7-2006

Software based deployment of encryption keys in wireless sensor networks.

Timothy William Hnat
University of Louisville

SOFTWARE BASED DEPLOYMENT OF ENCRYPTION KEYS IN WIRELESS SENSOR NETWORKS

By

Timothy William Hnat
M.Eng., University of Louisville, 2006

A Thesis
Submitted to the Faculty of the
University of Louisville
Speed School of Engineering
as Partial Fulfillment of the Requirements
for the Professional Degree of

MASTER OF ENGINEERING

Department of Computer Engineering and Computer Science

July 2006

SOFTWARE BASED DEPLOYMENT OF ENCRYPTION KEYS IN WIRELESS SENSOR NETWORKS

Submitted by

Timothy William Hnat

A Thesis Approved on

(Date)

by the Following Reading and Examination Committee:

Rammohan K. Ragade, Thesis Advisor

Adel S. Elmaghraby

John F. Naber

ABSTRACT

Sensor networks are just in their infancy. Their use will continue to grow as the technology becomes cheaper and more efficient. A current shortcoming with sensor networks is the inability to efficiently provide secure communications. As sensor networks are deployed to monitor and control systems, the security of communications will become a more important.

This thesis proposes a new approach to key establishment and renewal through the use of point-to-point keys and software verification and validation to ensure the integrity of two nodes. Sensor networks exist on limited resources, so power efficiency is a concern. The proposed protocol allows for the use of small keys instead of large pre-distributed keys.

This thesis explores the design and implementation of a new point-to-point key generation and renewal algorithm. The main contribution is the development of an algorithm that utilizes a software integrity check to ensure the validity of a node. The thesis also utilizes a simulated sensor network to test and validate the new software algorithm.

ACKNOWLEDGMENTS

I would like to express my appreciation to my advisor, Dr. Rammohan Ragade, for his support and guidance during my pursuit of this research and my committee, Dr. Adel Elmaghraby and Dr. John Naber. I would like to thank Jeff Hieb and Brian Carter who have helped me through the research process and have provided insight into the technical side of sensors and security. I would also like to thank my parents, William and Janice Hnat, for supporting me through these past 5 years of college.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I
INTRODUCTION

1.1. Background

Wireless sensor networks are becoming popular as a net method to gather information. The miniaturization of digital electronics, with regards to chip and sensor design, are allowing wireless sensor grids to become an economically practical solution.

Early work in sensor technologies used large wired sensors to gather data on large remote devices to monitor from afar. However, the technology today allows sensors to be battery powered and manufacturing costs are low enough to place multiple sensors within a given area. A device is created that can monitor and report information to a central location by reducing the cost per sensor and making their communication wireless. “A sensor grid is comprised of many individual and independent sensors which can be deployed inside or near a phenomenon” [1].

Wireless grids can be utilized for a variety of scenarios. Many sensors monitor temperature, humidity, lighting conditions, pressure, noise levels, chemical makeup of the area, etc... The sensors can be utilized for military applications from personnel detection to battlefield surveillance and chemical attack detection [43, 11]. Other uses for sensor technology include monitoring human health and the environment [43, 11, 28, 29]. For the consumer, home automation and smart environments require the abilities of the sensor grid to gather and distribute information. Commercial impacts involve better control of environmental conditions in office buildings [1].

The new wireless sensor grids present a whole new set of problems including, but not limited to: device integrity, communication security, and operating lifetime (battery power). As the devices become smaller, more power efficient, and less expensive, limitations are placed on the ability to secure each device and to maintain the security of network communications. A problem possible in hostile situations can be the enemy, with more sensors, has a larger opportunity to capture and/or break into the communication network of sensors [43, 15, 20].

A lack of organization is inherent to the nature of wireless grids. Each sensor does not know where it will end up and with whom it can safely communicate. It is not aware of the location of other sensors within the grid. The sensor knows of nearby sensors but not their locations in the environment. This lack of organization presents problems in resolving communication channels back to a master controller, as well as problems with trust among other devices in the grid.

A critical component that must be addressed for large scale deployment of wireless sensor grids is the cost of the system. At present, each wireless sensor node costs upward of 250 dollars, a price which is cost prohibitive for wide-scale deployment [10]. The cost needs to be under one dollar per sensor for the economics of wireless grids to make fiscal sense [1]. At that price, sensors are expendable, something which is necessary in battlefield scenarios.

The power consumption of these wireless sensors is still too large to be of use in long-term environments and scenarios. The advantage of wireless sensor grids is limited due to the inability to provide long-term monitoring capabilities. Power consumption needs to be reduced and power sources must be improved. This could involve alternative power from solar and other environmental sources to support batteries. The long term capabilities need to rely on a power source that is available for extended periods of time. Without a long term wireless sensor, the network is neither fiscally nor logistically feasible to maintain [1, 35].

These sensors are usually built upon a framework package known as TinyOS [1]. This framework provides the ability to quickly program and control sensor packages without the need to custom program each device during assembly. These sensors are built upon an Atmel micro-controller and run the AVR instruction set. TinyOS is capable of running on other types of sensors. TinyOS wraps this in a framework to provide a quick and easy interface for the sensors to work with.

Communication between sensors in the grid environment is cause for concern in security oriented deployments [1]. Basic communication protocols broadcast the transmitted information in a ASCII stream. The thrust of research up to this point has been on getting sensor networks functional and scaleable, not in the security of their information. This problem of scalability has been addressed in a variety of methods. These methods are based in software and network routing schemes.

Basic communication protocols and transmission methods utilize an omnidirectional antenna to both send and receive signals, though this does not provide any type of security in the transmission. However, directional antennas can limit the signal broadcast to the direction of the sensor in which communication is going to take place. The complexity of a directional approach makes this technique infeasible for most scenarios.

Wireless sensor grids are currently receiving a lot of research attention; however, the majority of research examines the communication and handoff protocols or the implementation side of networks. This thesis will describe the design and implementation of a software authentication procedure for establishing unique security keys between sensor nodes.

1.2. Organization of Thesis

Chapter II presents a detailed literature review of wireless sensor grid concepts, techniques, and concerns. Chapter III discusses the design of the software based key determination approach. Chapter IV describes the implementation of the software-based key determination algorithm. Chapter V contains the testing and results from both simulation of the wireless grid system and a real-world scenario. Chapter VI contains conclusions and future directions.

CHAPTER II
LITERATURE REVIEW

Sensor grids are deployed for a wide variety of uses. A potential application area of concern is hostility to the grid infrastructure [32, 43, 15]. Sensor grids will be subjected to various security attacks within the confines of their deployment [32]. Three issues must be addressed with the deployments of sensors: (1) attacks on communications and data integrity from the sensors, (2) the physical vulnerability associated with deployed sensors, and (3) a method to provide a “self-checking” aspect within the sensor network and on the sensors themselves [32].

In order to verify that the program residing within a sensor has not been modified (integrity), some sort of authentication mechanism must be utilized [32]. One method is the use of a Program-Integrity Verification (PIV) server. A controller on a wireless grid is able to maintain a list of all authenticated nodes [32]. This process involves checking and authenticating the nodes as they join the network or after experiencing a long service blockage. This verification process does contain overhead. However, this overhead can be made relatively small as compared to traditional cryptographic functions. The verification process can utilize a cryptographic hash function that is “orders-of-magnitude cheaper and faster than other approaches” [32]. The hash functions can be optimized for the embedded controllers at both the 8- and 16-bit sizes.

The system architecture plays a major role in how resilient to attack the network and individual nodes can be made. Nodes can be made relatively secure through a variety of well-defined hardware design methods. There are two general classes of hardware security mechanisms: read-proof and tamper-proof. Read-proof hardware

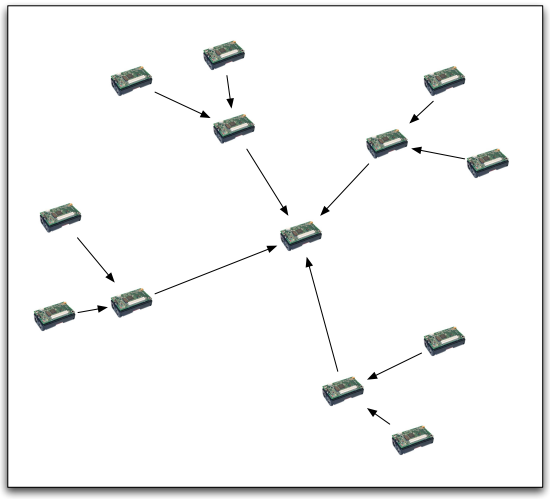

FIGURE 2.1 – Wireless Network Example

are the group of technologies that prevent the enemy from reading data contained within the chip. Tamper-proof hardware are the group of technologies that prevent the enemy from changing data on the chip [18]. This technology, so far, is relatively expensive to produce and cost prohibitive for deployment in sensor networks [33]. How the wireless system is setup in regards to the architecture will affect its security and resiliency.

Figure 2.1 demonstrates an example of a wireless sensor network. The solid arrows denote secure wireless traffic. A breakdown of a master key encrypted network, as shown in figure 2.2, has all of its communications insecure. Once a master key is discovered, the entire network is lost. The utilization of a group-based key system prevents the entire network from being lost (Figure 2.3). The best method is to use a point-to-point key system so if a link or node is compromised, only the current node is lost and not the rest of the network (Figure 2.4).

Attacks on a wireless grid come from many sources. "Node compromise is the central problem that uniquely characterizes the sensor network's threat model" [35, 21]. The first is a hostile attack on the transmitted signal. This involves capturing some of the wireless signal data and processing it in an attempt to break into

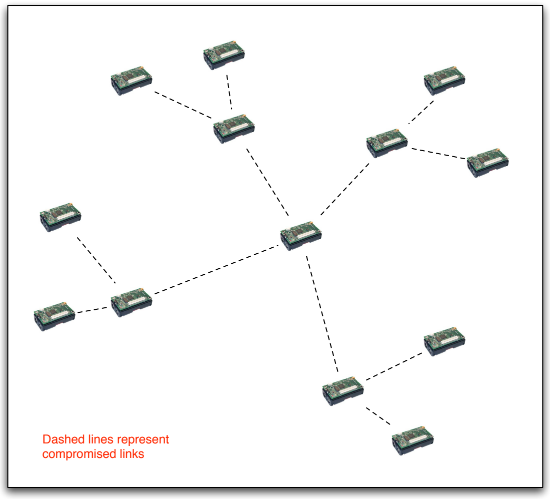

FIGURE 2.2 – Compromised Master Key Example

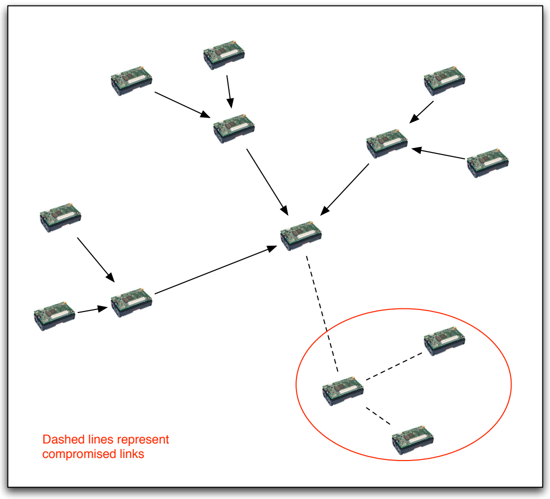

FIGURE 2.3 – Compromised Group Key Example

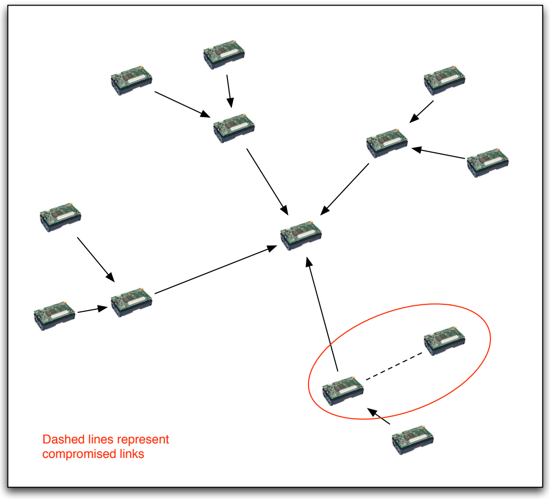

FIGURE 2.4 – Compromised Point-to-Point Key Example

the current encryption of the wireless network. "Sensor nodes use wireless communication, which is particularly easy to eavesdrop on" [35, 9, 11]. This is impossible to detect from within the network. Passive monitoring will not cause any interference to the network.

The second attack vector is a physical attack on the node. This involves obtaining a node from the network or a replica of one and taking it into a lab to use a more powerful computer to break into the system. In the worst case, a scanning electron microscope could be utilized to dissect the memory from the device and rebuild the system on a new device, though this would be a very expensive process and most likely take too much time to be of much use.

