take a look at the underlying principles of self organization.

3.2 Principles of Self Organization

In this section we give a quick review of some of the core properties of self organization. Self organizing systems exhibit emergent properties, where emergence refers to a process by which a system of interacting agents or nodes acquires qualitatively new properties that cannot be understood as the simple addition of their individual contributions [5]. These emergent properties arise unexpectedly from nonlinear interactions among a system's components. Case studies such as bark beetle larvae clustering [45] show how interactive components in nature demonstrate these processes. Self organization is prevalent across the natural world and permeates most every corner of nature.

Termite takes a strong cue from the mechanics of termite hill construction. Termites, just like all other social insects, have no centralized control, blueprints, or coordinators. All of their work coordination is done by examining their localized environment and using that as input into a rule base which tells them which task to perform (sometimes a repeated encounter of a stimulus is necessary to perform a task

switch). This process is generally referred to as stigmergy, and is the method by which workers communicate to one another by incomplete work left in their environment. Positive and negative feedback are used as agents of amplification and control respectively. Feedback and stigmergy are two core principles that drive and control self organizing processes and are discussed in the following sections.

3.2.1 Feedback

Feedback is a driving mechanic in the process of self organization. Positive feedback generally promotes change in a system where negative feedback slows down a reactive process. A termite's attraction towards large piles of pebbles biases it towards adding to large piles, which is positive feedback. Large piles tend to have more pheromone which attracts more termites that in turn potentially drop more pebbles on the pile. The amplifying effect of positive feedback takes an initial change in a system and reinforces that change in the same vector as the original deviation. Positive feedback amplifies fluctuations in populations in interacting subunits. The nature of positive feedback implies that it has the potential to cause large scale and unwieldy implosions or explosions in the pattern where it plays a role. Negative feedback is the mechanism by which we keep positive feedback and amplification under control.

With social insect societies, such as termites, evaporation provides the system a source of negative feedback, as it weakens the pheromone over time resulting in less gradient. As the pheromone gradient weakens, it will attract less termites and therefore less pebbles will be deposited in the area. Negative feedback is what keeps our system from converging on sub-optimal solutions too quickly. Biological termites employ

negative feedback in the form of pheromone evaporation which keeps their pebble piles from clustering in many small piles too quickly.

3.2.2 Stigmergy

Stigmergy is defined as information gathered from work in progress [5]. Termite is based off of the classic effect of stigmergy, although more towards qualitative stigmergy than the more common quantitative stigmergy found in ant colonies [4]. Stigmergy refers to indirect communications between individuals, generally through their environment. Qualitative stigmergy is based on taking into account the relative quality of the stimulus encountered. Quantitative stigmergy is based on the number of successive encounters of a type of stimulus. Self organizing systems can also use information acquired directly from other individuals. Stigmergy allows for scalable decentralized coordination amongst relatively unsophisticated agents. Top down control hierarchies tend to lose the ability to efficiently manage large groups of agents whereas self organizing processes scale with much success [5]. In termite colonies, pheromone is employed to create the stigmergic effect. Termites lay pheromone in order to cluster pebbles to create their nest and chambers. In the next section, we take a look at the specific mechanics of pheromone and how it works with Termite.

3.3 Pheromone

Pheromone is the biologically inspired aspect of Termite that creates a probabilistic goodness-map of quality pathways through the ad-hoc network based on topology and network conditions. Termite runs in an on-line manner, so all packets carry pheromone on them and update each node’s pheromone table along their path. The

amount of pheromone on a neighbor link for a destination in a node's neighbor table indicates the quality of the route for that destination via that neighbor as the next hop.

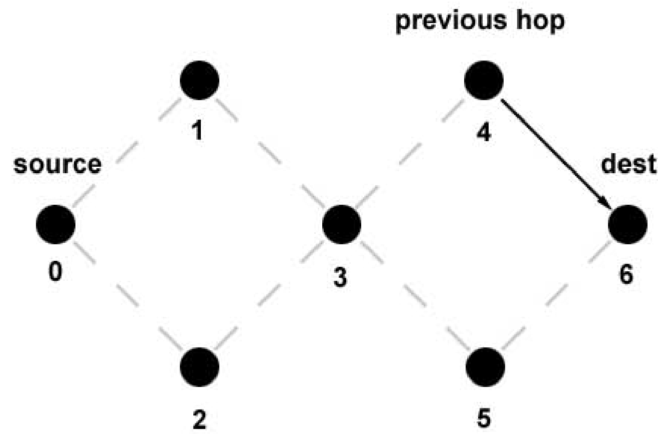

Figure 2: Packet movement through Termite network. A packet moves from node 4 to node 6 on its final hop to the destination node 6.

destinations

neighbors	0	1	2	3
4	**0.6**	0.7	0.2	0.1
5	0.4	0.3	0.8	0.9

Figure 3: Pheromone table update. As the packet in Figure 2 hops to node 6, node 6 updates its pheromone table to reflect the incoming packet's information. The pheromone table is updated such that neighbor node 4's destination entry for node 0 (source node) has its value adjusted. The rows in node 6's pheromone table reflect the nodes that node 6 can hear locally. The destinations are nodes that node 6 knows about.

Each node in the network has a pheromone table which contains a matrix of values for neighbor and destination pairs. As neighbors are discovered, they are entered as a new row in the matrix. As destinations are discovered, they are entered as a column

for a neighbor. Each neighbor-destination pair has a pheromone value that indicates the goodness rating for that particular next hop neighbor for the intended destination. In Figure 2 above we show a sample network with a packet hopping from node 4 to node 6. In Figure 3 we see the pheromone table entry for the neighbor node 4 and destination node 0 (the source of the packet) being updated. Periodically each value in the matrix is degraded according to an equation described in Termite [3]. This simulates the effect of pheromone evaporation and allows the network to continuously explore alternative paths while not prematurely converging on solution. Once the amount of pheromone decays to zero for a destination cell on a neighbor row in the pheromone table, the destination entry is removed for that neighbor.

The routing mechanics in Termite are such that each packet is routed on a per hop basis. As packets arrive the pheromone value for the source via the previous hop are updated. Termite creates a pheromone map of the network in reverse as the source is treated as the destination and the previous hop is treated as the next hop in the local pheromone map. Packets moving towards the source of the current packet will now be affected by the information carried from the current packet's path. In Figure 2 above, as packets move through node 4 to node 6, node 6's pheromone table is updated to reflect the information carried from source node 0.

Once pheromone accounting has been taken care of, the node then checks to see if the packet's destination exists in its pheromone table. If it does, then the packet is given a next hop based on the sets of probabilistic equations described in Termite [3] and is sent onwards towards its next hop. The routing mechanics of Termite are illustrated in Figure 4 below. If the node does not have a destination cached in its pheromone table, then the

packet is queued and a RREQ packet is generated for the desired destination. The RREQ packet hops from node to node where each node checks to see if it has an entry in its pheromone table for the destination. If the node has an entry, it generates a RREP packet and sends it back towards the source of the RREQ packet. The RREP packet follows the source trail of pheromone back to the source of the RREQ. Once the originating node receives the RREP packet it updates its pheromone table and then forwards the original packet onwards with a new next hop based on the information returned from the RREP.

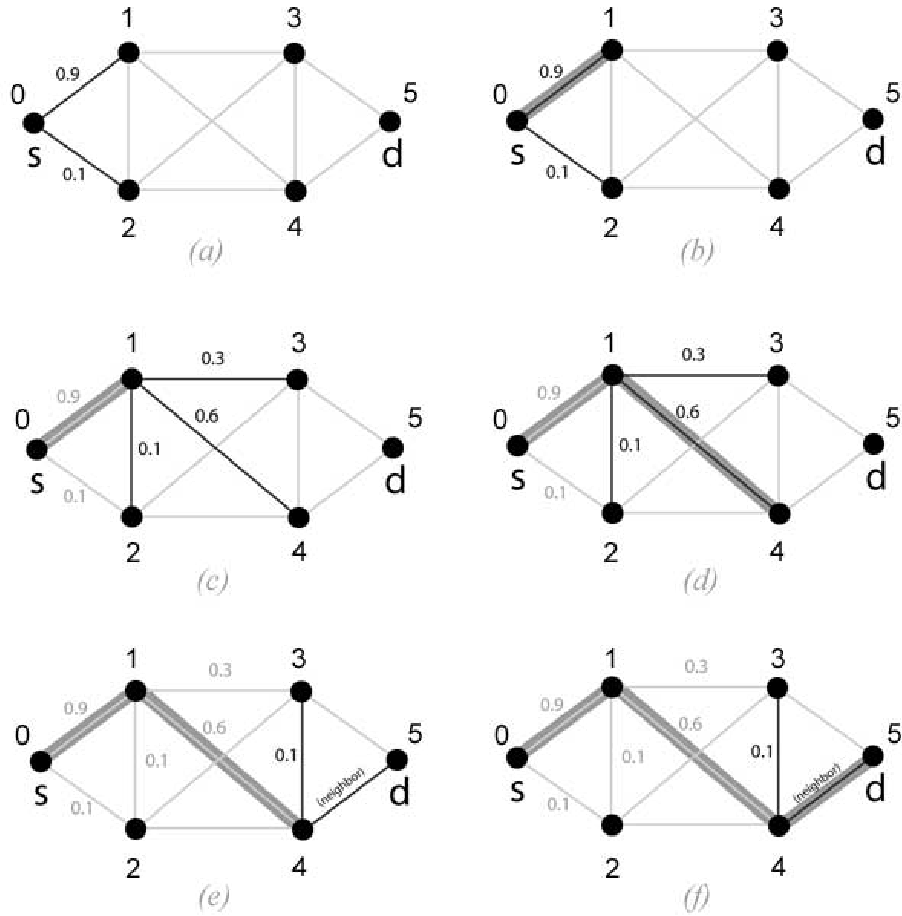

Figure 4: Routing in Termite. In (a) our source node 0 has 2 neighbors to choose from. The neighbor link for node 1 has a pheromone value of 0.9 and the link for node 2 has a value of 0.1. The next hop is selected by a set of probabilistic equations and in this case node 0 routed the packet via node 1 in (b). In (c) node 1 has 3 possible next hop neighbors excluding the previous hop from node 0. Again the next hop is generated by the set of probabilistic equations and node 4 is selected as shown in (d). In (e) we see that node 4 has 2 other possible neighbor hops other than its previous hop. Node 4 determines that neighbor 5 is the destination node, and forwards the packet in (f).

3.4 Packet Types and Layout

Termite employs 3 types of packets:

- DATA – a packet carrying actual data
- RREQ – a route request packet
- RREP – a route reply packet

Some routing protocols use a Hello packet to advertise the existence of a node. In Termite this is accomplished with a RREQ setup with a route request for sending node itself. In the initial stage a newly powered on node will send out this pseudo hello packet with a 1 hop Time to Live (TTL).