A third option for attack would be to cause service disruptions to the wireless grid. This could be as easy as jamming the wireless signal or doing any thing else that would disrupt the ability of the network to do its task. Nodes could be destroyed in an attempt to decrease the range of the network. Once the network has been compromised, a data attack can be made on the system. Compromised nodes could be made to send invalid data throughout the network in an effort to disrupt the ability of the sensing grid [31, 16, 35, 33].

The last attack vector is a more traditional method from the world of network computing. The wireless sensors are battery-powered devices and because of this, severely constrained in resources and lifetime. A resource consumption attack would attempt to get the individual nodes to utilize their sensors, processor, or transmitter in an effort to consume available battery resources [3]. This would quickly end the life of the sensor and eliminate it from the system. This also parallels denial-of-service attacks as seen on traditional networks [32, 35, 33, 11].

The ultimate problem with any wireless sensor network is the ability of a foreign body to capture and reproduce nodes in a more sophisticated device with more memory in such a way that the new device functions the same way from network's perspective as the original node but contains malicious code [32, 11]. In traditional computing and security, physical security is one of the primary foci and communication restrictions another. A generally accepted practice is once an intruder gains physical access to a machine, anything on that machine is no longer secure and safe.

Security within wireless sensor grids needs to be able to scale efficiently to large-scale deployments [35, 2, 6, 20]. The mechanisms and protocols need to have either bandwidth efficiency or be self-contained within a cluster of sensor nodes. This will allow for large-scale deployments to occur.

Sensor networks are expected to grow to many thousands of nodes in the future. This large scale deployment presents a scalability problem that is similar to problems occurring in large scale super-computing. The main difference is the computational and bandwidth resources available to the sensor networks. Security is usually not of too much concern within the super-computing environment other than at the perimeter of the system. The security model for sensor networks needs to be ultra scaleable for any viable deployment [35, 33, 2, 9, 6].

2.1. Cryptographic Keys

The security requirements for a key distribution protocol involve the availability of sensor nodes, authenticating sensor nodes, integrity of communication messages, confidentiality of communication, non-reputation of nodes, survivability of sensor nodes, and degradation of security services [4, 19]. All of these are critical to ensure that total security of the system is kept intact. Key distribution mechanisms involve the ability to scale the key scheme to large networks, the efficiency at which this distribution occurs is measured in storage, processing and communication complexity, key connectivity and resilience to capture [4, 15, 19, 28]. The key distribution schemes involve one of two types of techniques: pre-distribution or post-distribution schemes.

The management systems need to provide and ensure key independence (if the protocol allows this), perfect forward secrecy, and backward/forward secrecy [2]. The key independence guarantees that an adversary, knowing a subset of the keys, cannot discover future or prior keys. "Forward secrecy guarantees that a passive adversary who knows a subset of old group keys cannot discover subsequent group keys, while backward secrecy guarantees that a passive adversary who knows a subset of group keys cannot discover preceding group keys" [2].

Key distribution within wireless sensor networks is critical to the ability of the system to keep communications secure. Many different approaches exist in this field. The pair-wise key pre-distribution scheme involves computing distinct pair-wise keys to distribute to the nodes [13]. The base case is typically only to utilize a single master key and distribute this to all nodes. This is a significant problem when one of the nodes is captured and the entire network is compromised. The best case is to generate n-1 keys for the network of size n in order to keep communication between all nodes open. This poses problems with the scalability of the network [4, 8, 27, 40].

The random pair-wise key scheme addresses the storage component of the basic scheme at the expense of the inter-node key connectivity [12, 4, 13]. This scheme loads a small subset of the total keys into each node in an attempt to provide a high probability of matching a fellow node's key. This scheme has the same benefits and weaknesses as the basic key distribution scheme mentioned with regards to node compromise and scalability. The only benefit is with the storage of keys within each node [8, 34].

The closest pair-wise key pre-distribution scheme provides a more efficient system as compared to the random or basic schemes; however, this scheme relies on the ability to know approximately where each sensor is going to be placed [4, 25, 27]. This is not a good technique for networks in which sensors are not going to be placed in specific areas [14]. The ID-based one-way function scheme provides a mechanism in which two nodes can share a key in, at most, two hops [4]. Multiple ID based one-way function schemes offer a more efficient approach to the single one-way function scheme in terms of memory usage. The disadvantage of these schemes is that with a compromised node, multiple nodes within the network will now be compromised [12, 4, 27].

The broadcast session key negotiation protocol is based upon the master key pre-distribution solution. Each sensor receives a key prior to deployment and all other keys are based upon this key. A compromised master key results in a completely compromised network. The lightweight key management system is a modification of the broadcast session protocol. The lightweight system utilizes multiple groups (generations) in which each generation contains a unique key. These generations vary in size from entire networks to small subsets of nodes [4].

The random keychain-based key pre-distribution solution is based upon a probabilistic key scheme. This involves generating a large key-space and randomly drawing keys without replacement for use within the sensor networks. Probability is used such that sensors must have at least one key pair in common with each other and have to search through their list to obtain this information [4, 7]. The difficult part about this scheme is in what constraints to make on key generation and distribution to ensure a secure and reliable network.

The key-chains that are utilized in the random key-chain scheme are determined at runtime. The idea behind combinatorial design-based key pre-distribution schemes is to design the key-chains using combinatorial design theory. This ensures that sensor nodes will find exactly one common key and the probability of the network communication succeeding is 100 percent. Scalability becomes a problem in computing these key-chains for utilization [4].

In a key matrix based dynamic key generation solution, Blom's scheme utilized two N x N matrices. There are public and private key matrices. The disadvantage to this method is that all sensor nodes within a given radio range must respond with a computation of their two matrices. This is a significant overhead for the computational resources on nodes. The multiple space key pre-distribution scheme utilized multiple private matrices and a shared key discovery scheme. This eliminates some of the computation and communication overhead [4].

The polynomial based dynamic key generation scheme utilized a method to symmetric polynomial distribution for key generation. Each pair of sensor nodes can generate their unique key from initial information common to both. The location of the sensor node (if known) can be utilized to more effectively compute shared keys [14, 21, 24, 25]. This approach also is in utilization of a shared key discovery algorithm [4].

The lightweight key management system is a group-wise key distribution scheme. This utilizes a pair-wise key scheme that can be distributed to groups of sensors as they are deployed in different phases. The group-wise keys are distributed through the already secured pair-wise keys among sensors [4].

Network-wise key distribution schemes rely on a wireless sensor network to be based upon a tiered security level. For a master key solution, the base station can broadcast a network-wise key to the nodes. A better approach is to use a multi-tiered solution in which nodes are broken up into different security levels. The more sensitive the information, the more secure the node is required to be. The problem with both solutions is that once the master/network keys are compromised, the entire security level is compromised [4, 35].

Public key cryptography is difficult to implement within sensor networks. The computational overhead for generation and validation of keys is more than the sensor's processors can handle [6, 14, 26, 37, 38, 40]. One approach is to move the computationally expensive operations for the public key system to external components of the sensor networks. This is only possible when these resources exist [39]. This approach depends on extra resources that may not be available in all sensor networks. Another approach is allowing the network to manage and create the trust relations required for public key cryptography [5].

An alternative method of implementing public key cryptography is to include a custom hardware chip to offload the computational overhead of the public key system from the small processor [17]. This would allow the public key computations to occur more quickly and consume much less power. "The implementation of NtruEncrypt consumed less than 20μW" [17]. This is significantly better than a general purpose processor could hope for.

An alternative to basic public key cryptography is to utilize the elliptic curve cryptography [38]. This has been difficult to implement because of the time and resources required to generate the keys. It has been shown that it is possible to implement elliptic curve cryptography on the sensor nodes and have "reasonable" results [30]. The definition of reasonable is left to the system designer and is dependent on the conditions the network is being designed for.

	Benefits	Drawbacks	Key Compromise
Pair-wise key	Complete communication	Can not scale	Entire network
Random Pair-wise	Communication	Can not scale well	Almost entire network
Closest Pair-wise	Efficient	Requires sensor locations	Compromised area
ID-based	Low memory usage for keys	Compromised Nodes	Multiple nodes
Broadcast Session	Keys can be changed	Utilizes a master key	Entire network
Random Keychain	Space efficient	Scalability	Portions of the network
Group-wise	Better Scalability	Requires sensor locations	Groups of nodes
Public Key	Secure	Requires computation power	Groups of nodes
Point-to-Point Key	Scalable	Overhead to create keys	Captured node links

TABLE 2.1

Benefits and drawbacks of key distribution schemes

2.2. Authentication

Authentication is a component of the overall security of a sensor network and is necessary to allow sensors to verify and check the packets for malicious activity, which ensures the integrity of data [35, 20]. However, authentication can only go so far in protecting the network. A compromised node will be able to inject invalid data into the stream and still pass the authentication checks. This integrity ensures protection against man-in-the-middle attacks [42].

2.3. Availability

A sensor network must maintain availability. The entire goal of the network is to monitor the area and report its information back to a specified location. If the availability is broken, part of the network will be unable to perform this task. A Denial-of-Service (DOS) attack is the most common vector in which to attack the availability of the sensor network. An important feature of the network is to provide a graceful degradation as nodes are either compromised or reach the end of their operating life [35, 9, 11, 20, 28].

Availability is a real concern for applications of sensor networks within disaster areas, for utilization with public safety, and in home healthcare [41, 28, 29]. Each of these three areas present their own concerns for security and availability of the network. In the disaster area, the network will likely be subjected to harsh environment and the system needs to ensure availability and robustness. The system needs to provide information as long as possible with a significant number of nodes ceasing to function. This robustness should provide the availability for the network [41]. In the case of public safety, the networks would be monitoring areas for hazardous material. Availability is crucial to not miss anything. An attack on this kind of network could be cause to look for a real threat [41, 28]. The home healthcare environment has a necessity to provide high availability and confidential networks. These networks could provide vital signs and accident-notifications to the authorized users [41, 28, 29].

2.3.1. Sybil Attack

A common attack to sensor network availability is called the Sybil attack. This is an attack where a malicious node illegitimately claims multiple identities [35, 31]. This attack can occur at multiple logical layers. An attack to the routing of the network can alter the paths packets will take. By assuming multiple identities, a node can attempt to get more than its share of packets to come to it and selectively forward packets [31]. The Sybil attack can also be utilized at the Media Access Control (MAC) layer to attempt to consume all the available shared communication bandwidth. This has the effect of causing the real nodes to be unable to transmit their information [35, 31].

2.4. Scalability

Scalability is a critical problem for sensor networks. As most security protocols have been designed for node to base station security, this is a problem for massive (thousands) of sensor nodes operating within a single mesh network [35, 9, 6, 40]. Scalability is necessary in order for the networks to be successful. A point-to-point network which is arranged into groups of sensors will provide a very scalable system. Each node will only be concerned with communicating among its nearby neighbors and not with the entire network.

2.5. Efficient Cryptographic Systems

Sensor nodes are severely constrained in regards to their processor and power capabilities. This is a problem for traditional security measures due to the fact that many require a large computational capability to accomplish their tasks. There seems to be a direct correlation between the capabilities of the cryptographic protocol and the amount of power required to accomplish this encryption [35].

2.6. Attacks

A multitude of attack vectors exist for wireless sensor networks. Outsider attacks require robustness and strength of the protocols to provide security against passive monitoring of the communication channels. The outsider attack can also be by packet injection. Packet injection involves inserting incorrect data into an existing network. This attack can be prevented through the use of a large quantity of sensor nodes and placing the nodes in such a way that their coverage areas overlap. Overlap allows the sensors to verify the information provided from other locations in the network [35]. Insider attacks are much more difficult to create; however, if successful they should be virtually undetectable. In order to cause an insider attack, a sensor must be compromised through an attack sufficient to allows modification of the sensor functionality without showing this modification to the sensor network. One method is to physically capture a node and interface it with a more powerful computer. This computer will be able to utilize the sensor, in its original state, to insert the data it wishes into the network [35].

DOS is an effective vector with which to attack a sensor network. Sensor networks rely on wireless communications and low power system. A DOS attack on the wireless communications will leave little bandwidth for utilization by legitimate traffic. Jamming of the signal involves transmitting noise on the frequencies used within the sensor network for communication. This will prevent the nodes from communicating and cause DOS to occur. This technique is effective only within the area the jammer can transmit. This will stop an entire network, given a large enough deployment area. A slightly more effective method is to only transmit data to cause packet collisions. It is not necessary to jam the signal all of the time. Constant jamming will alert the network to the problem. Selective collisions of wireless packets is a more directed attack and should eliminate a selected area of the network [41].

Sensor network, unlike traditional wired networks like the Internet or local area networks (LAN), rely on their batteries for operation. Attacking this power source is a method to eliminate the sensor network. A DOS attack, using directed packet injection, aimed at the sensor nodes could cause them to waste their power receiving and examining the packets. Any other method of causing the sensor to waste power should be just as effective at attacking the sensors [41].