Bit Offset	0-3	4-7	8-15	16-31	
0	Source IP Address				
32	Destination IP Address				
64	Previous Hop IP Address				
96	Next Hop IP Address				
128	Pheromone Amount				
160	Packet Type		Packet TTL	Data Length	Flags
192+	Data				

Figure 5: Termite packet layout. Bit offset describes the starting bit for the row relative to the first bit.

A Termite packet is represented by an array of memory with bit ranges being slotted for the different values to be transmitted. The bit offset in Figure 5 above indicates where

the value starts in relation to the beginning of the byte array. Termite has multiple types of packets but all packet types have the same packet format. The first four fields are each 32-bit IP-addresses for Source IP Address, Destination IP Address, Previous Hop IP Address, and Next Hop IP Address. The next field is a 32-bit value representing the amount of pheromone the packet carries. This value is read at each hop and updated before being sent out again towards its next hop. The pheromone field is followed by four more 8-bit values Packet Type, Packet TTL, Data Length, and Flags.

The packet type field indicates whether the packet is a Data, RREQ, or RREP packet. The packet TTL field indicates a TTL value, or how many more hops the packet can travel before being dropped. This value is decremented at each hop. The data length field indicates the size of the data payload region of the packet. The data payload region of the packet, as shown in figure 5 above, contains the data for the Data packets. This region is not used for RREQ or RREP packets. The flags field is specified in the Termite paper but does not have a specific use listed.

3.5 Comparison to AODV

Termite can be considered a probabilistic version of AODV, as they share many common traits. Termite employs an order of magnitude less control overhead as it does not attempt to maintain strict paths for routing. It does employ the RREQ / RREP packet overhead for path discovery as AODV does, but it does not flood the network with overhead for new route discovery every time a route breaks.

Termite is compared with AODV with respect to the following criteria:

- Data Goodput – the fraction of successfully delivered data packets.
- Control Packet Overhead – measures the ratio of control packets to the total number of transmitted packets in the system.
- Control Packet Distribution – measures how many of each type of control packet were transmitted.
- Medium Load – the ratio of data packets successfully delivered versus the total number of packet transmissions. It characterizes how inefficient an algorithm is in delivering packets.
- Medium Efficiency – the ratio of the number of transmissions of successfully arriving data packets to the number of total packet transmissions, where multiple transmissions of the same packet are counted individually.
- Medium Inefficiency – the ratio of transmitted packets versus the number of packets offered to the network for delivery.
- Link Failure Rate – the average number of links that are lost per node per second. It is a measure of how chaotic the network is and how much time each node has to acquire a routing solution before the topology will change again.
- End-To-End Delay – the average of the delay for all packets delivered, where a lower number if better.

The results in [3] demonstrated that Termite delivers 95% of the packets while AODV delivered 90% of the packets. The original paper also compared Data Goodput versus Node Speed, where Termite again outperformed AODV. Termite has an order of magnitude less control packet overhead and produces a nearly constant amount over a large range of node mobility. Whenever there is a route break, AODV must issue a route

discovery flood. However with Termite a full route discovery is almost never needed given its retransmission link repair policy. With AODV the proportion of control packets increases with node speed because links break more often and the limiting factor is the number of RREQ packets. Termite generates more RREPs than RREQs due to a liberal route reply policy and its route caching mechanisms. Termite works better as more pheromone is seeded around the network as it provides hints on regions the destinations might be located.

With regards to medium load Termite was able to beat AODV. Both algorithms show an increasing load on the medium as the network volatility increases. For AODV this is caused by the increasingly large amount of control traffic generated. For Termite, this is a result of the fact that data packets must take longer paths to find their destination because the network topology is changing faster. With regards to medium efficiency, Termite is able to easily deliver packets at low node speeds. Both AODV and Termite have to expend more resources to deliver a packet as node speed increases. Medium efficiency for both routing algorithms decreases linearly as node speed increases. Both Termite and AODV performed at a similar level with respect to medium inefficiency. Termite and AODV also had similar results when compared for link failure rate. For end-to-end delay AODV has a constant performance due to the fact that packets are only sent when a full route is known to exist. Termite's delay results are not positive given the fact that AODV's delay is nearly an order of a maginude lower at higher speeds. Table 2 summarizes the comparison of Termite with AODV. As we can see, Termite compares well with and exceeds AODV in many ways. Given these measures we chose Termite as

our base routing algorithm. The robustness and flexibility of Termite gave us a good foundation to improve on and secure with TinyTermite.

Metric	**Comparison**
Data Goodput	Termite delivers 95% of the packets while AODV delivered 90% of the packets.
Control Packet Overhead	Termite has an order of magnitude less control packet overhead.
Control Packet Distribution	Termite issues more RREPs than RREQs due to a liberal route reply policy. However, Termite still uses much less control traffic compared to AODV.
Medium Load	Outperforms AODV, but both algorithms struggle in this area as network volatility increases.
Medium Efficiency	Termite outperforms at low node movement speeds, but both algorithms struggle as node speed increases.
Medium Inefficiency	Outperforms AODV, but both algorithms tend to struggle as node speed increases.
Link Failure Rate	Termite performs very similarly to AODV
End-to-End Delay	AODV's delay is nearly an order of a maginude lower at higher speeds.

Table 2: Comparison of Termite and AODV with respect to eight key measures.

CHAPTER 4

TinyTermite

4.1 Introduction

TinyTermite is a fully implemented routing algorithm for MANETs that is secure against a replay attack [6] in conjunction with a selective forwarding attack. The two major contributions of this work are adding security to the Termite algorithm and the implementation of Termite on a MANET. We propose a technique called "suspicion" to defend against replay attacks. We also implemented Termite on the Berkley mote sensor mote variant Imote2 [7] under the TinyOS operating system [8] and with the nesC programming language [9]. We extended Termite with the suspicion technique to create TinyTermite.

4.2 Contribution

Our contributions are the addition of suspicion and the implementation of Termite on the Imote2 platform. Suspicion is a technique inspired by task response thresholds in ant colonies which allows an ad hoc network to detect certain classes of attacks even given inconclusive information and alter its behavior in a proportional response. In our experiments we implemented a detection scheme for the replay attack and show how countermeasures can be employed in TinyTermite. We also propose an alternative method to deal with selective forwarding attacks based on disjointed paths taken by packets. The other major component of our contribution is the implementation of Termite on the TinyOS based platform and the measurement of the system's performance in a

real-world setting. (We also measure some of the properties of the Imote2 platform such as raw throughput and note the realities of implementing on this platform.)

4.3 Attacks On MANETs

Most MANETs by default are very insecure and are susceptible to common attacks. Our major focus with this project was to develop some defense mechanisms against the more basic attacks and provide a foundation to build on. Our primary addition was focused around adding an aspect to each neighbor entry in the pheromone table called “suspicion”. Termite can be disrupted in a number of ways such as selective forwarding and replay attacks. Security in MANETs is viewed from the perspective of confidentiality, integrity, availability, authentication, and reliability. We chose the replay and selective forwarding attacks to demonstrate TinyTermite’s effectiveness with respect to network integrity and availability.

A network’s integrity and reliability are compromised with a selective forwarding attack. With a replay attack, the network’s availability is challenged as resources are drained during the attack and not used to service normal network behavior. Although Termite cannot prevent these attacks, its inherent nature minimizes their affect. One possible solution is to use link layer security [46] with Termite, which can help protect against certain vulnerabilities, but it cannot eliminate them. A routing algorithm must be designed with security in mind.

A replay attack can be devastating to a MANET given that resources are limited and flooding a node quickly drains these resources. Using this strategy a rogue node can quickly effectively disable many nodes in a MANET. Another attack we consider is the

selective forwarding attack where a rogue node either only forwards a subset of packets or no packets. In our experimental setup, our rogue node uses both of these attacks to probabilistically route more packets through it and then not route them at all. This is shown to be quite disruptive to the network.

4.3.1 Replay Attack

In a replay attack (Figure 6 below), a rogue node will capture or create a valid packet and repeatedly send this packet to a neighbor. To the neighbor, the packet will have valid fields and or cryptographic information as the rogue node has not changed any of the bits in the packet. However, the unassuming neighbor node still has to use resources to process each incoming replayed packet. In a setup where the node is running off limited battery power, this can quickly deplete resources and effectively disable the neighbor node.

With Termite, incoming packets influence a node's pheromone table. Each outbound packet for this node is then influenced by the same pheromone table. If a replayed packet continuously comes from a neighbor rogue node, it will heavily influence the pheromone table of the good node. The good node will quickly rate the rogue node as a good next hop for the source node listed in the incoming packets, which sets the rogue node up for more nefarious behavior as it now sits on a high quality route for the network towards a particular destination. Once this situation is setup, the rogue node can then execute a selective forwarding attack where it only forwards a subset of the received packets or none of them at all. To counter this situation and create a basis to counter other attacks, we introduced the technique of Suspicion.

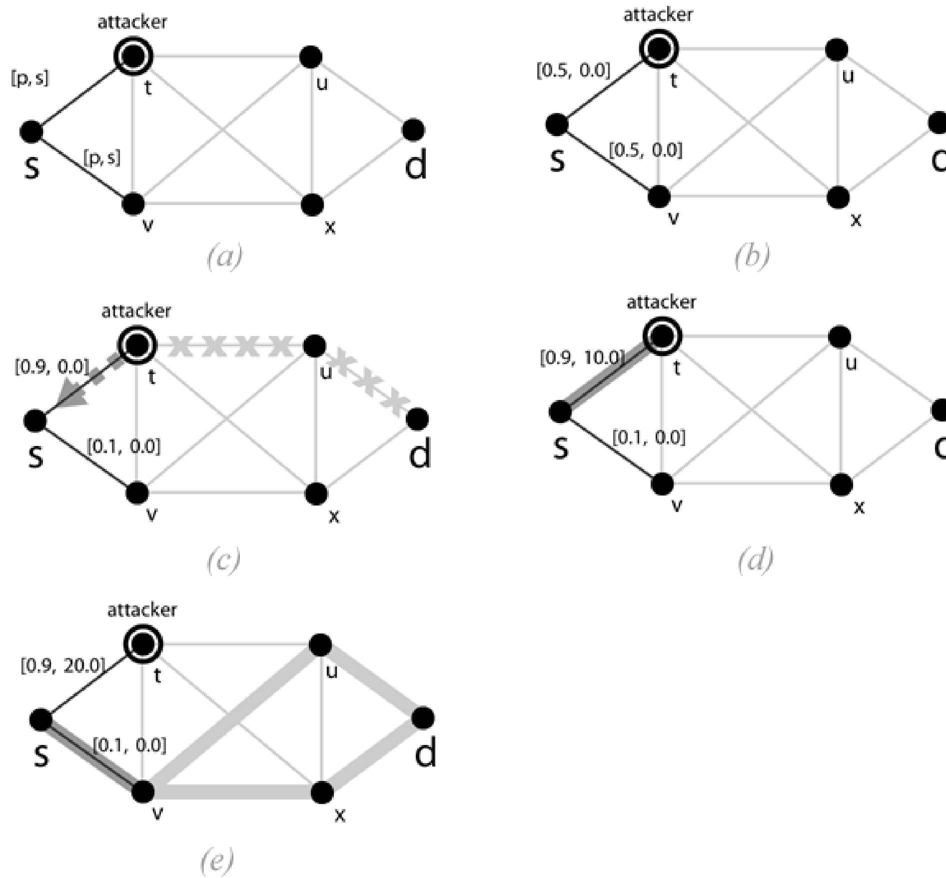

Figure 6: Replay attack diagram. In (a) we have a source node s wanting to send data to destination node *d*. Source node *s* has two neighbors, node t and node v. Each link in source node *s*'s pheromone table has a pheromone value *p* and a suspicion value *s* denoted as [*p, s*]. Node *t* is the attacker. In (b) we see that each neighbor entry in the source node's pheromone table has an equal amount (0.5) of pheromone and no (0.0) suspicion. In (c) the attacker begins sending a valid or captured packet to source node *s*. The packets appear to have come from source node *d* with the attacker as the previous hop and quickly raise the pheromone level for the traversed neighbor hop in source node *s*'s pheromone table. In (d) we see where node s has engaged the suspicion countermeasure, detecting the replayed packet. The suspicion factor for the neighbor link quickly rises towards the threshold level. In (e) the suspicion factor on the neighbor link to the attacker is above the threshold and node *s* is now actively routing packets around the attacker towards the destination *d*.