The prior two attacks relied on brute-force attacks to the network. A more power efficient and effective strategy would be to selectively target and attack the network. A homing attack is a method of DOS that relies on traffic observance prior to commencing the attack. By observing the traffic pattens in a network, the attacker will be able to identify the critical nodes. The network relies on these particular nodes to pass information through the network. These can either be a node that just happen to be located in a critical route or that have a more sophisticated system and/or more power. The attacker can target these critical nodes in order to more effectively bring down the network. A similar method to the homing attack is to flood the network with connections. By opening many connections to the victim, the attacker will cause the victim to consume its battery. This also will not allow this sensor to receive any legitimate connections, thus preventing it from participating with the network [41].

Misdirection and black hole attacks involve a similar method for compromising the network. Both vectors rely on the attacker to already have packet access to the network . The misdirection attack will take all the packets it receives and forward these packets in the wrong direction. This should cause the network to spend more battery power than necessary to correctly route this information. This route is usually chosen to target a single node that is likely to be a critical node [41].

A black hole attack is an advertisement of zero-cost routes. These zero-cost routes will make the sensor routing tables think the new node is the most power effective route. This has a two-fold effect: (1) the target node will be DOS attacked by its own system and (2) all of the packets can be lost to the attacking node. In the second case, the attacking node will simply discard the message or selectively forward the ones it wishes to allow. Either method will result in a compromise in the sensor network [41].

CHAPTER III
THEORY AND DESIGN

3.1. Wireless Sensors

A wireless sensor is a miniature computation device capable of detecting information about the environment, processing this information, and the ability to transmit this information to other locations. Each sensor is capable of functioning independently of the others. Multiple sensors comprise a sensor grid or network. This is the functional object which is designed to perform some kind of monitoring task.

This project uses the Mica2 mote and the MTS300 sensor board. The Mica2 mote (FIGURE 3.1) is powered by two AA batteries with a lifetime upwards of one year utilizing the sleep modes. There is 128 Kbytes of available flash memory for the program space. The board supports Universal Asynchronous Receiver/Transmitter (UART) serial communications. This device has a 10-bit Analog to Digital Converter (ADC) for monitoring analog sensors. The power consumption is 8mA in active mode and less than 15μA in sleep mode. This small amount of current allows this device to have a lengthy lifetime [10].

The multi-channel radio onboard the mote allows for bidirectional communication within the network. The center frequency is 916MHz. There can be four or five channels of communications utilizing frequency-shift keying (FSK). The data rate of the communications is 38.4 Kbaud. This device typically has a 500 foot line-of-site (LOS) range outdoors. The radio draws 27mA transmitting at maximum power and 10mA receiving data. The radio sleep mode draws less than 1μA [10].

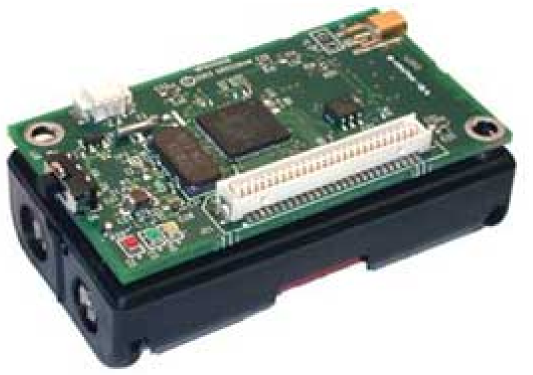

FIGURE 3.1 – Mica2 mote package from XBow [10]

The mote has 3 LEDs to provide feedback from the system. These are programmable through TinyOS and their functions are only what the developer programs. The mote is 58 x 32 x 7 mm and weighs 18 grams. An expansion connector of 51 pins is utilized to attach any number of sensor boards to the mote [10].

The MTS300 sensor board (FIGURE 3.2) is the basic sensor board which can detect light levels, sound levels, and temperatures, and is capable of sounding a tone as a feedback mechanism from the mote software. TinyOS can control how each sensor operates, such as turning the individual sensor on and off to conserver power. This board is not critical to this thesis's research; however, it will provide data to the sample simulation and tests [10].

3.2. TinyOS Framework

TinyOS is the open source framework which is utilized to develop and run embedded sensor networks. It is designed to support concurrency intensive operations required by sensor grids with a minimal hardware configuration. TinyOS utilizes a special language and compiler called nesC to program and develop for individual sensors [36].

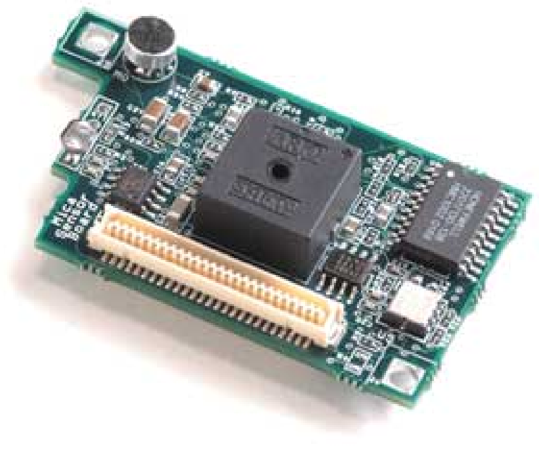

FIGURE 3.2 – XBow Sensor Board with Temperator, Sound, and Light sensors [10]

The TinyOS Framework is designed to be completely modular. By wiring components together, nesC allows for this component-based design with a C-like syntax. The modules can be written to perform a number of tasks from wireless grid communication to hardware interfaces to encryption routines. The robustness of the modules allows for easy and powerful expansion of the system, within bounds of the limited hardware [36].

3.3. TOSSIM

TinyOS SIMulator (TOSSIM) is the simulator developed for a TinyOS environment. This simulator is useful when developing sensor network applications. The sensor's small form factor and remote locations prohibit examination of the state of the sensor. This is where TOSSIM becomes very useful. The simulator can scale to more nodes than a developer usually has available. This scalability of the simulation allows developers to determine what will happen when the network scales to a particular size. Since TOSSIM emulates the operations of the sensors to the bit level, any possible experiments can occur and should function the same on physical sensors. This simulator provides a drop-in replacement for the actual sensor and all of the tools utilized to communicate and test the networks can be utilized within the simulator [23].

A graphical tool, TinyViz, allows visualization and interaction of sensors with the user. This is very useful to graphically see what the network is doing. The basic plugins allow for monitoring, in a graphical fashion, the radio packets and other communications, sensor status, and anything else which can be monitored. This graphical tool allows the user to move and manipulate the sensor positions to test the network under a variety of conditions.

3.4. Energy Consumption

Energy consumption is a direct result of a wireless transceiver and processing of data. The goal is to minimize the energy consumption while maximizing security. There are main methods of increasing the energy efficiency of the device. The first is to build more power efficient hardware. The other method is to develop more efficient software to take advantage of the hardware. TinyOS and some of the motes support sleep states to attempt to preserve power. The disadvantage of sleeping is that the node can not respond to anything while it is sleeping. Features and energy efficiency do not go hand in hand and sacrifices must be made as to which one is more important.

3.5. Design of Algorithms

This section describes how the algorithms were designed. It provides the necessary figures and discussion to replicate these algorithms.

3.5.1. Session Keys

Session keys are utilized for node to node communication. They exist only between any two communicating nodes. Session keys are typically deployed in large sensor networks and when nodes are resource constrained. These nodes are assumed to be immobile and must share a master key [22]. The BROSK method assumes the master key is the sole determining factor in authenticating other nodes [22].

3.5.2. Soft Key Distribution

This BROSK method forms the basis on which new key distribution procedures are built. The new process combines this procedure from Lai [22] and the soft authentication ideas from Park [32] to form a new mechanism for generating and ensuring security within wireless sensor networks.

The goal is to provide smaller keys, generated on demand as the network sees fit, to secure point-to-point communications among pairs of nodes. The authentication mechanism examines the sensors to ensure the validity of the underlying system.

Each procedure (FIGURE 3.3,3.4) will run concurrently on a pair of motes, with mote A running the (setupKey) and mote B will run the (validationRequest). These procedures will run concurrently. Prior to utilizing the key generation and authentication algorithms, all communications for the key determination occur over a pre-shared master key. This key is only to be utilized during the startup procedures of the sensor network. Once the network has been established, the generated keys are the only form of encryption.

Mote A discovers which mote it wishes to setup a key with and adds this id to a list. Mote A then generates four random byte locations, within the operating system code, which then computes a cyclic-redundancy check (CRC) based upon the data at these four locations. Mote A sends the four byte locations in a data packet

```
procedure setupKey(mote)
        add mote to list
        generate 4 random byte locations
        checksum <-- CRC(byte1, byte2, byte3, byte4)
        send byte[1..4] to mote
        motesum <-- response from mote
        if checksum == motesum then
                send confirmation message to mote
                byte <-- get bytes from mote
                checksum2 <-- CRC(byte1, byte2, byte3, byte4)
                send checksum2 to mote
                moteconfirmation <-- response from mote
                if moteconfirmation != null then
                        add moteconfirmation as key for the mote
                else
                        add 0 as invalid key for the mote
                endif
        else
                add 0 as invalid key for the mote
        endif
end
```

FIGURE 3.3 – First half of a procedure to setup a key on a pair of nodes.

```
procedure validationRequest(byte[1..4],mote)
        add mote to list
        checksum <-- CRC(byte1, byte2, byte3, byte4)
        send checksum to mote
        confirmation <-- response from mote
        if confirmation == true then
                 generate 4 random byte locations
                 checksum <-- CRC(byte1, byte2, byte3, byte4)
                 send byte[1..4] to mote
                 motesum <-- response from mote
                 if checksum == motesum then
                         key <-- generate random key
                         send key to mote
                         add key to list for use with mote
                 else
                         add 0 as invalid key for the mote
                 endif
        else
                 add 0 as invalid key for the mote
        endif
end
```

FIGURE 3.4 – Other half of the procedure to setup a key on a pair of nodes.

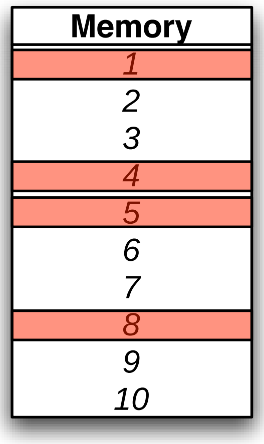

FIGURE 3.5 – Example Memory Map

to mote B (Figure 3.5), the discovered mote. Mote A now waits for a response from mote B. Mote B performs the same steps as mote A and adds mote A's ID to a list. Mote B computes the CRC and returns this value inside of a data packet to mote A. Mote A check the received value from mote B to ensure it is the same as its checksum. If the checksums do not match, then mark mote B as a compromised node. If the checksums match, then send a confirmation message to mote B. Mote B will now generate four random byte locations and compute the CRC for these memory locations. Mote B sends these byte locations to mote A. Mote A receives four byte locations and computes the CRC based upon this information and returns the checksum to mote B. If the checksum from mote A matches the one for mote B then mote B will generate a 16-bit cryptographic key. This key is added to the list for use with mote A and sent to mote A. Mote A will receive the key and add it to its list for use with mote B (Figure 3.6).

If anything happens in either procedure, the motes will assume a security problem has occurred and mark each other as invalid nodes. This will ensure the security and integrity of the procedures by not allowing a method for attack against the algorithm. This deny-on-error policy should guarantee these procedures to be hardened.

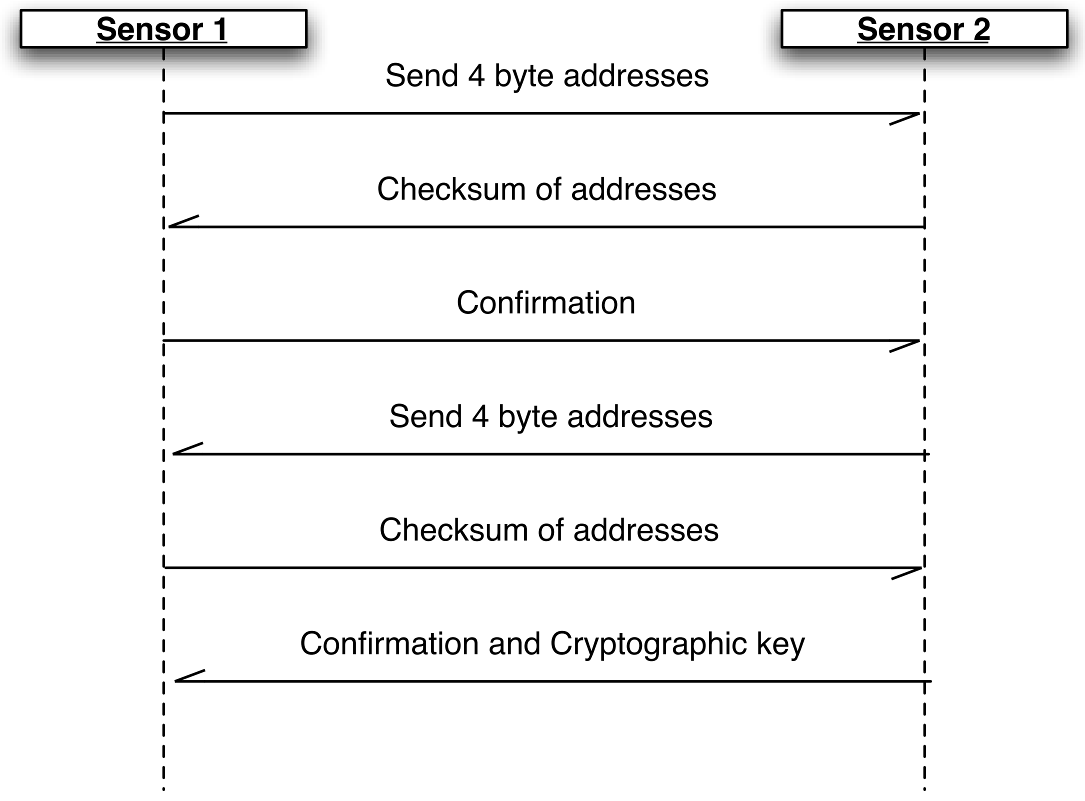

FIGURE 3.6 – Algorithm Message Passing

The controlling network layer should ensure the delivery of the data packets, even in the event of failures and interruptions. These two procedures will provide a 16-bit symmetric key for utilization between the two motes. This key is only valid for communication on this link. If a mote wishes to communicate with a different node, it must utilize the appropriate key for the link. This algorithm utilizes the 16-bit key; however, this key length may be changed to whatever is desired, within the limits of a single data packet. The current structure limits this key to be 160-bits.