In order to come up with Suspicion, we drew on inspiration from the fields of Bayesian networks [47], Neural networks [48], and division of labor and task allocation in ant colonies as well as the mechanics of pheromone deposit and decay [3]. Bayesian networks are interesting in this situation because they take incomplete information and

determine a decision. Neural networks share a similar topology to a mesh of nodes, and output "activates" based on the strength of the signal coming and the weight on the connection. These provided interesting inspiration for the effect of what we wanted, but our major influence became how ants divide labor and balance their task allocations in real time without centralized control as well as the utility of pheromone deposit and decay over time.

With Termite each neighbor link was simply understood to exist or not exist, and the strength of the connection was based on the amount of pheromone on the link. Our model creates a more sophisticated neighbor link relationship as we give each neighbor a level of "suspicion". This suspicion factor is similar to the pheromone used for each destination, but it represents a small amount of temporally affected information about how much we trust the neighbor or are more worried about their recent activities. The difficult part in most MANET algorithms is that it becomes very difficult to separate out the good input (normal control traffic, data packets) from the bad (replayed packets, compromised packets) especially in such an uncertain environment with only local information. The one thing that TinyTermite does focus on with its suspicious functionality was that we wanted to be able to define certain types of observable neighbor behavior that might pique a node's interest, and as that behavior continued, we wanted it to affect how that node dealt with the offending node. However, if the behavior was simply a result of any number of normal but fluctuating traffic effects, then we wanted the node to ease its interest or penalties on the offending node. With this in mind we combined the effects of an ant colony task stimulus threshold mechanic with the properties of pheromone deposit and decay to for our suspicion mechanic.

Each neighbor link has an associated threshold factor for suspicious input. When the suspicion factor on that link exceeds the threshold, the node then may take defensive precautions that attempt to preserve its own operation or pro-active attack operations that seek to disable or combat the offending node. Another more latent use of the suspicion factor is another weight on the normal termite routing algorithm, however we leave that for future experimentation. The following equations represent the suspicion build up and decay mechanics of TinyTermite.

$$S \leftarrow S + y \qquad (1)$$

$$S \leftarrow S - d \qquad (2)$$

In (1), S represents the level of suspicion on the link to neighbor r and y represents the constant amount of suspicion that is being deposited on this link. This equation is executed anytime a stimulus occurs that was predefined as a suspicious behavior for a neighbor node. Once the buildup of suspicion on the link surpasses a threshold level, our node ceases communication with the neighbor node until the suspicion decays below the threshold level. This type of mechanic allows for a "forgive but not forget" system.

Equation (2) deals with how a TinyTermite node degrades the buildup of suspicion on a neighbor link over time. At each timestep this equation is applied to each existing neighbor link in the routing or pheromone table. Variable S refers to the suspicion level on node r and d is the constant amount by which the suspicion factor degrades. Although this is a simple set of linear equations, their effect allow for a quick buildup and slow back off of suspicion levels given the proper parameters. For our experimentation, we used $y = 7$ and $d = 1$ with a threshold for suspicion of 20. We also

capped the maximum amount of suspicion on a neighbor link to 100, and if the amount of suspicion on a link was above the threshold then pheromone could not be laid on the link but could continue to evaporate. The suspicion could then continue to buildup past the threshold but our routing system will ignore any next hop node with a suspicion level beyond the threshold. Another slight change is that a link was not removed until all pheromone had evaporated and the suspicion was under the threshold so as to create a memory of a rogue neighbor even after the pheromone was gone. This gave the link a zero percent chance of being selected as a next hop for a packet while keeping it in a suspended state. We used round integers in this case since it is easier to work with for a base case experiment.

To implement the detection scheme for this stimulus, we implemented a seen packet list. (This suspicious input could be the rate at which packets are arriving, the size of the packets, or malformed packets.) This list of packets allows the node to detect any repeated packets that have occurred in the last N seconds, where N is typically the route timeout factor used in routing the packets. For our experiment we used a packet timeout of 1000ms. As packets come in from neighboring nodes they are checked against our seen packet list. The Imote2 throughput theoretically is 250 kbps, which is equivalent to 32 KB/s. At 32 KB/s and 128 bytes per packet we can have 256 maximum messages per second ((32 *1024) / 128 = 256) that the node could possible hear. Our packet timeout is 1 second so we should see at the most (1 x 256 = 256) messages at any given moment. Each forwarding structure is 9 bytes, so (9 x 256 = 2304 bytes) of space taken up to track messages coming through our node.

4.3.2 Selective Forwarding

A selective forwarding attack [6] is performed by a node either forwarding only a subset of the packets sent through it, or forwarding none. Highly decentralized routing algorithms such as Termite depend on all nodes acting reliably to achieve nominal operations. Selective forwarding attacks have been shown to be countered by multipath routing [6]. In this thesis, we introduce a scheme using dual packets to route packets in an edge disjoint manner. (Two edge disjoint paths consist of two paths that can share vertexes or nodes but cannot share edges.) Termite coupled with edge disjoint paths make it difficult for attackers to completely disrupt the network. This is due to the fact that each packet has a copy of it moving through the network and Termite routes packets in a probabilistic manner.

Our design for multipath routing was setup by a bit flag in the packet header to indicated the *A* or *B* packet and then a seen packet list. TinyTermite reserves 8 bits of the packet header for a series of flags as shown below in Figure 7. We designate one of the bits in the flag region for the *A* / *B* flag which indicates whether the packet is the *A* or *B* version of a packet pair in the dual packet scheme. Each node in TinyTermite maintains a list of packets it has seen recently as well as which neighbor the packet took for its next hop. The source node for a packet sends out two copies of the same packet each with the second copy having the *A* / *B* bit flag set to 1. As each packet is forwarded, the forwarding node checks this seen packet list to see if it has seen the packet before. If the node has seen the packet before, then it forwards the packet to a next hop that ensures an edge disjoint path. We implemented the packet flag and the seen packet list, but did not implement and test the actual routing modifications due to time constraints.

<table>
<tr><td>Bit Offset</td><td>0-3</td><td>4-7</td><td colspan="8">8-15</td><td colspan="2">16-31</td></tr>
<tr><td>0</td><td colspan="10">Source Node ID</td><td colspan="2">Destination Node ID</td></tr>
<tr><td>32</td><td colspan="10">Previous Hop Node ID</td><td colspan="2">Next Hop Node ID</td></tr>
<tr><td>64</td><td colspan="2">Packet Type</td><td colspan="8">Packet TTL</td><td colspan="2">Pheromone Amount</td></tr>
<tr><td>96</td><td colspan="2">Sequence Number</td><td>AB</td><td>Thr</td><td>Lat</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>Data Length</td><td>Data</td></tr>
<tr><td>128+</td><td colspan="12">Data</td></tr>
</table>

Figure 7: TinyTermite packet layout. Bit offset describes the starting bit for the row relative to the first bit.

4.5 Implementation

TinyTermite is an implementation of the MANET routing algorithm Termite implemented on the TinyOS platform with the addition of the aforementioned security measures. This new system was implemented in around 10,000 lines of code and we made no modifications to the base radio stack or the TinyOS base code. We did change the base packet size from 29 bytes to 128 bytes, the maximum allowed by the CC2420 chip's hardware. TinyOS is an open source operating system and platform intended for wireless sensor networks. It is meant for embedded operating systems and employees the nesC programming language. TinyOS was designed with a focus on limited resources, reactive concurrency, flexibility, and low power constraints [8].

TinyOS features a component-based architecture which helps to minimize code size as required by the severe memory constraints inherent in sensor networks. TinyOS's component library includes distributed services, sensor drivers, network protocols, and data acquisition tools. These components can be extended or combined to create new components. TinyOS has an event-driven execution model which provides for fine-

grained power management while allowing the scheduling flexibility required by the chaotic nature of wireless communication. For this project we used the Crossbow Imote2 mote which has the Intel Xscale2 processor onboard as well as the Chipcon CC2420 WIFI chipset. The Imote2 runs at 415 MHZ and has 32 Megabytes of ram which makes it one of the top motes on the market today.

Due to the role that probability plays in the Termite routing equations and pheromone updates, any implementation of Termite needs to be able to deal with floating point numbers. The Xscale2 processor does not have a floating point unit included with its ALU, which presents a problem since allowing the system to do floating point calculations via software can be up to 100 times slower. Since many of the probabilistic equations are run on a per packet schedule, we needed to minimize the impact our floating point calculations would make on our performance due to energy and efficiency considerations. With efficiency in mind, we constructed all of our TinyTermite equations with fixed point operands. Fixed point is a mechanism by which we assign the floating point bit representation of a number to an integer with the decimal point set to the middle of the bits, and then allow the integer to be used inside the data path of the processor. The operations of add and subtract work the same for fixed point and integer value so they could be used without any changes. The multiply and divide operations required special functions to be done properly.

Another issue that we faced was exponentiation and floating point values in the probabilistic equations. In C libraries functions exist to do these approximations, yet with the TinyOS platform we did not have access to these functions. For the function of our prototype implementation, we implemented a solution that created an array of pre-

calculated values and a function which takes a floating point exponent and returns the nearest curve value for that value.

Given the nature of TinyOS's architecture, most programs built for it do not allocate memory at runtime since memory is such a limited resource. We implemented a minor memory management system for pre allocated packet queues such that the size of our queue was known at compile time, and linked lists were used to maintain a free list and an allocated list of packet slots. As a task requested a free packet slot from the queue, the linked list data structures are updated and a pointer to the free queue slot is returned to the task, which gave us a similar mechanic to using new and free in a traditional programming environment while keeping our memory usage in good bounds for our architecture.

We maintain two queues, the active-packet queue and the passive-packet queue. The active packet queue only contains packets that have already been assigned a next hop for their destination and are only waiting on the radio subsystem to be broadcasted over the wireless physical medium. Packets do not timeout in the active queue since they are already considered to be routed. The passive queue contains packets that we do not currently have next hops for given the destination. This queue is built the same way as the active queue and holds packets waiting on RREPs to be received for their destination. If a packet waits longer than the active timeout in this queue, it is removed and its slot is returned to the free list. Once a RREP is received, the pheromone table updated, and the packet is given a next hop for its destination, the packet is moved to the active queue. The separation of the passive and active queues allowed for tasks to scan shorter lists of packets which lowered task execution time.

Since TinyOS is an event driven architecture, the code had to be constructed differently than it would be for a traditional process based operating system. As events occur, the system will call events which preempt any executing task. For our code the main event was the message received callback function, and we had to be careful how we updated data structures inside the event method body since another task could be updating the same data structure or packet queue when it got preempted. Anything other than minor variable updating had to be done with a new task that was posted to the task queue.

The pheromone table was implemented as a simple pre allocated array of structs. Given our test bed of motes could not exceed a certain size, we could constrain the pheromone table maximum entries to an arbitrary size and mark entries as unused until they were needed. In an unbounded system of arbitrary size, this limitation could be overcome by simple dropping the lowest pheromone value destination or neighbor when a new one was added to a full table.

4.6 Testing Procedures

We tested and measured the TinyTermite performance implementation in four different ways:

- Single line test with preset path
- Thoughput with Termite routing engaged, 6 node graph
- Latency with Termite routing engaged, 6 node graph
- Attacker simulation using replay attack coupled with selective forwarding.

First, we measured the throughput of the network in a static route with multiple hops to see how the system degraded as load increased. Next we tested the network's throughput when the TinyTermite routing algorithm is used. Then we tested the latency of a packet going from source to destination and back to source. Lastly, we examined how a simple replay attack degraded system performance, and how the suspicion mechanic affected the system under attack.

The single line test bed (Figure 8 below) was setup such that we had N nodes in a line, where each node on average could only hear its immediate neighbor. Due to the properties of the wireless physical medium, sometimes nodes would not be able to hear anyone else, or would be able to overhead nodes multiple hops away. We spaced the nodes horizontally with a 11 to 12 inch gap between each node.

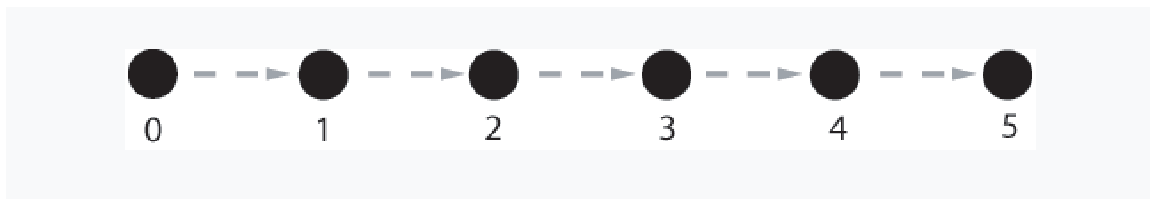

Figure 8: Single line test scenario. (For this setup, nodes are hard coded.)

Our routing tests for throughput, latency, and the attacker were measured using the six-node graph (Figure 9 below) where the source node is on the left side, the destination node is on the right side, and there are 4 nodes in the middle in a square formation between the source and destination.

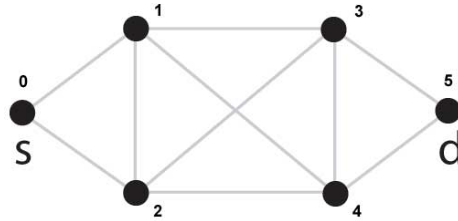

Figure 9: Six node graph for experimental setup to test throughput and latency.

Packets routed by the TinyTermite algorithm may take their own independent path through the network from the source to the destination. To measure the throughput, we count the number of packets received for the session at the destination node, and then multiply that number by the size of each packet to get a total number of bytes transmitted. We then take the timestamp of the last packet received for the session, and subtract the first packet's timestamp to get the elapsed time of the session. We then divide the total number of bytes transmitted by the elapsed time to get the system throughput, or number of bytes per second.

Latency is tested in a similar manner as throughput, using the same classic six-node graph in Figure 9. We send a RREQ out for the destination node, with a specialized flag set in the packet so we can identify it on return. The destination sends back a RREP with the latency flag set in the packet header, and when the RREP is received by the source node, the difference between the sent timestamp and the received timestamp is calculated.

We tested the suspicion aspect of TinyTermite to see how it performed with an active attacker. The attacker periodically sends a data packet that is marked as being from the destination so as to update the pheromone table and quickly build up pheromone for

the attacker neighbor as a good next hop candidate for the destination. First we ran the six-node routing simulation with no suspicion and the attacker sending out a data packet in a replay attack (in conjunction with a sinkhole attack) to measure how many packets were successfully sent from source to destination compared to the simulation with no attacker. Then, we ran the simulation with suspicion turned on, and compared the results. One of the four nodes in the middle (Figure 6 (a)) acted as the rogue node and executed special routines which attacked the source in an attempt to derail normal operation of the network. With a replay attack a node simply sends out a captured packet over and over again as it is considered by the rest of the network to be a valid packet. Replaying the packet at high rates can degrade overall quality of the network and drain valuable resources in a wireless ad hoc network. The rogue node influences how Termite views its local neighbors by giving more weight to the neighbor with the replay attack as this influences the pheromone table. This makes the node more likely to send a packet via the rogue neighbor setting up the sinkhole attack. A sinkhole attack is when a node simply does not forward packets as a normal node would, which in this case provides a fairly effective attack as we demonstrate with our data.

Our rogue node was setup to send out a replayed packet once a second, and the source node queued five normal packets to be routed to the destination node once a second. The rogue and the source nodes both began sending packets at the same time. Each session 100 packets were sent, and the source node queued five packets up at a time until it had queued 100 total packets. We ran each set of experiments 10 times.

4.7 Results

We initially implemented the Termite routing algorithm and attempted to measure some basic throughput tests. These efforts resulted initially in inconsistent results. Rebuffed, we studied the system parameters, theoretical bounds, and physical limitations of the software and hardware that makes up the Imote2 radio subsystem in order to move forward and truly measure the platform. One of the major things we found in our early prototype was that we had trouble getting the basic routing mechanisms of TinyTermite working correctly due to high levels of dropped control packets.

Once we reviewed the state of our research and consulted with other universities doing TinyOS research, we concluded that we needed to study the rate at which we pushed bytes onto the radio subsystem and the effects of IEEE 802.11 interference. We found out from other researchers that a backoff or timeout period is needed after the radio subsystem reports that the radio is finished sending the bytes of the packet. Our other main obstacle to testing was radio interference.

We knew we had to find a way to configure the motes such that we could consistently and reliably reproduce all results. We also had to find a radio backoff window that would reliably send all packets yet not keep Termite packets waiting in queue unnecessarily. A radio backoff window is the number of milliseconds we wait until we send another packet after the radio reports the last send call as complete and we denote it as [min, max]. We obtain this value by generating a pseudo random number that lines in a range. We created a series of exploratory stress tests in order to build a rudimentary map of the performance of various configurations. We looked at the variables:

- Distance, layout and obstacles – simulating obstacles and interference
- Radio backoff window – the length of time before we can send another packet
- Radio power setting – what power to test on, and how this works with distance

Our first exploratory test was intended to gather some course data in order to get an idea of how to proceed. We used the single line network (Figure 8) with a spacing interval of 11 to 12 inches between nodes. This provided an easy visual designation and was simple to setup. Since data gathering tests are run many times, this gave us an easy way to lay out a group of motes in different configurations. Every mote was adjusted or shifted slightly between each run of a test as interference patterns could change greatly based on a slight difference in mote position on a flat surface. In Figures 10 and 11 below we see the average node radio broadcast distance in a single line setup (Figure 8). Figures 10 and 11 are versus radio powers of 1 and 2, respectively, based off our setup.

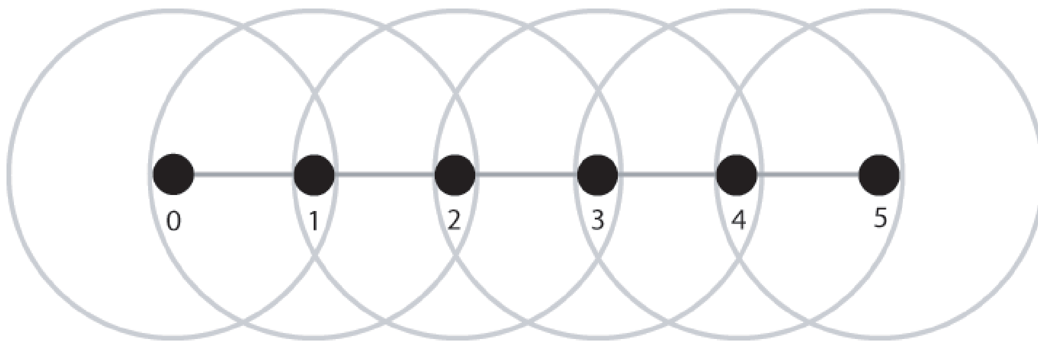

Figure 10: Showing radio range of nodes at power 1 in a single line setup.

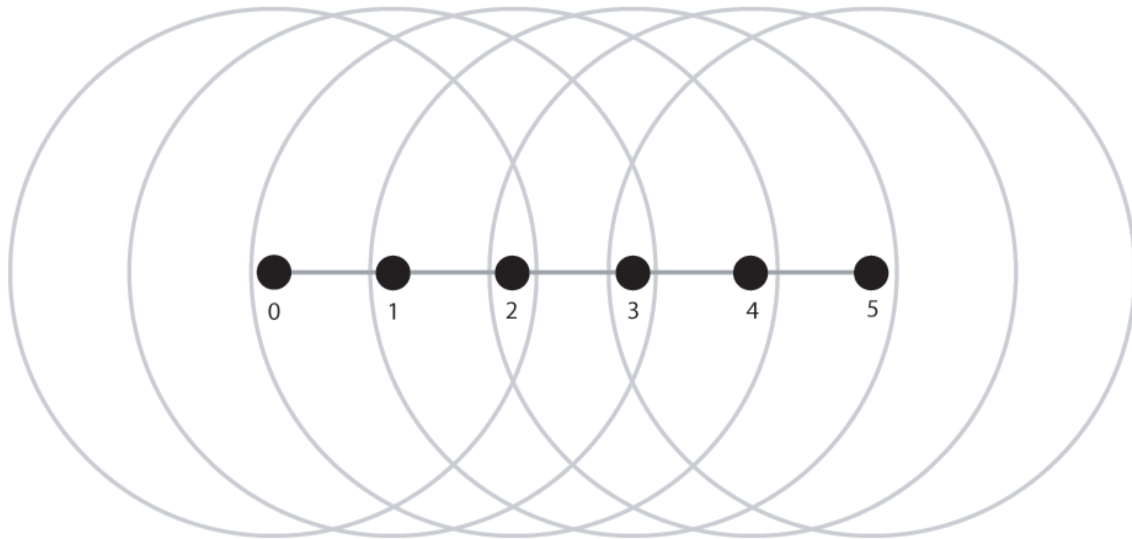

Figure 11: Show radio range of nodes at power 2 in a single line setup.