Multi-hop communications will take a slight power and performance hit when routing information with the unique point-to-point keys. In order for data to route from one node to another node via multiple hops through different sensor nodes, the data packet will have to be decrypted and re-encrypted at each hop in the route. This will take only slightly more effort than a network utilizing a single master key. The major benefit to the point-to-point keys is with the extra security from node compromise. The losses from the decryption and re-encryption should not affect the power consumption and performance of the network very much. There will be a larger problem with power in multi-hop communications.

CHAPTER IV
IMPLEMENTATION

4.1. nesC

NesC is the language in which TinyOS is written. All modifications to the TinyOS system must be implemented in this language. NesC is a language for programming structured component-based applications. It has a C-like syntax, but supports the concurrency model from TinyOS. NesC allows for mechanisms for structuring, naming, and linking software components into embedded applications. This facilitates the designers in building components for complete concurrent systems [36].

One benefit of the nesC language is the ability to do extensive compile time testing and validation of software. The compilers can identify and detect many errors and problems both in syntax and in concurrency within the language. This provides valuable information to the developer. Concurrency errors are significantly reduced through the use of code checking [36].

The language, nesC, provides a component object. The application is comprised of one or more components linked together to form the application. Components provide and use *interfaces.* This is the only interaction available to the component. The interfaces are bidirectional for communication and interaction. The interface declares a set of methods called *commands* which must be implemented by the interface designer. These methods are similar to the sense of method in Java where they are part of a class. The other set of methods the interface declares is *events.* Events give the component a method to force the application designer to provide the implementation which will likely be unique to the individual application. These events are similar to Java interfaces. All events must be implemented to utilize the component [36].

There are two different types of components in a nesC application. The first is a module. Modules provide the application code for the platform. The second is the configuration. The configuration links the interfaces of the provided models together to form a working application. This linking is known as wiring [36].

The concurrency model is a unique design that is necessary for TinyOS to function with the hardware. The system contains two type of handlers: task and hardware event. Tasks are run in such a way that they run to completion and do not preempt one another. Hardware events are similar to tasks in that they run to completion; however, they may preempt another hardware event or task. Race conditions become a problem when multiple asynchronous handlers occur and can preempt each other. This is why TinyOS contains checks for race condition. There are methods that the programmer can employ to resolve the race conditions reported by the TinyOS compiler. These are discussed in the NesC documentation [36].

4.2. Application Code

Data and control values must be passed between motes in order to accomplish the key establishment protocol. A TOS_Msg (FIGURE 4.1) structure is the building block on which custom structures may be added for data exchange. The message contains the destination address (addr) and type of message (type). A group can be specified if necessary and the length of the message is stored. The data structure is where a custom structure can be added for different tasks. A CRC is computed prior to transmission to ensure the integrity of the data within the packet.

For the new protocol, an IntMsg structure provides the necessary information. The source of the data (src) is transmitted with the structure. This allows the

```
typedef struct TOS_Msg
{
        /* The following fields are transmitted
        and received on the radio. */
        uint16_t addr;
        uint8_t type;
        uint8_t group;
        uint8_t length;
        int8_t data[TOSH_DATA_LENGTH];
        uint16_t crc;
} TOS_Msg;
```

FIGURE 4.1 – TOS_Msg Structure (from AM.h)

```
typedef struct IntMsg
{
        uint8_t src;
        uint8_t val;
        uint16_t data[10];
} IntMsg;
```

FIGURE 4.2 – Int_Msg Structure

receiving mote to identify where the data is coming from and is important in the key establishment procedure (FIGURE 4.2). Message types are stored in the val attribute. This is utilized for tracking which data is contained within the data portion of the structure and is how the setup key algorithm keeps everything in order.

Passing of message between motes prompts the inclusion of the "val" attribute into the IntMsg structure. This is required to keep track of the process of key-generation. Each step of the key-generation algorithm updates this value with where its message should be routed. The receive method, which processing incoming packets, examines this information and calls the appropriate function to process either the next stage of key establishment or any other user defined procedure. This message handler is responsible for the decryption of the incoming message and routing this message to the appropriate location.

A procedure named *sendSecure* takes the place of the usual *Sendmsg.send* method. This function provides the required encryption procedure to encode the data portion of the TOS_Msg structure. If a point-to-point link is not setup for the particular communications, then this will initiate the setup procedure before sending the current message. The current message is stored in a backup object and completed at the conclusion of the key setup procedure. Since multiple messages are being passed between motes, a flag is set to indicate the presence of the backup data and the last thing the mote does is check this flag before completing the key establishment procedure.

The encryption procedure is a simple exclusive-or (XOR) between the data bytes and the encryption key. This is the simplest form of encryption possible as well as the most power efficient. Initially, the encryption occurs with the master key built into the system; however, this is only utilized for the initial creation of a point-to-point key. Once the point-to-point key is established, all other communications occur with the generated key. The lifetime of a key can be set so the system will renew a particular key after this limit is reached. This will allow for key changing to prevent unauthorized access to the data streams. Each mote contains a list of keys for use with its communications that are stored in RAM. A physical attack on the mote will not easily allow the contents of this RAM to be determined.

CHAPTER V
TESTING AND RESULTS

5.1. Testing Overview

There are two main components to the system: key generation and network communication. The first test is for the key generation and establishment of dynamic point-to-point keys; while, the second test is to ensure the scenario is functional for multi-hop communications. Final testing combines both the prior tests to form the basic structure of the sensor network security. This is tested for correctness.

5.1.1. Testing Key Generation

Testing the key generation algorithm involves setting up two sensor nodes (FIGURE 5.1) and having them establish a shared key. Through the use of debug messages, TOSSIM can be utilized to test the key generation algorithm. Appropriate messages are placed at critical sections of the algorithm to display the contents of memory concerning the key generation. These messages are also used to show the progression of the algorithm, much like hand-tracing an algorithm.

For testing purposes, the key generation algorithm is slightly modified. Instead of reading data directly from the program code, byte data will be read from the data EEPROM. The structure of TinyOS places a limitation on reading and writing information from within the program EEPROM. This change will only affect the implementation of the algorithm, but will provide similar features and energy consumptions. For a practical implementation, TinyOS will need to be modified to allow

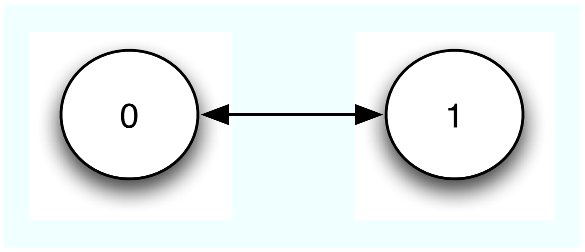

FIGURE 5.1 – Key Generation Test Setup

reading of the program EEPROM data.

Node 0 and 1 will utilize the master key to secure the initial communication and establish a shared secret key. Once this key is established, some data will be transmitted between the nodes to ensure the communications and encryption are properly implemented. A default key utilization length of five transmissions is implemented within this application.

The following is the output excerpts from a sample run of the test. These results (TABLE 5.1) show the progression of the key generation algorithm. The debug dump message show the state of the data contained within the packets as well as the node transmitting and a status value for process control. Each step of the algorithm will utilize a different number of elements within the data packet. At the end of the process, an encryption key is generated and provided to each node for their private communications.

5.1.2. Testing Network Communication

Testing the network communications involves establishing a sensor network, in simulation, to verify the communications are functioning. This test will pass information from the initial node through a series of hops to the final destination node. (FIGURE 5.2) This is done with no encryption of any kind on the network, just a basic network.

```
1: TIMER Message: 0
1: Dump: 1 7 12321,46472,23066,38187,6976,6047,3617,15709,23461,38485
1: Dump2: 1 7 12321,46472,23066,38187,6976,6047,3617,15709,23461,38485
1: Dump: 1 100 7612,6759,5585,2749,0,0,0,0,0,0
0: Dump: 0 101 61560,0,0,0,0,0,0,0,0,0
1: Completed sending message
1: Dump: 1 102 11653,10846,9704,14980,12345,12345,12345,12345,12345,12345
0: Completed sending message
0: Dump: 0 103 53954,42243,19085,34196,12345,12345,12345,12345,12345,12345
1: Completed sending message
1: Dump: 1 104 43365,6759,5585,2749,0,0,0,0,0,0
0: Completed sending message
0: Dump: 0 105 2000,38202,31412,46509,0,0,0,0,0,0
1: Completed sending message
1: Mote Link 1 <--> 0 is complete. Key: 2000
1: Dump: 1 7 12321,46472,23066,38187,6976,6047,3617,15709,23461,38485
0: Completed sending message
```

TABLE 5.1

Simulation Results for Key Generation Testing

4: Sent Message: (Node,value) 3,1
3: Received Message: (Node,value) 4,1
3: Sent Message: (Node,value) 2,1
2: Received Message: (Node,value) 3,1
2: Sent Message: (Node,value) 1,1
1: Received Message: (Node,value) 2,1
1: Sent Message: (Node,value) 0,1
0: Received Message: (Node,value) 1,1

TABLE 5.2

Simulation Results for Network Communications Testing

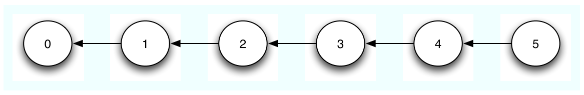

FIGURE 5.2 – Network Communications Test Setup

This test will send data from node 5 to node 0 by passing the data packet to each node in the route. Debug messages are produced by each node upon reception and transmission of a data packet. This allows for easy analysis of the route and the functionality of the sensor network. The results (FIGURE 5.2) show the debug messages and progression (FIGURE 5.3) of data through the network. The arrows are the messages passing to the next node in the network. Meanwhile, the circle is a broadcast message from node 0 looking to see if any other nodes are in the network. The debug messages show node 4 sending a message to node 3, then node 3 sends a message to node 2 and this continues until node 0 receives the message. This is just an excerpt from the simulation data; the rest can be found in appendix II.

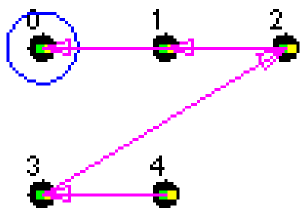

FIGURE 5.3 – TOSSIM Network Test

5.2. Experimental Design

Given the procedures work as designed, the question is how much energy is it going to cost to deploy this algorithm and security system? This cost affects the capabilities of the network; however, when security is required, what kind of threat is there to the integrity of the communications and networks.

5.2.1. Sample Tree Application

The following application is for the testing of a basic multi-route wireless sensor network. It is developed and simulated completely on the computer. The goal is to monitor information at the bottom of the tree and pass this information through the sensor network back to the root node. This scenario is used because of the ease of implementation and predictability of the network.

Nodes follow two simple rules. First, generate a random number between zero and seven and send this information to the parent. This is the data the network is to provide to the root node. This routing of information ensures delivery to the

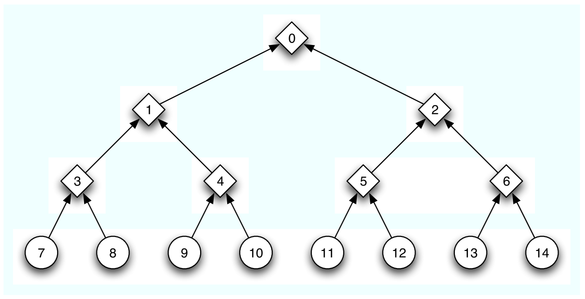

FIGURE 5.4 – Routing diagram of the Tree test application

root node. With this first rule, the nodes will just generate numbers and send their information to their parents. This does not provide a route or method for multi-hop transfer. The second rule is if a node receives a radio packet with data, the node will cease to generate data and become a intermediary node within the tree structure. Once a node has determined if it is an intermediary node, its function changes to receive the data from both of its children and determine the larger of the two. This information is then passed along to its parent. The circles denote data/sensing nodes. This is where the data is generated to be passed along to the root node. The diamonds denote intermediary nodes whose only goal is to pass the largest of its children up the tree. The directed graph shows the routing of information to the root node. (FIGURE 5.4)

This application is tested with TinyViz for two scenarios. The first one being a basic unencrypted network of 15 nodes. TinyViz, with the directed graph and radio link plugins enabled, shows the radio packets in the network and is utilized to test and verify the simulation. Power profiling is tested with the test scripts (APPENDIX I). The profiling simulation will run for 180 seconds. Node power consumption is reported in Figure 5.5.

The second test is with the new encrypted point-to-point communications. The parameters are kept the same. 15 nodes will be utilized for the test and the power profiling will run for 180 seconds. Node power consumption is reported in

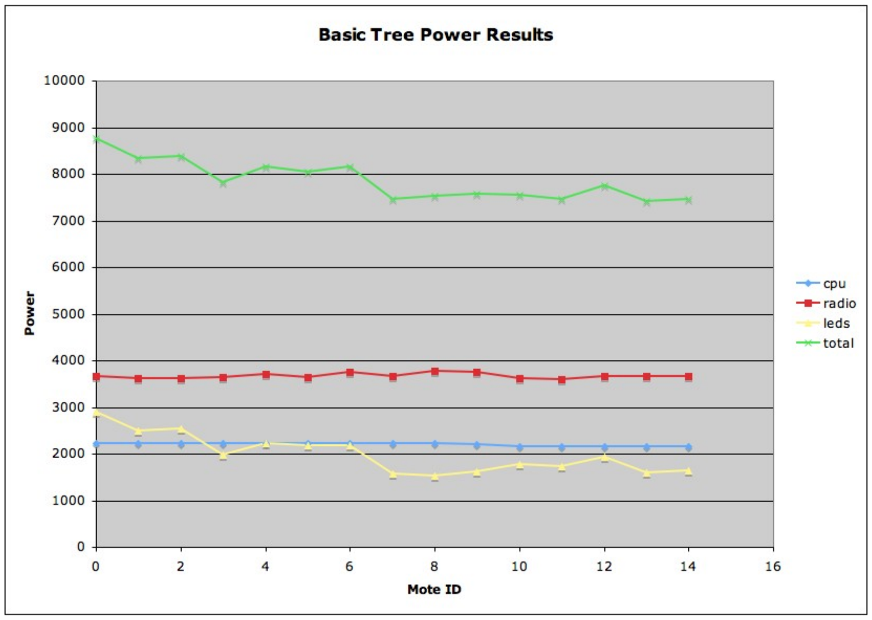

FIGURE 5.5 – Power consumptions for the Basic Tree Applications

Figure 5.6.

The differences between each test are provided in Figure 5.7. The power capacity of a typical AA battery is 2700mah. The motes utilize two batteries, providing them with 5400mah of power. The average utilization of energy (mJ) over the three minute trial is 7836mJ. This is equilivent to $\frac{7836mJ}{180sec} = 43.5mW$. $\frac{43.5mW}{3V} = 14.5ma$ is the amount of current consumed each second by each mote. The approximate lifetime of a sensor running at full power with no sleep is $\frac{5400mah}{14.5ma} = 372hours = 15.5days$. This is independent of the encryption technology overlaid on the communications.

An initial comparison of the results shows no noticeable differences between the two tests. Since the tests ran for the same length of time this is expected. With each mote running full speed, the cpu consumed a similar amount of power across both tests for all motes. The same holds true for the radio power consumption. The LEDs caused some variance to the total power levels. This is due to asynchronous nature of a mote. There is no guarantee exactly when the led will change.

These two tests show the energy consumption of an active running sensor

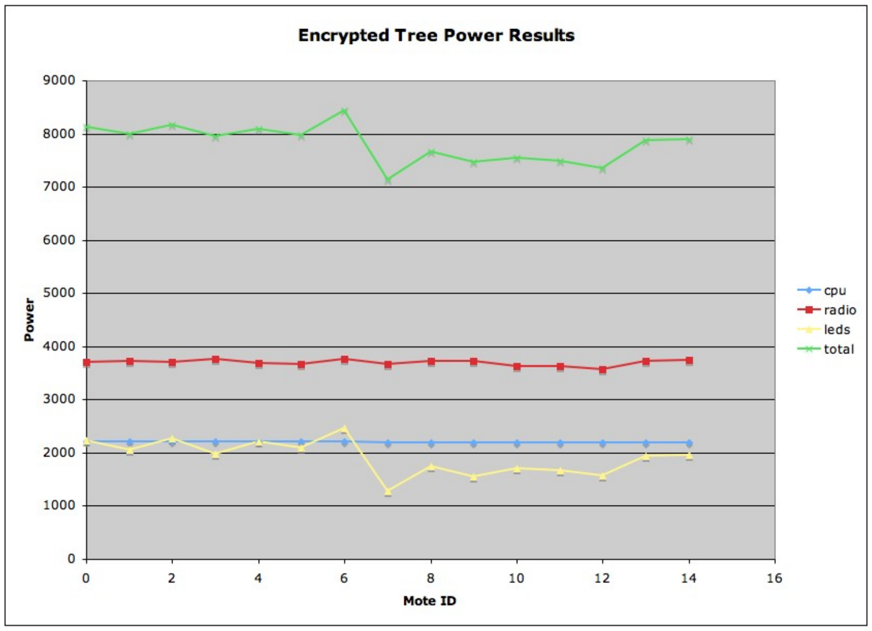

FIGURE 5.6 – Power consumptions for the Encrypted Tree Applications

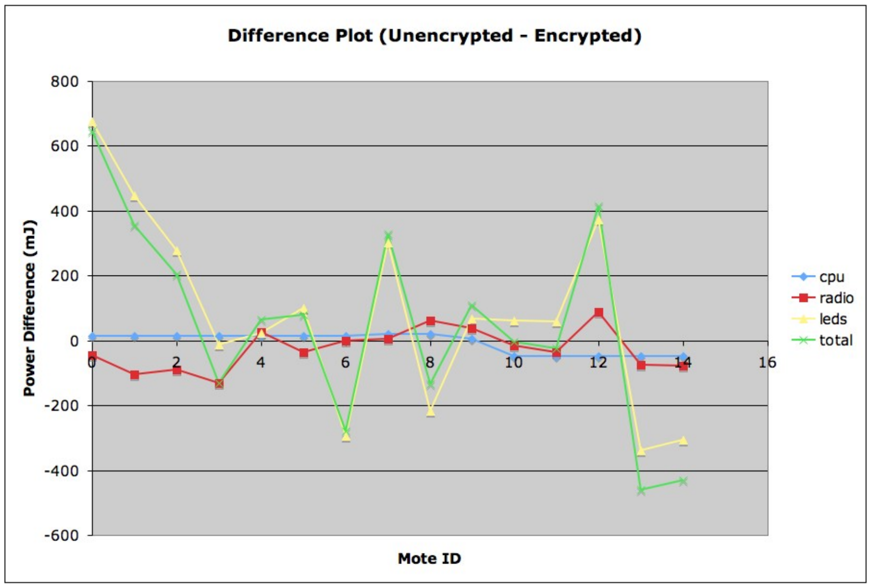

FIGURE 5.7 – Power consumptions difference between encryption and plain

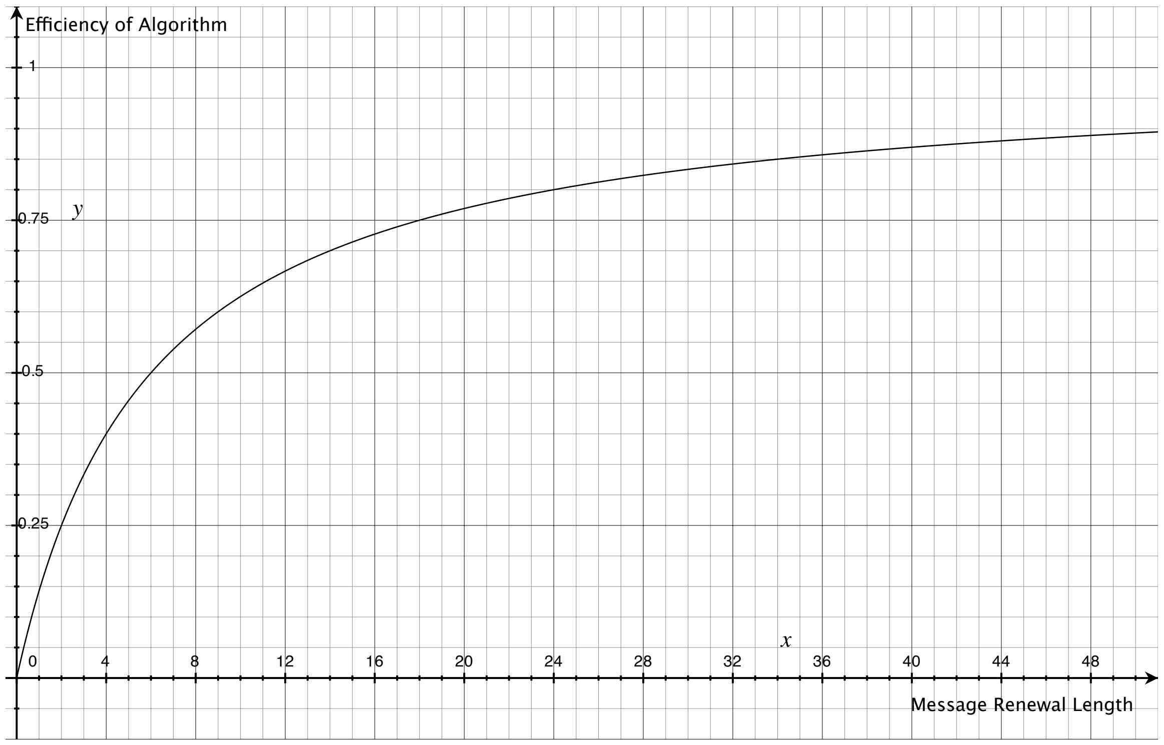

FIGURE 5.8 – Power Efficiency of the Algorithm

network. Due to the overhead within the protocol (radio messages and cpu time), the encrypted test did not complete nearly as many bottom to top communications as the basic unencrypted test. For each effective message, the encrypted technique requires on average two radio message to send. There is an overhead of radio packets in the key creation and renewal algorithms. This overhead is the component to minimize for optimal power consumption.

The actual efficiency of the new protocol is 45 percent. This is from the total renewal period of messages. 11 total messages for sending 5 data messages yields the 45 percent efficiency. Figure 5.8 shows the effects of changing the renewal time (number of messages) before requiring a new key to be generated. A 45 percent efficiency show that the encrypted version of the software will only provide 45 percent of the amount of data in the given time period and energy consumption level.

5.3. Security

There are three main security problems with the new protocol and with any sensor network. Intrusion into the network is a serious problem and can result in the collapse of the functional network. The integrity of the wireless encryption is critical to keeping intrusions at bay and the communicated information secure. Wireless integrity is the most likely attack method and potentially the most damaging. The last major topic is that of misuse and abuse of the sensor network systems. Once an intrusion has occurred, the attacker will be able to manipulate the network at will and change data within the network.

5.3.1. Intrusion

Intrusion into the sensor network involves breaking either the encryption keys or compromising the nodes. A compromised node is a significant problem for the safety of the network. This compromised node can also be used to affect the network communications through the use of flooding or black-hole attacks. These attacks mostly depend on how the routing is implemented within the wireless network. Many of the problems occur when the compromised node's security keys are discovered. The new key establishment protocol protects this network communication because each node-to-node link is unique. If a link is broken, only the compromised encryption is lost, not the rest of the network. If this link is discovered, through intrusion detection means, the area around the compromised key can be isolated and removed from the active network. The compromised key allows incorrect and misleading information into the network and the attack will be able to create an environment that does not exist. For example, a sensor network is monitoring an area for vehicle movement. A compromised node(s) will be able to simulate this motion through data injection into the network. Another method is to utilize the nodes to show normal data in the presence of vehicle movement, preventing the network from detecting it.

The new key establishment protocol provides an alternative method to the problems associated with pre-distribution procedures. These key pre-distribution methods have significant problems when a key compromise occurs. Attackers usually will acquire access to all the network's communications once the key or node has been compromised. All of the algorithms for encryption and decryption will be visible from this compromised node. These procedures will provide insight into how the encryption layer is constructed on the network and allow for a targeted attack against the network's security. For example, a compromised node has allowed the attacker to discover the encryption key for the network. Most likely, this node will contain enough information to easily communicate with the other nodes. This communication is all that is necessary to allow misinformation to be injected into the network.