Our first basic stress test involved sending 500 packets through the system as fast as the radio subsystem would allow for given the radio backoff window. Our source mote would send a packet to the radio subsystem as soon as it reported done with the last send message call and the radio backoff window had passed. We ran an initial test with 5 motes (4 hops) in a line setup as shown in Figure 8 but the test only included nodes 0 to 4. The route was pre-programmed so that the packet would hop through each node in a specific sequence so as to make sure each packet made 4 hops. We set we set the radio power to 7 in order to absolutely rule out poor connection as it gave a strong connection well outside the testing area. We treated the results of this test as a rough pass and only used them to guide our observation of where to explore next in terms of backoff window settings. In Table 1 below we see the results of our settings test for the backoff window.

Backoff Window Min	Backoff Window Max	Packet Drop Percentage	Throughput
2ms	125ms	5.6%	1.82 KB/s
7ms	80ms	6.6%	2.24 KB/s
1ms	16ms	20.6%	3.16 KB/s
1ms	32ms	12.8%	3.28 KB/s
1ms	64ms	9.6%	2.65 KB/s
16ms	32ms	2.8%	3.39 KB/s
16ms	64ms	2.6%	2.46 KB/s
32ms	64ms	3.8%	1.98 KB/s

Table 3: Results of exploratory pass on radio backoff window parameters with radio power setting of 7.

With our exploration of finding a good setting for the backoff timing window setting we started with a very wide window of [2ms, 125ms] and a throughput of 1.82 KB/s. This meant that the backoff window at any given time could be as low as 2ms or as high as 125ms. We used this initial wide backoff window as a baseline value to compare other results with. Our next parameter setting was a window of [7ms, 80ms], which attempted to narrow our range considerable while still keeping the minimum window relatively low at 7ms. At this setting the drop rate increased slightly to 6.6% which was slightly worse than the previous setting, but the throughput increased to 2.24 KB/s which was better. The next three ([1ms, 16ms], [1ms, 32ms], [1ms, 64ms]) runs were all based on a much narrower window and lower values. We saw that as the maximum window value

was raised from 16ms to 64ms, with a minimum window value of 1ms, that the packet drop rate declined from 20.6% to 9.6%. However, the throughput tended to decline from 3.16 KB/s to 2.65 KB/s. We then moved our minimum window backoff up to 16ms for the next 2 runs of ([16ms, 32ms], [16ms, 64ms]). Here the packet drop percentage became considerably lower and the throughput spiked to 3.39 KB/s at [16ms, 32ms], which was the best so far at that point. We made another set of runs at [32ms, 64ms] which showed to have a slightly higher drop rate but a much lower throughput. Since we were looking for a good configuration to test with, we decided that the backoff window [16ms, 32ms] gave us a very good packet drop rate of 2.8% while giving us the best throughput on the board at 3.39 KB/s. We saw that the backoff window needed to be a relatively low value, but not too far under 16ms. We also saw that after 32ms throughput tended to decrease as it created a relatively long waiting period between packets.

Once we had selected a backoff window, we had to decide what power setting we would use such that our motes would be able to maximize Termite's routing potential for the Imote2 platform. For our next test, we wanted to see how the Imote2 platform fared on a range of single line setups as described in Figure 8. We started with a simple 1 hop experiment as a baseline, and progressively added hops to see how Imote2 would respond. This test (and all further ones) used the radio backoff window derived from the previous section of [16ms, 32ms]. In this test every packet took a preset path through the nodes where each node had a hard coded next hop for ever previous hop possible. This ensured that regardless of how many other nodes the forwarding node could hear, it always forwarded to its next preset node on the path. Termite routing was not used in this test and no RREQs or RREPs were used during this test. All packets on the network were

data packets on a preset path which allowed us to see the effects of a packet moving N number of hops consistently. This allowed us to compare network load and other platform attributes based off of power settings and hop count.

In Figure 12 below we see the results of a single line setup (Figure 8) for a single hop with a preset path. The graph in Figure 12 compares packet drop percentage versus radio power for 2 nodes. The test was run 5 times for each power setting. We sent 500 packets per run and measured both throughput and latency. In Figure 12 we see that at a power of 1 the loss rate for packets is quite high at 22.24% which shows that a mote can deliver a packet 1 foot at that power, but not very reliably. At power 2 for the radio we see a dramatic drop in the number of lost packets at 3.04%. Beyond a radio power of 2 the dropped packet percentage levels off to 0.62% on average.

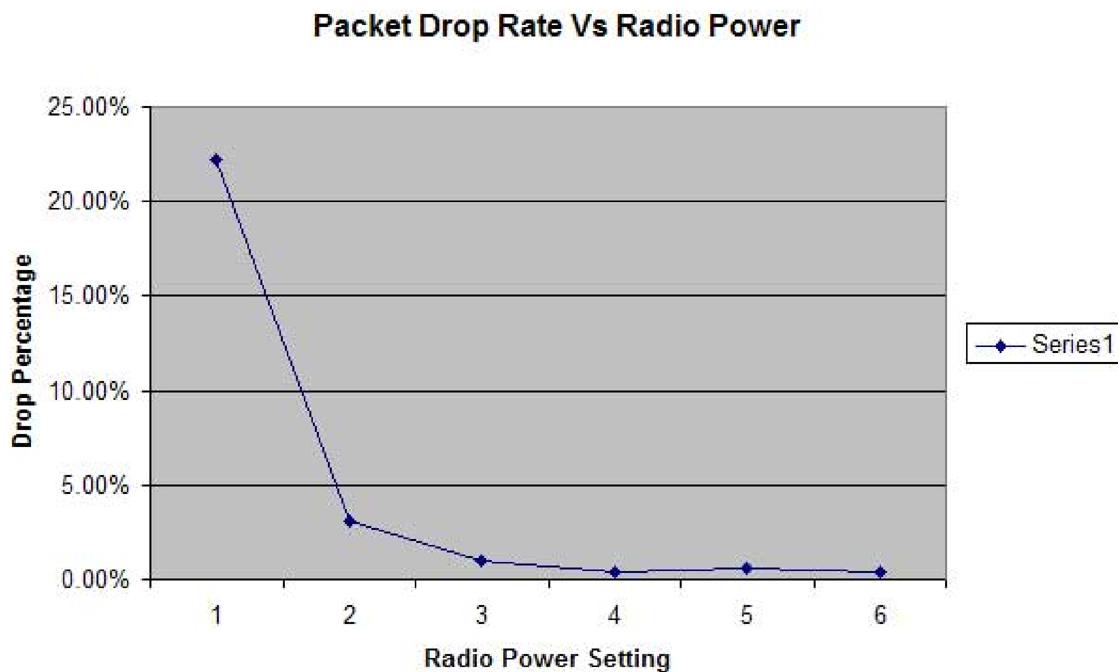

Figure 12: Packet drop percentage versus radio power, single line setup, 1 hop, Preset path with no routing.

We next examined how throughput correlated with the radio power setting as shown in Figure 13 below. At the minimum power setting of 1, the 1 hop setup yielded a throughput of 2.62 KB/s and 3.18 KB/s for a power level of 2. Again, performance of the system levels off beyond the power level of 2 at an average of 3.24 KB/s.

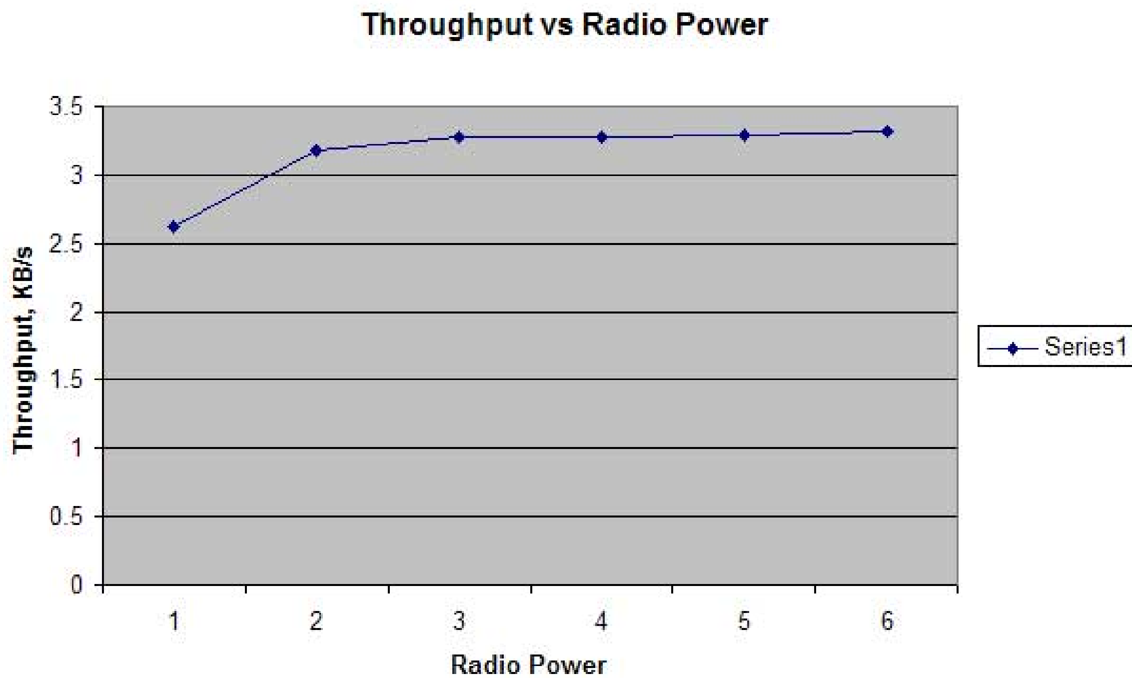

Figure 13: Throughput versus radio power, single line setup, 1 hop, preset path with no routing.

We then ran this same test for setups of 1, 2, 3, 4, and 5 hops in a single line setup as shown in Figure 8. In Figure 14 below we see the similar pattern of dropped packet percentage getting better as it approaches the radio power of 3 and then it levels off.