5.3.2. Breaking the Encryption

The most likely attack on a sensor network is a compromise of the wireless encryption. As discussed in the prior section, this can be accomplished by discovering a sensor node and examining its code to determine how the encryption works and the relevant keys. Another method is to attack the wireless signals directly. This is either a passive or active attack. In the passive attack, the wireless traffic is collected from the sensor nodes. This traffic is analyzed on a powerful microcomputer in an attempt to discover the data within the communications and the key utilized in the encryption. An active attack involved sending and receiving data packets at a sensor node in an attempt to discover how the encryption works. This attack is usually detected quickly and stopped before any significant damage can occur.

For example, compromising a 16-bit key with a powerful microcomputer, in the worst case, one needs to examine $2^{16} = 65536$ different keys and $\frac{2^{16}}{2} = 32768$ different keys, for the average case. All of these keys would have to be tried against

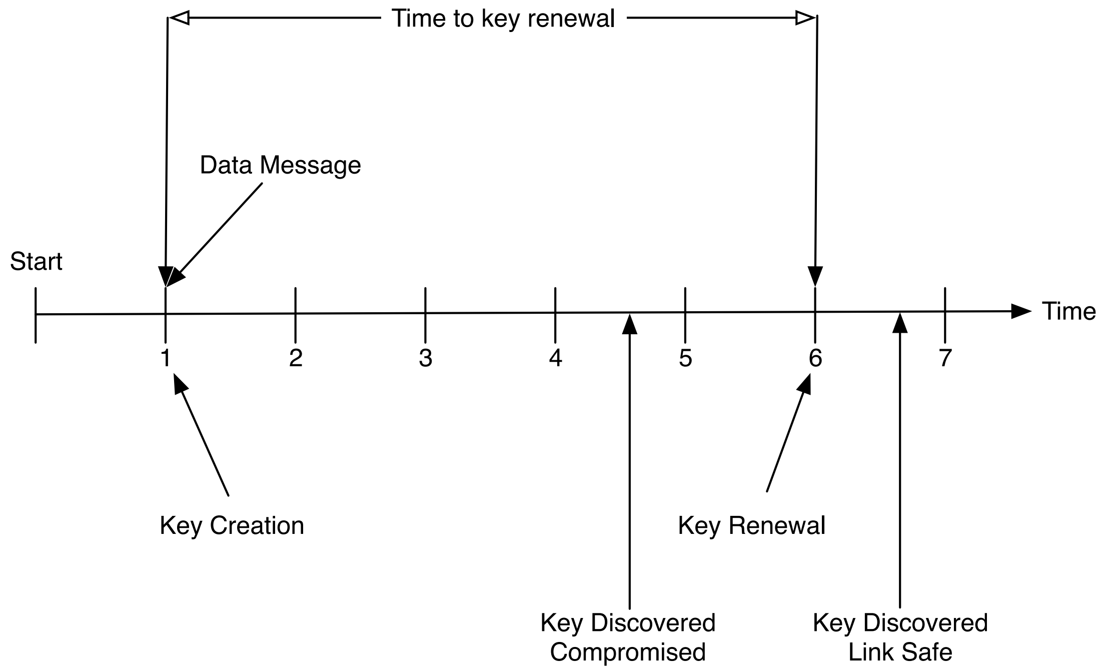

FIGURE 5.9 – Key Renewal Algorithm

the captured data. Given, the encrypted data portion of the example structure is 20-bytes long and the keys are 16-bits (2-bytes) in length, the computer would have to perform $2^{16} keys \times \frac{20bytes}{2bytes} \times 5packets = 3,276,800$ trials. Assuming the computer can perform each trial in 1ms, this computation will take 3276.8 seconds which is 54.6 minutes to try all the keys. This brute force method is practical if the key renewal time is longer than it takes to try the keys and make a decision about the correctness of each key. This is the minimum time required to break this encryption. Many other factors influence the strength of encryption. A longer key will require more time to evaluate the possible keys. For each bit that the key grows, the time required to check all the keys doubles. This example assumes the decryption routines are known. A more sophisticated encryption technique, such as a Advanced Encryption Standard (AES), would better secure the data. The price of security in sensor networks is limited lifetime and speed. AES will consume a lot more power than XOR (Figure 5.9).

The difficult part of breaking any type of encryption is determining if this trial contains the actual key. The data can be arranged such that there are 65536 different sets of attempted decryptions, with each set containing 5 separate packets. One method for attempting to discover the key is to look at the decrypted data (spanning all recorded data packets) for decryptions showing “reasonable” values. These reasonable values will have to be determined from environmental measurements of the data being sensed and detailed knowledge of the outputs of the sensors on the devices. If the device’s sensor is a custom design and not widely available, it may be necessary to capture a device and experiment on the sensor in order to determine “reasonable.” Data packets contain little information (20-bytes max). The process of breaking the encryption is going to be difficult given this small sample size. The ability of the attacker to determine the type of information contained within this data packet is critical to the speed and certainty with which the encryption may be broken.

Longer key lengths or more sophisticated encryption techniques will better secure the data and communications; however, the cost to implement and deploy these techniques may be more than the sensor networks can handle. Consider the prior example, except utilize a 32-bit key instead of 16-bit key. This means there are 2^{32} = 4,294,967,296 = 4.3 billion keys available. If the same computer is utilized to attempt to break the encryption, the computation will take $107,374,182,400$ trials which results in about 3.4 years of computation. There are $2^{160} = 1.461 \cdot 10^{45}$ keys possible in a 160-bit key length scheme. The processing time for brute-forceing this length of key is $4.63 \cdot 10^{37}$ years. This is the maximum key length supported by the data structures. There is a significant cost to compute cryptographic keys and encrypting the data on the sensor nodes prior to transmission. A similar cost occurs when the packet is received and decrypted. For this reason, small key sizes should be utilized for the encryption. The cost of key generation and encryption outweighs the cost of renewing a smaller key more times in a given period.

5.3.3. Misuse or Abuse

Misuse or abuse can occur when a foreign entity gains access to the sensor network. This is usually done through intrusion by a physical means or breaking the wireless encryption of a network. Once access is gained, under typical pre-distribution schemes, the intruder should be able to identify the encryption keys to most of the other network nodes. This will allow the exploitation of these nodes through the use of target messages or different routing of packets to misinform the intended node. This misinformed node will act as if the messages it received are valid and perform the necessary actions based upon the programming and data. Misinformation will allow the attacker to hide real information from the network, causing blind spots in the sensing ability of the network. The sensor can be fooled into thinking they are passing information along about a target, when in reality, there is nothing happening at that location.

This new point-to-point protocol separates each node from any other node more that a single hop away. This isolates the nodes into groups based on their communications. When a particular node has been compromised, only the data associated with the node needs to be invalidated. Compromised nodes do not provide keys for the rest of the network. The advantage of using the point-to-point protocol verses some kind of pre-distribution key is the integrity of the other point-to-point links is preserved. If the infrastructure supports detection of malicious nodes, this method will allow the entire rest of the system to continue to function unimpeded.

Abuse occurs when malicious nodes or outsiders attempt to consume resources within the sensor network. For a malicious node case, the protocol can do nothing to prevent DOS attacks on the rest of the network. The only thing the sensors can do is to ignore, as best they can, the incoming data packets. This does not stop the attack but hopefully reduces the impact on the sensor network's lifetime. The abuse can occur as a flood of data or attempts to change and disrupt the routing of the sensor network. There is no protection given by any encryption protocol to this type of abuse and the point-to-point protocol does nothing to stop this. The only hope is to prevent the abusive node from entering the network with the use of the encryption and not rely on stopping them once inside.

CHAPTER VI
CONCLUSIONS AND FUTURE DIRECTIONS

6.1. Conclusions

Wireless sensors will become more commonplace in the future, as the price of each individual sensor drops. These sensor networks will start to become part of everyday life and the security of their communication will become a concern. Good security is achieved through the practice of good implementation techniques and hardware technology. It is certain that security of sensor networks will be required and good security will rely on the development of new technologies.

In this thesis, a unique adaptation of two pre-existing ideas was utilized to provide a method to verify which node the communications are occurring with and develop a shared symmetric key between the two nodes. This algorithm allows individual nodes to choose who they wish to communicate with and generate a shared secret key with the node. This secret is only shared between these two nodes and is useless to any other pair of nodes on the sensor network. This secret key can be changes at necessary. The changing of the key is necessary because of the small size of the key.

Once deployed, sensor networks are at the mercy of their environment. This lack of physical security, at the sensor level, places a burden on the hardware and software. Not only is the software required to be energy efficient, it must be secure in its implementation. The hardware must be designed to prohibit discovery of the underlying software in the event of a captured node.

A current problem with sensor networks is with the effect of a compromised or malicious node. The compromised node can do serious damage to an existing network. For a pre-shared key scenario, the attacker can use the information contained within the node to target and manipulate all the other nodes in the network. In the new point-to-point algorithm, the compromised node is only able to communicate directly with its neighbors, thus eliminating the attacker's ability to capture a single node and attack the entire network. This security style effectively isolates each node from the rest of the network in the event of a failure.

The ability of the point-to-point system to accommodate a variety of encryption techniques is a good benefit. The system is designed to support small keys to provide energy efficiency as well as large keys and sophisticated encryption algorithms to provide more security. This ability to easily allow different key sizes allows for easy expandability of the type and strength of the encryption.

The results of the energy comparison between the two different setups show the effects of encryption. The encryption is shown to consume more energy than the unencrypted version. This is expected due to the computation and packet overhead needed to setup and maintain this encryption. The single largest benefit obtained from utilizing this point-to-point protocol is the ability to change keys and the utilization of smaller key lengths.

For an actual implementation and deployment, the base system, TinyOS, must be modified to allow the application to be able to read the program code data in order to perform the validation of the sensor. For testing purposes, this was not done and data was read from a different data memory. This is acceptable for simulations; however, it should not be used for practical implementation.

The random number generator in the TinyOS package is a linear feedback shift register (LFSR). This is a basic algorithm used to generate a string of pseudo-random numbers. A significant problem for the sensors is how this LFSR is initialized.

Currently, the seed value, is set to the mote ID. This lack of environmental seed significantly limits the random number sequences available to the mote.

This research has shown that there is a relationship between security and power efficiency. One cannot have both. There must be a line drawn between the two based upon the needs of the application. Security research is a tricky topic because it has no clear methodology for testing the security of the system. This is clearly evident in the Internet with the proliferation of thousands of viruses. Each virus takes advantage of a breakdown of security and there is no single solution to the problem. There are best practices, but no single solution. The best method for research is to examine security as a component of a larger scenario. This will allow security to be developed as part of a system, and avoid the problems of developing a security system and then finding an application for it.

6.2. Future Directions

Sensor networks are still in their infancy. There is still much to learn in terms of their capabilities and the drawbacks. The sensor networks utilized in this research are homogeneous. This allowed for each node to easily verify the validity of the others. A new direction for research would be to include multiple hardware configurations into the network. This will allow for larger hardware to provide the long-range communications and more powerful computation abilities. This would solve the problem with communication power consumption. The new device could contain more powerful batteries to provide the necessary resources to the nearby nodes. This approach is typically described as a flock and shepherd. This is one method of clustering nodes together. The drawbacks are in the validation algorithm. The new, more powerful nodes, will have to be able to communicate and validate the smaller sensor nodes as well as perform their own validations between other new nodes. This tiered approach to sensor networks adds complexity to the system; however, the benefits can be significant.

Another direction would be to improve on the random number generator. The simple LFSR would benefit from a more powerful algorithm as well as a hardware based random number generator. If a chip could be built to generate better random numbers, the whole system would benefit from the availability of these new random numbers.

The energy consumption is a good direction to pursue with new research. This is a critical component to the lifetime of a sensor network. Barring better power sources, energy consumption minimization is the only way to extend the life of a network. From this research, more sophisticated test cases are needed to better understand the energy consumption of real world networks. The example chosen is a simple, easy to test and trace algorithm. The real world applications will not be so trivial. Energy mapping of a network could show where the point of failures will occur. This mapping could be utilize to reposition sensors to better utilized the available resources. It could also be used to redefine the routing system within the network to spread out the power consumption.

REFERENCES

[1] I.F. Akyildiz, W. Su, T. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: A survey. *Computer Networks*, 38, 2002.

[2] Yair Amir, Cristina Nita-Rotaru, Jonathan Stanton, and Gene Tsudik. Secure spread: An integrated architecture for secure group communication. *IEEE Transactions on Dependable and Secure Computing*, 2(3):248–261, Jul–Sep 2005.

[3] Michael Brownfield, Yatharth Gupta, and Nathaniel Davis IV. Wireless sensor network denial of sleep attack. *Proceedings of the IEEE Workshop on Information Assurance and Security*, pages 356–364, 2005.

[4] Seyit A. Camtepe and Bülent Yener. Key distribution mechanisms for wireless sensor networks: A survey. *IEEE Transactions on Mobile Computing*, 4(3), 2005.

[5] Srdjan Capkun, Levente Buttyán, and Jean-Pierre Hubaux. Self-organized public-key management for mobile ad hoc networks. *IEEE Transaction on Mobile Computing*, 2(1):52–64, Jan-Mar 2003.

[6] H. Chan and A. Perrig. Pike: Peer intermediaries for key establishment in sensor networks. *Proceedings of IEEE Infocom*, 2005.

[7] H Chan, A Perrig, and D Song. Random key predistribution schemes for sensor networks. *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 197–213, 2003.

[8] Haowen Chan, Virgil D. Gilgor, Adrian Perrig, and Gautam Muralidharan. On the distribution and revocation of cryptographic keys in sensor networks. *IEEE Transactions on Dependable and Secure Computing*, 2(3):233–247, Jul–Sep 2005.

[9] Haowen Chan and Adrian Perrig. Security and privacy in sensor networks. *IEEE Security*, October 2003.

[10] Inc. Crossbow Technology. http://www.xbow.com/.

[11] Jing Deng, Richard Han, and Shivakant Mishra. Enhancing base station security in wireless sensor networks. April 2003.

[12] R Di Pietro, LV Mancini, and A Mei. Random key-assignment for secure wireless sensor networks. *Proceedings of the 1st ACM workshop on Security of ad hoc and sensor networks*, pages 62–71, 2003.

[13] W Du, J Deng, YS Han, PK Varshney, J Katz, and A Khalili. A pairwise key predistribution scheme for wireless sensor networks. *ACM Transactions on Information and System Security (TISSEC)*, 8(2):228–258, 2005.

[14] Wenliang Du, Jing Deng, Yunghsiang S. Han, Shigang Chen, and Pramod K. Varshney. A key management scheme for wireless sensor networks using deployment knowledge. *IEEE Conference on Computer Communications*, 2004.

[15] L Eschenauer and VD Gligor. A key-management scheme for distributed sensor networks. pages –.

[16] Huirong Fu, Satoshi Kawamura, Ming Zhang, and Liren. Replication attack on random key pre-distribution schemes for wireless sensor networks. *Proceedings of the IEEE Workshop on Information Assurance and Security*, pages 134–141, 2005.

[17] Gunnar Gaubatz, Jens-Peter Kaps, and Berk Sunar. Public key cryptography in sensor networks – revisited. *First European Workshop on Security in Ad-Hoc and Sensor Networks*, August 2004.

[18] R. Gennaro, A. Lysyanskaya, T. Malkin, S. Micali, and T. Rabin. Algorithmic tamper-proof (atp) security: Theoretical foundations for security against hardware tampering. 2004.

[19] Fei Hu, Jim Ziobro, Jason Tillett, and Neeraj K. Sharma. Secure Wireless Sensor Networks: Problems and Solutions.

[20] Linqxuan Hu and David Evans. Secure aggregation for wireless networks. *IEEE Workshop on Security Assurance in Ad Hoc Networks*, 2003.

[21] Dijiang Huang, Manish Mehta, Deep Medhi, and Lein Harn. Location-aware key management scheme for wireless sensor networks. *ACM Workshop on Security of Ad Hoc and Sensor Networks*, pages 29–42, October 2004.

[22] Bocheng Lai, Sungha Kim, and Ingrid Verbuwhede. Scalable session key construction protocol for wireless sensor networks. *IEEE Workshop on Large Scale Real-Time and Embedded Systems*, December 2002.

[23] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: Accurate and scalable simulation of entire tinyos applications. *ACM Conference on Embedded Networked Sensor Systems*, pages 126–137, November 2003.

[24] Zang Li, Wade Trappe, Yanyong Zhang, and Badri Nath. Robust statistical methods for securing wireless localization in sensor networks. *Proceedings of the International Conference on Information Processing in Sensor Networks*, pages 91–98, April 2005.

[25] D Liu and P Ning. Location-based pairwise key establishments for static sensor networks. *Proceedings of the 1st ACM workshop on Security of ad hoc and sensor networks*, pages 72–82, 2003.

[26] Donggang Liu and Peng Ning. Establishing pairwise keys in distributed sensor networks. *ACM Conference on Computer and Communications Security*, October 2003.

[27] Donggang Liu, Peng Ning, and Wenliang Du. Group-based key pre-distribution in wireless sensor networks. *Proceedings of ACM Workshop on Wireless Security*, 2005.

[28] Konrad Lorincz, David J. Malan, Thaddeus R.F. Fulford-Jones, Alan Nawoj, Antony Clavel, Victor Shnayder, Geoffrey Mainland, Matt Welsh, and Steve Moulton. Sensor networks for emergency response: Challenges and opportunities. *IEEE Pervasive Computing*, pages 16–23, 2004.

[29] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless sensor networks for habitat monitoring. *ACM International Workshop on Wireless Sensor Networks and Applications*, September 2002.

[30] DJ Malan, M Welsh, and MD Smith. A public-key infrastructure for key distribution in tinyos based on elliptic curve cryptography. *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, pages 71–80, 2004.

[31] J Newsome, E Shi, D Song, and A Perrig. The Sybil attack in sensor networks: analysis & defenses. *Proceedings of the third international symposium on Information processing in sensor networks*, pages 259–268, 2004.

[32] Taejoon Park and Kang G. Shin. Soft tamper-proofing via a program integrity verification in wireless sensor networks. *IEEE Transactions on Mobile Computing*, 4(3), 2005.

[33] A Perrig, J Stankovic, and D Wagner. Security in wireless sensor networks. *Communications of the ACM*, 47(6):53–57, 2004.

[34] Mahalingam Ramkumar. Safe renewal of a random key pre-distribution scheme for trusted devices. *Proceedings of the IEEE Workshop on Information Assurance and Security*, pages 142–149, 2005.

[35] Elaine Shi and Adrian Perrig. Designing secure sensor networks. *IEEE Wireless Communications*, December 2004.

[36] TinyOS. http://www.tinyos.net/.

[37] Patrick Traynor, Heesook Choi, Guohong Cao, Seneun Zhu, and Tom La Porta. Establishing pair-wise keys in heterogeneous sensor networks. 2004.

[38] AS Wander, N Gura, H Eberle, V Gupta, and SC Shantz. Energy analysis of public-key cryptography for wireless sensor networks. *Third IEEE International Conference on Pervasive Computing and Communications (PERCOMÍ05)*, 2005.

[39] R Watro, D Kong, S Cuti, C Gardiner, C Lynn, and P Kruus. Tinypk: securing sensor networks with public key technology. *Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*, pages 59–64, 2004.

[40] R. Wei and J. Wu. Product construction of key distribution schemes for sensor networks. May 2004.

[41] Anthony D. Wood and John A. Stankovic. Denial of service in sensor networks. *IEEE Computer*, pages 54–62, 2002.

[42] Qi Xue and Aura Ganz. Runtime security composition for sensor networks (securesense). *IEEE Vehicular Technology Conference*, 2003.

[43] Chee yee Chong and Srikanta P. Kumar. Sensor networks: Evolution, opportunities and challenges. *Proceedings of the IEEE*, 91(8), August 2003.

APPENDIX I
TinyOS System

1.1. TinyOS Installation

The following procedure is the method used to install TinyOS.

1. Download and install the stable version of TinyOS from
   `http://www.xbow.com/Support/Support_pdf_files/tinyos-1.1.0-1is.zip`
   This will include all the necessary components for the Cygwin, TinyOS packages, TinyViz, and the special Crossbow libraries.

2. To launch the system, run the Cygwin shortcut on the desktop.

3. Navigate to the tinyos system
   ```
   bash$ cd /opt/tinyos-1.x
   ```

4. You are now ready to use the TinyOS system.

1.2. TinyViz

TinyViz is installed with TinyOS. This is the simulator for running motes. In order to run a simulation, one must follow this procedure:

1. Build the application for the computer
   ```
   bash$ make pc
   ```

2. Enable the appropriate debug environment variables. For example:

```
bash$ export DBG=usr,usr2,usr3
```

3. Run main.exe and save the output to a trace file

```
bash$ /opt/tinyos-1.x/tools/java/net/tinyos/sim/tinyviz -run
./build/pc/main.exe 15
```

1.3. Power Profiling

1. Build the application

```
bash$ make pc
```

2. Enable the power debug environment variable

```
bash$ export DBG=power
```

3. Run main.exe and save the output to a trace file

```
bash$ ./build/pc/main.exe -t=60 -p 15 ¿ tree.trace
```

4. Run a post process script to compute the power consumption for each mote

```
bash$ /opt/tinyos-1.x/tools/scripts/PowerTOSSIM/postprocess.py –sb=0
–em /opt/tinyos-1.x/tools/scripts/PowerTOSSIM/mica2EnergyModel.txt
tree.trace
```

5. Single node example results which can be directly compared to other nodes to determine relative energy consumption.

```
Mote 0, cpu total: 2224.706574

Mote 0, radio total: 3617.270259

Mote 0, adc total: 0.000000

Mote 0, leds total: 2892.881511

Mote 0, sensor total: 0.000000

Mote 0, eeprom total: 0.000000

Mote 0, cpucycle total: 0.000000

Mote 0, Total energy: 8734.858344
```

APPENDIX II
Testing

2.1. Key Generation Testing

Results from 180 seconds of simulation with 2 mote sensors. This output shows the debug messages containing the packet data, the timer fired messages pushing data into the network, and the process of setting up the keys.

```
SIM: Random seed is 429878
SIM: Time for mote 0 initialized to 6139606.
SIM: Time for mote 1 initialized to 11929239.
0: Starting Node
1: Starting Node
1: TIMER Message: 0
1: Dump: 1 7 12321,46472,23066,38187,6976,6047,3617,15709,23461,38485
1: Dump2: 1 7 12321,46472,23066,38187,6976,6047,3617,15709,23461,38485
1: Dump: 1 100 7612,6759,5585,2749,0,0,0,0,0,0
0: Dump: 0 101 61560,0,0,0,0,0,0,0,0,0
1: Completed sending message
1: Dump: 1 102 11653,10846,9704,14980,12345,12345,12345,12345,12345,12345
0: Completed sending message
0: Dump: 0 103 53954,42243,19085,34196,12345,12345,12345,12345,12345,12345
1: Completed sending message
1: Dump: 1 104 43365,6759,5585,2749,0,0,0,0,0,0
0: Completed sending message
0: Dump: 0 105 2000,38202,31412,46509,0,0,0,0,0,0
1: Completed sending message
1: Mote Link 1<-->0 is complete.  Key: 2000
1: Dump: 1 7 12321,46472,23066,38187,6976,6047,3617,15709,23461,38485
0: Completed sending message
1: Completed sending message
1: TIMER Message: 4
1: Dump: 1 7 12321,8920,25775,59457,57748,62002,54646,39934,1774,11459
1: Completed sending message
1: TIMER Message: 3
1: Dump: 1 5 12321,37196,4994,1563,11561,31565,55173,40476,3362,15195
1: Completed sending message
1: TIMER Message: 2
1: Dump: 1 5 12321,11660,31239,54545,39732,1906,12283,32489,56525,34956
1: Completed sending message
1: TIMER Message: 1
1: Dump: 1 3 12321,58153,63300,57234,36406,11646,31715,55001,40100,2130
1: Completed sending message
1: TIMER Message: 0
1: Dump: 1 1 12321,42445,31372,54279,39192,826,10091,28617,65165,52236
1: Dump2: 1 1 12321,42445,31372,54279,39192,826,10091,28617,65165,52236
1: Dump: 1 100 43266,25366,59187,65392,12345,12345,12345,12345,12345,12345
0: Dump: 0 101 13050,42243,19085,34196,12345,12345,12345,12345,12345,12345
1: Completed sending message
1: Dump: 1 102 44754,25798,57571,63648,14313,14313,14313,14313,14313,14313
0: Completed sending message
0: Dump: 0 103 19754,35546,5427,15080,14313,14313,14313,14313,14313,14313
1: Completed sending message
```

```
1: Dump: 1 104 53584,25366,59187,65392,12345,12345,12345,12345,12345,12345
0: Completed sending message
0: Dump: 0 105 55858,36106,4835,15672,12345,12345,12345,12345,12345,12345
1: Completed sending message
1: Mote Link 1<-->0 is complete.  Key: 55858
1: Dump: 1 1 12321,42445,31372,54279,39192,826,10091,28617,65165,52236
0: Completed sending message
1: Completed sending message
1: TIMER Message: 4
1: Dump: 1 0 12321,36810,11910,31763,55609,33636,14290,20155,48233,18884
1: Completed sending message
1: TIMER Message: 3
1: Dump: 1 0 12321,35183,9160,26255,60417,59668,58162,63350,57342,36590
1: Completed sending message
1: TIMER Message: 2
1: Dump: 1 3 12321,53273,37156,4946,1979,11881,32205,55941,33820,14626
1: Completed sending message
1: TIMER Message: 1
1: Dump: 1 1 12321,15940,23959,39473,1396,11255,30449,52477,43244,24770
1: Completed sending message
1: TIMER Message: 0
1: Dump: 1 0 12321,53610,37830,5790,3107,14681,21421,34373,15772,23079
1: Dump2: 1 0 12321,53610,37830,5790,3107,14681,21421,34373,15772,23079
1: Dump: 1 100 38225,7092,5751,3569,14313,14313,14313,14313,14313,14313
0: Dump: 0 101 20288,35546,5427,15080,14313,14313,14313,14313,14313,14313
1: Completed sending message
1: Dump: 1 102 20323,49542,52293,55235,60891,60891,60891,60891,60891,60891
0: Completed sending message
0: Dump: 0 103 49684,33967,2533,836,60891,60891,60891,60891,60891,60891
1: Completed sending message
1: Dump: 1 104 43004,7092,5751,3569,14313,14313,14313,14313,14313,14313
0: Completed sending message
0: Dump: 0 105 15490,24221,54231,55670,14313,14313,14313,14313,14313,14313
1: Completed sending message
1: Mote Link 1<-->0 is complete.  Key: 15490
1: Dump: 1 0 12321,53610,37830,5790,3107,14681,21421,34373,15772,23079
0: Completed sending message
1: Completed sending message
1: TIMER Message: 4
1: Dump: 1 7 12321,57809,62132,54386,39414,766,9443,26841,61613,53324
1: Completed sending message
1: TIMER Message: 3
1: Dump: 1 6 12321,1331,11129,30701,52933,44188,26658,61787,54176,38506
1: Completed sending message
1: TIMER Message: 2
1: Dump: 1 3 12321,5177,2413,13253,18069,44085,27004,62439,55000,40122
1: Completed sending message
1: TIMER Message: 1
1: Dump: 1 3 12321,17081,42093,31180,53895,37912,6458,4971,1993,11917
1: Completed sending message
1: TIMER Message: 0
1: Dump: 1 5 12321,33596,14178,20443,48809,19524,47511,16952,42351,31688
1: Dump2: 1 5 12321,33596,14178,20443,48809,19524,47511,16952,42351,31688
1: Dump: 1 100 54927,39944,2330,13099,60891,60891,60891,60891,60891,60891
0: Dump: 0 101 55750,33967,2533,836,60891,60891,60891,60891,60891,60891
1: Completed sending message
1: Dump: 1 102 59917,41098,13720,4009,53593,53593,53593,53593,53593,53593
0: Completed sending message
0: Dump: 0 103 34739,4077,3924,3622,53593,53593,53593,53593,53593,53593
1: Completed sending message
1: Dump: 1 104 3762,39944,2330,13099,60891,60891,60891,60891,60891,60891
0: Completed sending message
0: Dump: 0 105 2314,13167,13270,12964,60891,60891,60891,60891,60891,60891
1: Completed sending message
1: Mote Link 1<-->0 is complete.  Key: 2314
1: Dump: 1 5 12321,33596,14178,20443,48809,19524,47511,16952,42351,31688
0: Completed sending message
1: Completed sending message
```

```
1: TIMER Message: 4
1: Dump: 1 0 12321,42810,32614,57299,36528,11402,30731,53513,37636,5906
1: Completed sending message
1: TIMER Message: 3
1: Dump: 1 1 12321,24525,40581,3100,14631,21329,34749,15980,24007,39569
1: Completed sending message
Simulation of 2 motes completed.
```