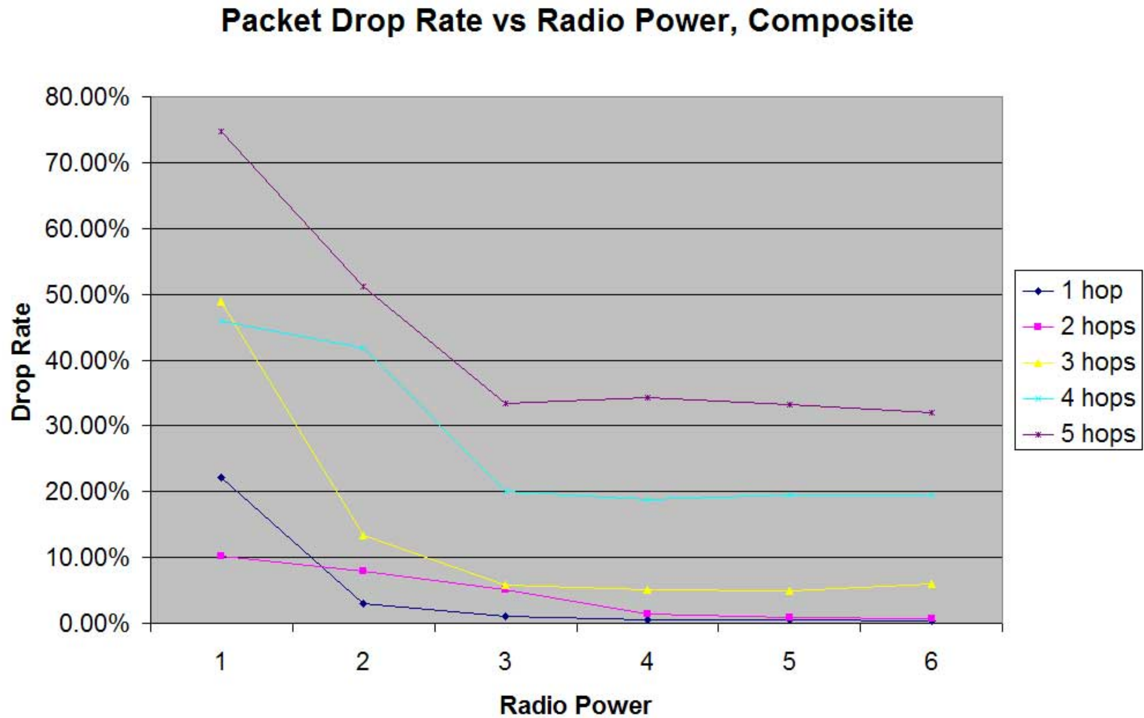

Figure 14: Packet drop rate versus radio power; Each test uses nodes in a line formation with a preset path and no routing.

This same correlation is true in Figure 15 (as shown below) as we see throughput rising as it approaches the radio power level of 3, and then levels off. However, in Figure 14 we can see that although the trend between each of the test setups look similar, as the number of hops increases for the test the minimum amount of dropped packets gets worse at every single power level. This effect can also be observed in Figure 15 where as the number of hops increases in a test, the throughput is worse at each power level compared to tests with a few number of hops.

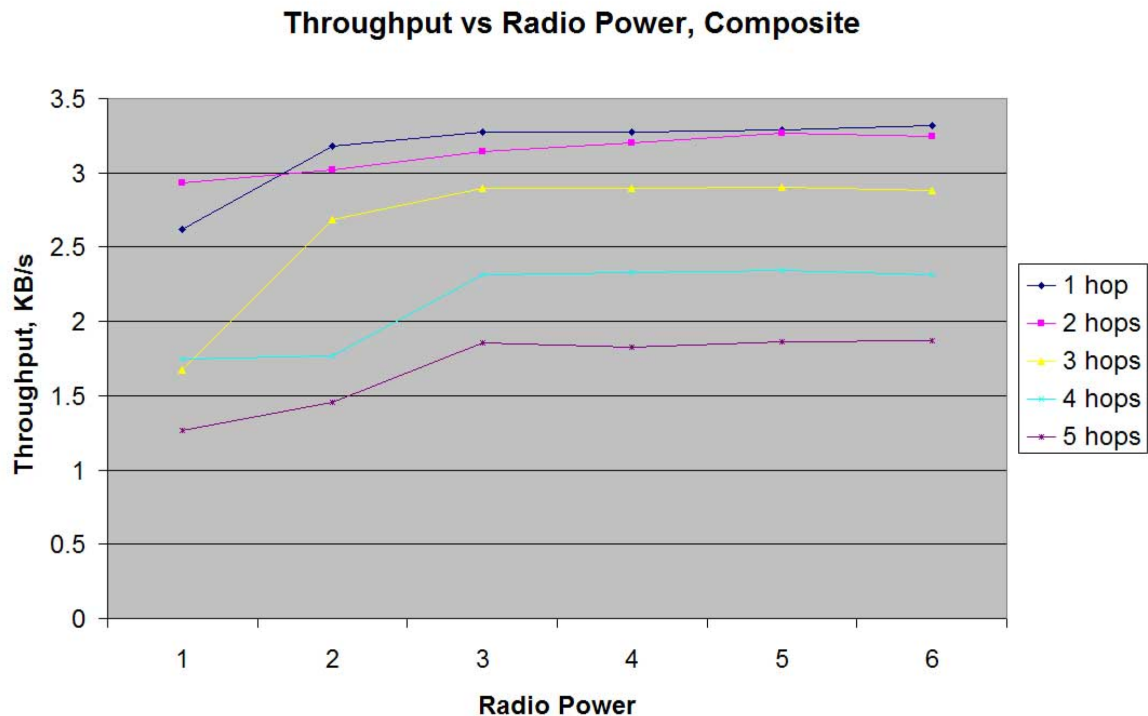

Figure 15: Throughput versus radio power; Each test uses nodes in a line formation with a preset path and no routing.

After talking to engineers involved with the Imote2 project, we found out that the TinyOS's radio stack only has a limited amount of space for incoming radio tasks at any given time. As the radio operates or hears traffic, it creates tasks for the operating system. If a task is generated that cannot fit onto the queue, then the task is lost. As the power level for the radio rises, it can transmit further and more nodes can hear it. As more of the nodes in the single line setup can hear one another, the amount of radio overhead they have to process increases dramatically. This follows with both figures as after the power level of 2, neither packet drop rate nor throughput gets any better for that number of nodes no matter how high the power gets raised. However, as the number of nodes

increase, the overall performance of that test gets worse. We concluded that as we generated successively more network traffic due to increasing power levels, the nodes began to drop packets as they simply could not process the amount of traffic coming in off the physical radio medium.

The next set of tests we ran were to judge the same setup as the previous test, but now we no longer preset the path. The nodes were in the same formation as the previous test (single line as shown in Figure 8), but each node actively used the Termite routing algorithm in order to get a packet to its intended destination. This meant that a node such as the source node could attempt to forward a packet to the destination directly if it detected the destination as a neighbor. All routing traffic overhead such as RREQs and RREPs were active in this test. The tests were run in a similar manner as the preset path test where for each power setting and number of hops we sent 5 runs of 500 packets each. This test showed the same basic trends as the first preset path test as shown in Figure 16 and Figure 17 below. However, the level off point at the radio power of 3 showed a significantly narrower range as the number of hops (where nodes = hops + 1) increases. A radio power setting of 1 (Figure 16 below) still yielded a similarly spread results, where less hops meant much better performance. However, as the radio power increased to 3 and beyond, the packet loss increased a very slight rate and the throughput decreased at a much slower rate. This led us to believe that at a power level of 1 Termite was using multiple hops to move a packet from source to destination but had trouble with poor connectivity at a power of 1 which led to high percentages of dropped packets.

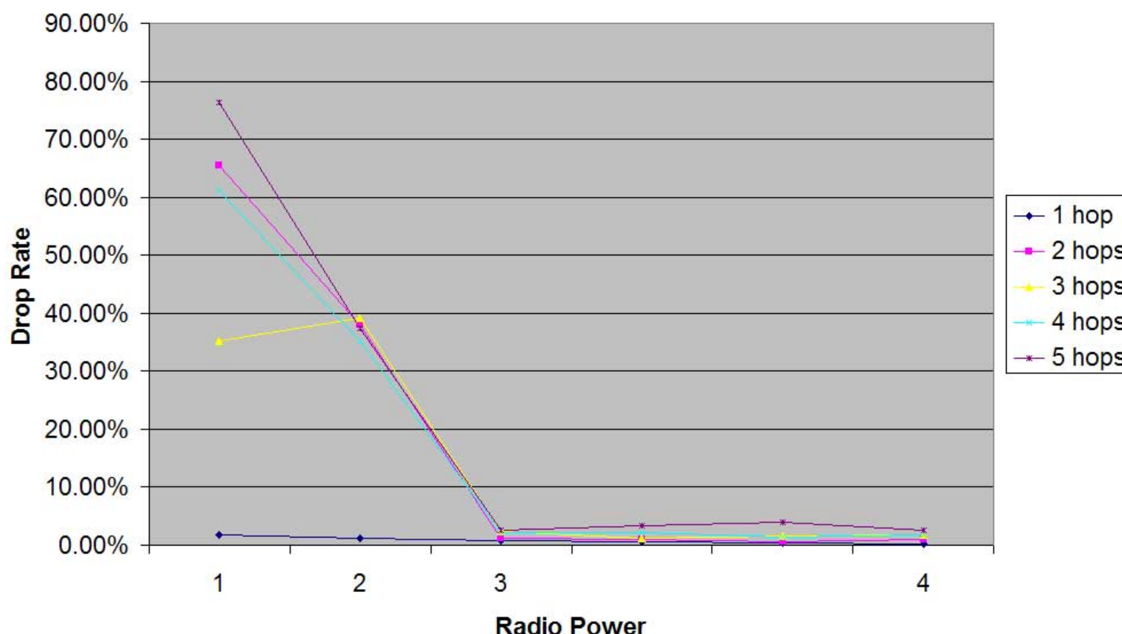

Figure 16: Packet drop rate versus radio power; Each test uses nodes in a line formation with active Termite routing.

At a radio power of 2, our packet drop percentage (Figure 16 above) was cut in half and our throughput (Figure 17 below) increased considerably which led us to believe that Termite was still using multiple paths. As the power increased beyond a power of 2 (Figure 16 above), the results were very similar which led us to believe that at that power all nodes could reliably hear one another as neighbors, and Termite was simply doing as it was instructed to do which was choose the destination if it is a neighbor. To confirm this we re-ran the test but this time saving the previous hop for each packet at the destination and discovered that this was true as most of the packets beyond a radio power of 2 were being sent directly from source to destination.

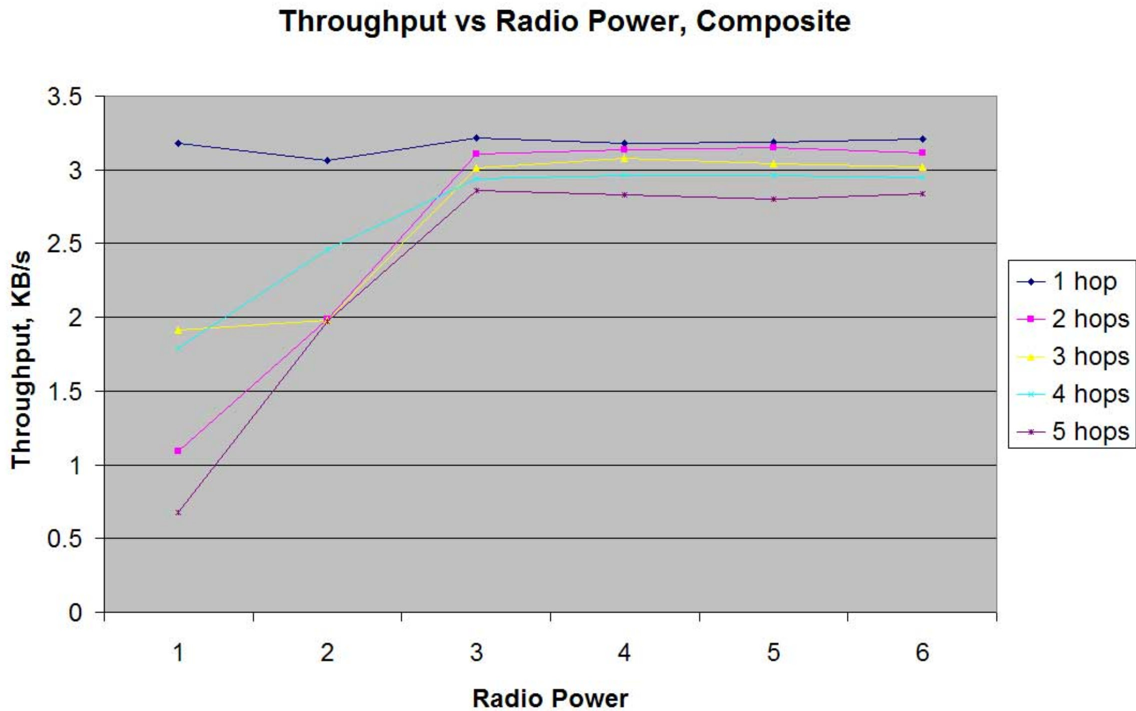

Figure 17: Throughput versus radio power; Each test uses nodes in a line formation with active Termite routing.

In order to run multipath experiments on Termite, we had to make some assumptions about our test environment. We knew we could not simulate every type of obstacle or interference pattern, so we decided to use the standard six-node graph shown in Figure 9. We also shifted each node to adjust radio interference patterns between run. Our radio backoff window of [16ms, 32ms] had proven reliable in Imote2 platform tests, but we still had to figure out how to reliably test mutipath routing where all the nodes did not hear one another as neighbors. However, we needed them to hear one another well enough such that the drop rate was not so high that it disrupted normal operations of Termite (dropped RREQs or RREPs are hard to detect). We decided that to

simulate normal MANET deployments we needed to take into account nodes that had bad reception and nodes that had acceptable reception, but not take into account the case of all nodes hearing one another. We believe the case of all nodes to hear one another to be non-existent since a MANET is designed for decentralized operation where only subsets of the nodes can hear one another. To approximate this we decided to run our multipath routing tests on the 6 node graph (Figure 9) at both a radio power setting of 1 and 2. Then we would take those results and average them together to get a more accurate picture with regards to performance of Termite. Figure 10 and Figure 11 above show how the radio powers overlap with regard to their setting. In Table 4 below, we see a comparison of throughput between two nodes (their raw throughput), a single line network (Figure 8) with a preset path, and multipath routing on a six-node graph (Figure 9) with TinyTermite.

Raw Throughput	2.9 KB/s
Throughput of line network with preset path	2 KB/s
TinyTermite Throughput	1.84 KB/s

Table 4: Comparison of throughput between two nodes (raw throughput), a line network with a preset path, and then multipath routing on a 6 node graph (Figure 4) with TinyTermite.

To calculate throughput on the six-node graph (Figure 9) we made 10 runs of the algorithm each with 500 packets being sent across the network. This was done as described above for radio power settings of 1 and 2, and then the measurements were

averaged together. The throughput we measured for this implementation of TinyTermite was 1.844 Kbytes per second. In Table 4, we see raw throughput at 2.9 KB/s which is the amount of data we could move over one hop with no routing. (In Table 3 we see a throughput value of 3.39 KB/s but that was at a radio power of 7. The raw throughput of 2.9 KB/s (single hop) is based on the average of the throughputs of the six power settings in Figure 13.) We also see the throughput of a line network with a preset path as 2 KB/s compared to the throughput we calculated for TinyTermite at 1.84 KB/s. This shows that TinyTermite does not reduce the throughput significantly.

Next we measured the packet drop rate for TinyTermite. TinyTermite's packet drop rate was found to be 33%. The throughput test put the system under a high level of stress and moved as much data as possible for a period of time. We also calculated a baseline packet loss where we sent 2 packets a second across the six-node graph (Figure 9) for powers of 1 and 2 and averaged the drop rates together. The baseline packet loss for this graph was found to be 12.7%. Latency was calculated by sending a packet across the graph from the source node to the destination node, and then back to the source node. We ran the test 10 times and calculated the average. This round trip time, or latency, for the TinyTermite algorithm was 47.813ms on the same 6 node graph setup (Figure 9).

We simulated the attacker as shown in Figure 6 (c) where a neighbor node was replaying a valid packet that appeared to come from the destination. The rogue node was also actively performing a selective forwarding attack where it forwarded no node other than its own replay packets meant to attack the source node. This strongly influenced the source node's routing table to route packets through the attacker for the intended destination. In the first run of this experiment, no suspicion defense was employed and

the rogue was free to disrupt the network as it could. On average 11.5 packets were received by the destination node and the rogue node captured 70.2 packets. This works out to an 88.5% effective loss rate for packets when the network is under siege by the single rogue node.

In the second experiment, the rogue was setup the same as in the first experiment, but this time we engaged the suspicion defense mechanism on the source node as illustrated in Figure 6 (d) and (e). On average 67.1 packets were received and the rogue node only captured 12.9 packets. The effective loss rate dropped to 32.9%. For both of these experiments we must also consider that our drop rate is 12.7% on average for a 6 node graph (Figure 9) of the same topology from source to destination. The summary of the results for the comparison of TinyTermite with Termite with regards to security are listen below in Table 5.

Method	Number Packets Received by Destination Node	Number Packets Captured by Attacker Node	Packet Loss Rate
Termite	11.5	70.2	88.5%
TinyTermite	67.1	12.9	32.9%

Table 5: Comparison of TinyTermite with Termite with regards to security. Experiments performed on network in Figure 9 where baseline packet loss for network is assumed to be 12.7%.

The suspicion defense was able to both increase the number of packets delivered and decrease the number of packets captured while the network was under attack. With

suspicion defense active we delivered 583% more packets, from 11.5 packets on average up to 67.1 packets on average per 100 packets sent from source. We also were able to significantly reduce the number of packets captured with the sinkhole attack. Initially the rogue node captured 70.2 packets per 100 with no suspicion defense on and we improved that to a capture rate of 12.9 packets per 100 with suspicion defense active. This works out to a decrease of 81.6% less packets in terms of number of packets the rogue was able to capture.

CHAPTER 5

Conclusions

The goal of this thesis was to introduce a secure routing algorithm for MANETs. For this purpose, we proposed TinyTermite a novel probabilistic routing algorithm that is secure, distributed, and suitable for dynamic networks. TinyTermite is implemented on Imote2 and its performance is measured in a real world setting.

5.1 TinyTermite Implementation

First, we investigated the properties of the Imote2 platform by measuring its one hop performance, radio subsystem backoff windows, theoretical throughput across many hops, and how radio power affects performance. Then, we implemented Termite and TinyTermite on the Imote2 platform and measured their performance. We compared these two routing algorithms with respect to throughput, latency, and packet loss, and found TinyTermite to be an effective routing algorithm for the platform.

5.2 Suspicious Pheromone

We studied the effects of attacks on MANETs and came up with a defense mechanism called “Suspicious Pheromone”. We examined how the replay attack and selective forwarding can disrupt routing in MANETs with the TinyTermite implementation. We then measured how well the suspicious pheromone defense successfully routed packets around the attacker and towards the intended destination in our implementation. We found that this defense mechanism was highly effective against

these attacks coupled together and successfully route significantly more packets to the intended destination than without the defense mechanism.

5.3 Suggestions for Future Work

This thesis examined the properties of the Imote2 platform and Termite as implemented in TinyTermite. In the following, several opportunities are listed for future investigation.

- Problems related to implementation
  - Finer tuning of the backoff window
- Problems related to Suspicion
  - More types of attacks tested against
  - Different types of buildup and decay models
  - Other uses for suspicion in normal routing
- Problems related to Disjoint Path
  - Implementation and testing of disjoint path in TinyTermite

APPENDIX A

WIRING FILE FOR MAIN TINYTERMITE MODULE

```
configuration TinyTermite {

} implementation {

    components
        Main, TinyTermiteM, BluSHC, RadioCRCPacket as Comm, CC2420ControlM,
        TimerC, LogicalTime, LedsC, RandomLFSR
        ;

    // Time Interfaces
    TinyTermiteM.Time -> LogicalTime.Time;
    TinyTermiteM.TimeUtil -> LogicalTime.TimeUtil;

    // Timer Interfaces
    TinyTermiteM.TimerControl -> TimerC;
    TinyTermiteM.Timer -> TimerC.Timer[unique("Timer")];
    TinyTermiteM.TimerData -> TimerC.Timer[unique("Timer")];
    TinyTermiteM.TimerTxBackoff -> TimerC.Timer[unique("Timer")];

    // LED Control Interface
    TinyTermiteM.Leds -> LedsC;

    //  Standard Control Interfaces
    Main.StdControl -> TinyTermiteM.StdControl;
    Main.StdControl -> LogicalTime;

    // CC2420 Radio Chip Control Interface
    TinyTermiteM.CC2420Control -> CC2420ControlM;

    // Messaging Subsystem Interfaces
    TinyTermiteM.RadioControl -> Comm;
    TinyTermiteM.RadioSend -> Comm;
    TinyTermiteM.RadioReceive -> Comm;

    // Random Number Generator Interface
    TinyTermiteM.Random -> RandomLFSR.Random;

    // BluSH Shell Command Interfaces
    BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.Hello;
    BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.Debug;
    BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.Chirp;
    BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.NodeID;
    BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.Suspects;
    //BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.CogMapDump;
    BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.MessagesRecieved;
    BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.MessagesSent;
```

```
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.SendHello;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.SendRREQ;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.SendData;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.CheckPTable;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.ListNeighbors;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.DebugHello;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.DebugRREQ;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.DebugRREP;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.DebugData;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.MessageStats;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.PrvMsg;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.ClearPTable;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.SetupTest;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.RunTest;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.AddDebugDest;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.AddDebugNeighbor;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.RunRREQBounceTest;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.SeenData;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.SeenRREQ;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.LastRREQ;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.SetupStartNode;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.SetupMidHop;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.SetupEndNode;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest1;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest1_a;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest2;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest3;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest3_a;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest3_b;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest4;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest4a;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest5;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest5a;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest6;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest7;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest8;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest9;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest10;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest10a;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.EqTest10b;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.TestSeenPackets;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.TestSeenPackets2;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.TestFwdQueue;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.TestFwdQueue2;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.Timestamp;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.ExpLookupIndex;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.RefreshDemoLink;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.SetupTable;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.RouteSelData;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.RouteSelRREQ;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.RouteSelRREP;
```

```
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.TurnPDecayOn;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.TurnPDecayOff;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.SimPktIn;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.TestForward;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.LastRREP;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.CycleDemoDst;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.DebugFwdQ;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.DPkt;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.c;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.m;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.IncPow;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.DecPow;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.TurnDataTestOn;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.TurnDataTestOff;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.SimRepeats;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.i;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.StartLatTest;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.StopLatTest;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.ViewLatStats;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.StartThroughputTest;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.ResetThroughputTest;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.StopThroughputTest;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.ViewThroughputStats;

BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.s; // StartThroughputTest
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.r; // ResetThroughputTest
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.v; // ViewThroughputTest
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.t;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.q; // run suspicion bursts
//BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.l; // latency start
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.p;  // start packet drop test

BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.StartRangeTest;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.StopRangeTest;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.StartBounceTest;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.StopBounceTest;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.ParamEx;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.ClearRouteQueue;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.DebugRouteQueue;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.DebugRouteHeap;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.ScanPassive;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.StartDropTest;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.StopDropTest;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.StartRogue;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.StopRogue;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.RogueStats;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.IncMaxBackoffDelay;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.DecMaxBackoffDelay;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.IncNumPkts;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.DecNumPkts;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.ContactOn;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.ContactOff;
BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.SuspicionOn;
```

```
        BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.SuspicionOff;

        BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.SinkOn;
        BluSHC.BluSH_AppI[unique("BluSH")] -> TinyTermiteM.SinkOff;

  }

// end of module file
```

REFERENCES

1. M. Burk, “Smartphones: State of the Market”, *http://mobilemonday.org.uk/MMetrics%20MoMo%20Feb%204.pdf, 2008*

2. I. Chakeres, E. M. Belding-Royer, “Utilizing Resource-Rich Nodes in Ad Hoc Networks,” *ACM SIGMOBILE Mobile Computing and Communications Review, Volume 7, Issue 3* (July 2003)

3. M Roth, S Wicker, “Termite: Ad-Hoc Networking With Stigmergy,” *Global Telecommunications Conference, 2003. GLOBECOM ’03. IEEE*, Vol. 5 (2003), pp. 2937 – 2941 vol.5.

4. E. Bonabeau, M. Dorigo, G. Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems*, Oxford University Press, 1999.

5. S. Camazine, J.L. Deneubourg, N. Franks, J. Sneyd, G. Theraluz, E. Bonaneau. *Self Organization in Biological Systems*, Princeton University Press, 2001

6. C. Karlof, D. Wagner, *Secure routing in wireless sensor networks: attacks and countermeasures.* Elsevier’s AdHoc Networks, 1:293-315, September 2003.

7. Crossbox, “Imote2 platform”, http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/Imote2_Datasheet.pdf

8. Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin, Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler, “TinyOS: An Operating System for Sensor Networks,” *Ambient Intelligence*, Springer Berlin Heidelberg, 2005.

9. D. Gay, P. Levis, R. von Behren, M.Welsh, E. Brewer, and D. Culler. “The nesC language: A holistic approach to networked embedded systems,” *In Proceedings of Programming Language Design and Implementation (PLDI)*, June 2003.

10. J. Jubin and J. D. Tornow, “The Darpa packet network radio protocols,” *In Proceedings of IEEE* vol. 75, 1, pages 21 – 32, Jan. 1987

11. B. M. Leiner, D. L. Nielson, and F. A. Tobagi, “Issues in packet radio network design,” *Proceedings of the IEEE Special Issue on “Packet Radio Networks”*, 71, 1:6-20, 1987

12. S. Corson, J. Macker, and S. Batsell. “Architectural Considerations for Mobile Mesh Networking,” *http://tonnant.itd.nrl.navy.mil/mmnet/mmnetRFC.txt*, 1996

13. J. Macker and S. C. (chairs). “Mobile Ad-hoc Networks (MANET),” *http://www.ietf.org/html.charters/manet-charter.html*, 1997.

14. David F. Bantz and Frederic J. Bauchot. “Wireless LAN design alternatives,” *IEEE Network*, 8(2):43-53, March/April 1994.

15. George S. Lauer. “Packet-radio routing,” *In Routing in Communications Networks*, chapter 11, pages 55-76. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

16. John Jubin and Janet D. Tornow. “The DARPA packet radio network protocols,” *Proceedings of the IEEE*, 75(1):21-32, January 1987.

17. C. Hedrick, “Routing Information Protocol,” *Internet Request for Comments RFC 1058*, June 1988.

18. YihChun Hu, Adrian Perrig, David B. Johnson, “Ariadne: A Secure OnDemand Routing Protocol for Ad Hoc Networks,” *Proceedings of the Eighth Annual International Conference on Mobile Computing and Networking (MobiCom 2002),* pp. 12-23, ACM, Atlanta, GA, September 2002.

19. J. M. McQuillan, David C. Walden. “The ARPA network design decisions,” *Computer Networks*, 1(5):243-289, August 1977.

20. Gursharan S. Sidhu, Richard F. Andrews, and Alan B. Oppenheimer. *Inside Apple Talk*. Addison Wesley, Reading, Massachusetts, 1990.

21. Paul Turner. “NetWare Communications Processes,” *Netware Application Notes*, Novell Research, pages 25-81, September 1990.

22. Xerox Corporation. “Internet transport protocols,” *Xerox System Integration Standard 028112*, December 1981.

23. International Standards Organization. “Intermediate system to intermediate system intra-domain routing exchange protocol for use in conjunction with protocol for providing the connectionless-mode network service (ISO 8473),” *ISO DP 10589*, February 1990.

24. John M. McQuillan, Ira Richer, and Eric C. Rosen. “The new routing algorithm for the ARPANET,” *IEEE Transactions on Communications*, COM-28(5):711-719, May 1980.

25. J. Moy. “OSPF version 2,” *Internet Request For Comments RFC 1247*, July 1991.

26. C. Perkins, E. Belding-Royer, S. Das, “Ad-hoc On-demand Distance Vector

(AODV) Routing," *http://moment.cs.ucsb.edu/AODV/aodv.html*, 2004.

27. Charles E. Perkins, Pravin Bahgwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers," *ACM SIGCOMM Computer Communication Review*, Volume 24, Issue 4 (October 1994), Pages: 234 - 244

28. Nachum Shacham and Jil Westcott. "Future directions in packet radio architectures and protocols," *Proceedings of the IEEE*, 75(1):83-99, January 1987.

29. R. Bellman, "On a Routing Problem," *Quarterly of Applied Mathematics*, vol. 16, pp. 87-90, 1958.

30. T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduction to Algorithms*. The MIT Press, second edition. 2002.

31. D. Bertsekas and R. Gallager. Data Networks, pages 297-333. Prentice Hall, Inc., 1987.

32. C. Cheng, R. Riley, S. P. R. Kumar, and J. J. Garcia-Luna-Aceves. "A loop-free Bellman-Ford routing protocol without bouncing effect", *In ACM SIGCOMM '89*, pages 224-237, September 1989.

33. D. Johnson, D. Maltz, "Dynamic Source Routing in Ad-hoc Wireless Networks," *SIGCOMM '96*, 1996.

34. Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta G. Jetcheva. "A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols." *In Proceedings of the Fourth ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'98)*, pages 85–97, October 1998.

35. Per Johansson, Tony Larsson, Nicklas Hedman, Bartosz Mielczarek, and Mikael Degermark. "Scenario-based Performance Analysis of Routing Protocols for Mobile Ad-hoc Networks", *In Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99)*, pages 195–206, August 1999.

36. David A. Maltz, Josh Broch, Jorjeta Jetcheva, and David B. Johnson. "The Effects of On-Demand Behavior in Routing Protocols for Multi-Hop Wireless Ad Hoc Networks," *IEEE Journal on Selected Areas in Communications*, 17(8):1439–1453, August 1999.

37. Roy C. Dixon and Daniel A. Pitt. "Addressing, bridging, and source routing," *IEEE Network*, 2(1):25-32, January 1988.

38. Deborah Estrin, Daniel Zappala, Tony Li, Yakov Rekhter, and Kannan Varadhan. “Source Demand Routing: Packet format and forward specification (version 1),” *Internet Draft*, January 1995. Work in progress.

39. Philip R. Karn, Harold E. Price, and Robert J. Diersing. “Packet radio in amateur service,” *IEEE Journal on Selected Areas in Communications*, SAC-3(3):431-439, May 1985.

40. M. Schwartz and T.E. Stern. “Routing techniques used in computer communications networks,” *IEEE Transactions on Communications*, COM-28(4):539-552, April 1980.

41. Network Working Group, “Optimized Link State Routing”, *http://www.ietf.org/rfc/rfc3626.txt,* 2003.

42. L. Ford Jr., D. Fulkerson, “Maximal Flow Through a Network,” *Canadian Journal of Mathematics*, vol. 8, pp. 399-404, 1956.

43. E. Dijkstra, “A note on two problems in connection with graphs,” *Numerische Mathematik*, Vol. 1, 269-271, 1959.

44. C. Perkins, E. Belding-Royer, S. Das, “Ad-hoc On-demand Distance Vector (AODV) Routing,” *http://www.faqs.org/rfcs/rfc3561.html*, 2003.

45. J. L. Deneubourg, J. C. Gregoire, and E. Le Fort, *Kinetics of the larval gregarious behaviour in the bark beetle Dendroctonous micans*. Journal of Insect Behavior 3:169-182. 1990a.

46. C. Karlof, N. Sastry, D. Wagner, *TinySec: A link layer security architecture for wireless sensor networks.* Proc. SensSys, page 162-175, November 2004.

47. R. G. Cowell, A. P. Dawid, S. L. Lauritzen, D. J. Spiegelhalter, *Probabilistic Networks and Expert Systems*. Springer-Verlag, New York, 1999.

48. M. M. Gupta, L. Jin, N. Homma, *Static and Dynamic Neural Networks: From Fundamentals to Advanced Theory.* Wiley-IEEE Press, April 2003.