2.2. Network Communication Testing

Results from 30 seconds of simulation with 5 mote sensors.

```
SIM: Random seed is 163419
4: Sent Message: (Node,value) 3,2
3: Received Message: (Node,value) 4,2
3: Sent Message: (Node,value) 2,2
1: Sent Message: (Node,value) 0,7
4: Sent Message: (Node,value) 3,6
3: Received Message: (Node,value) 4,6
3: Sent Message: (Node,value) 2,6
2: Received Message: (Node,value) 3,6
2: Sent Message: (Node,value) 1,6
1: Received Message: (Node,value) 2,6
1: Sent Message: (Node,value) 0,6
4: Sent Message: (Node,value) 3,4
3: Received Message: (Node,value) 4,4
3: Sent Message: (Node,value) 2,4
2: Received Message: (Node,value) 3,4
2: Sent Message: (Node,value) 1,4
1: Received Message: (Node,value) 2,4
1: Sent Message: (Node,value) 0,4
0: Received Message: (Node,value) 1,4
4: Sent Message: (Node,value) 3,4
3: Received Message: (Node,value) 4,4
3: Sent Message: (Node,value) 2,4
2: Received Message: (Node,value) 3,4
2: Sent Message: (Node,value) 1,4
1: Received Message: (Node,value) 2,4
1: Sent Message: (Node,value) 0,4
0: Received Message: (Node,value) 1,4
4: Sent Message: (Node,value) 3,1
3: Received Message: (Node,value) 4,1
3: Sent Message: (Node,value) 2,1
2: Received Message: (Node,value) 3,1
2: Sent Message: (Node,value) 1,1
1: Received Message: (Node,value) 2,1
1: Sent Message: (Node,value) 0,1
0: Received Message: (Node,value) 1,1
4: Sent Message: (Node,value) 3,0
3: Received Message: (Node,value) 4,0
3: Sent Message: (Node,value) 2,0
2: Received Message: (Node,value) 3,0
2: Sent Message: (Node,value) 1,0
1: Received Message: (Node,value) 2,0
1: Sent Message: (Node,value) 0,0
0: Received Message: (Node,value) 1,0
4: Sent Message: (Node,value) 3,7
3: Received Message: (Node,value) 4,7
3: Sent Message: (Node,value) 2,7
2: Received Message: (Node,value) 3,7
2: Sent Message: (Node,value) 1,7
1: Received Message: (Node,value) 2,7
1: Sent Message: (Node,value) 0,7
0: Received Message: (Node,value) 1,7
```

```
4: Sent Message: (Node,value) 3,2
3: Received Message: (Node,value) 4,2
3: Sent Message: (Node,value) 2,2
2: Received Message: (Node,value) 3,2
2: Sent Message: (Node,value) 1,2
1: Received Message: (Node,value) 2,2
1: Sent Message: (Node,value) 0,2
0: Received Message: (Node,value) 1,2
4: Sent Message: (Node,value) 3,1
3: Received Message: (Node,value) 4,1
3: Sent Message: (Node,value) 2,1
2: Received Message: (Node,value) 3,1
2: Sent Message: (Node,value) 1,1
1: Received Message: (Node,value) 2,1
1: Sent Message: (Node,value) 0,1
0: Received Message: (Node,value) 1,1
Simulation of 5 motes completed.
```

2.3. Sample Tree Application Test

Results from 30 seconds of simulation with 15 mote sensors.

```
SIM: Random seed is 505942
SIM: Time for mote 0 initialized to 38308441.
SIM: Time for mote 1 initialized to 11063463.
SIM: Time for mote 2 initialized to 37427157.
SIM: Time for mote 3 initialized to 6269218.
SIM: Time for mote 4 initialized to 7424788.
SIM: Time for mote 5 initialized to 21263943.
SIM: Time for mote 6 initialized to 30998662.
SIM: Time for mote 7 initialized to 26766772.
SIM: Time for mote 8 initialized to 37219119.
SIM: Time for mote 9 initialized to 793060.
SIM: Time for mote 10 initialized to 9022949.
SIM: Time for mote 11 initialized to 366235.
SIM: Time for mote 12 initialized to 13551507.
SIM: Time for mote 13 initialized to 35265355.
SIM: Time for mote 14 initialized to 28527150.
11: Starting Node
11: Starting Timer
9: Starting Node
9: Starting Timer
3: Starting Node
3: Starting Timer
4: Starting Node
4: Starting Timer
10: Starting Node
10: Starting Timer
1: Starting Node
1: Starting Timer
12: Starting Node
12: Starting Timer
5: Starting Node
5: Starting Timer
7: Starting Node
7: Starting Timer
14: Starting Node
14: Starting Timer
6: Starting Node
6: Starting Timer
13: Starting Node
13: Starting Timer
8: Starting Node
8: Starting Timer
```

```
2: Starting Node
2: Starting Timer
0: Starting Node
0: Starting Timer
11: Sending Message to  5 val: 5
11: TIMER Message: 0
11: Dump: 11 0 5,17519,22162,29540,14464,44872,37077,61423,4507,64890
11: Dump2: 11 0 5,17519,22162,29540,14464,44872,37077,61423,4507,64890
11: Dump: 11 100 13489,46894,40985,36471,0,0,0,0,0,0
5: Dump: 5 101 44345,0,0,0,0,0,0,0,0,0
11: Completed sending message
11: Dump: 11 102 1160,34583,36896,48718,12345,12345,12345,12345,12345,12345
5: Completed sending message
5: Dump: 5 103 25228,8763,41813,45440,12345,12345,12345,12345,12345,12345
11: Completed sending message
9: Sending Message to  4 val: 1
9: TIMER Message: 0
9: Dump: 9 7 1,49772,10639,60992,29143,24304,162,48158,54635,1921
9: Dump2: 9 7 1,49772,10639,60992,29143,24304,162,48158,54635,1921
9: Dump: 9 100 45660,51695,16009,49228,0,0,0,0,0,0
11: Dump: 11 104 51100,46894,40985,36471,0,0,0,0,0,0
5: Completed sending message
4: Dump: 4 101 40513,0,0,0,0,0,0,0,0,0
9: Completed sending message
9: Dump: 9 102 33381,63958,3760,61557,12345,12345,12345,12345,12345,12345
4: Completed sending message
5: Dump: 5 105 57186,4610,37740,33209,0,0,0,0,0,0
11: Completed sending message
4: Dump: 4 103 8142,59702,5327,65332,12345,12345,12345,12345,12345,12345
9: Completed sending message
11: Mote Link 11<-->5 is complete.  Key: 57186
11: Dump: 11 0 5,17519,22162,29540,14464,44872,37077,61423,4507,64890
5: Completed sending message
9: Dump: 9 104 5702,51695,16009,49228,0,0,0,0,0,0
4: Completed sending message
5: Processing Data 11 0
5: Dump: 5 5 5,17519,22162,29540,14464,44872,37077,61423,4507,64890
5: Dump2: 5 5 5,17519,22162,29540,14464,44872,37077,61423,4507,64890
5: Dump: 5 100 29943,3789,64185,600,12345,12345,12345,12345,12345,12345
11: Completed sending message
2: Dump: 2 101 54702,0,0,0,0,0,0,0,0,0
5: Completed sending message
5: Dump: 5 102 17614,16116,51840,12897,0,0,0,0,0,0
2: Completed sending message
2: Dump: 2 103 39250,20797,53738,49229,12345,12345,12345,12345,12345,12345
5: Completed sending message
4: Dump: 4 105 42812,55567,9462,53005,0,0,0,0,0,0
9: Completed sending message
5: Dump: 5 104 31159,3789,64185,600,12345,12345,12345,12345,12345,12345
2: Completed sending message
9: Mote Link 9<-->4 is complete.  Key: 42812
9: Dump: 9 7 1,49772,10639,60992,29143,24304,162,48158,54635,1921
4: Completed sending message
2: Dump: 2 105 42403,24836,57811,61556,0,0,0,0,0,0
5: Completed sending message
4: Processing Data 9 7
4: Dump: 4 1 1,49772,10639,60992,29143,24304,162,48158,54635,1921
4: Dump2: 4 1 1,49772,10639,60992,29143,24304,162,48158,54635,1921
4: Dump: 4 100 55049,26805,1984,55594,12345,12345,12345,12345,12345,12345
9: Completed sending message
5: Mote Link 5<-->2 is complete.  Key: 42403
5: Dump: 5 5 5,17519,22162,29540,14464,44872,37077,61423,4507,64890
2: Completed sending message
1: Dump: 1 101 29956,0,0,0,0,0,0,0,0,0
4: Completed sending message
2: Processing Data 5 5
2: Dump: 2 5 5,17519,22162,29540,14464,44872,37077,61423,4507,64890
2: Dump2: 2 5 5,17519,22162,29540,14464,44872,37077,61423,4507,64890
```

```
2: Dump: 2 100 8742,14300,7208,19392,12345,12345,12345,12345,12345,12345
5: Completed sending message
4: Dump: 4 102 59184,22668,14329,59667,0,0,0,0,0,0
1: Completed sending message
0: Dump: 0 101 22049,0,0,0,0,0,0,0,0,0
2: Completed sending message
1: Dump: 1 103 46472,23066,38187,6976,12345,12345,12345,12345,12345,12345
4: Completed sending message
2: Dump: 2 102 4639,2021,11281,31737,0,0,0,0,0,0
0: Completed sending message
4: Dump: 4 104 20097,26805,1984,55594,12345,12345,12345,12345,12345,12345
1: Completed sending message
0: Dump: 0 103 53954,42243,19085,34196,12345,12345,12345,12345,12345,12345
2: Completed sending message
2: Dump: 2 104 43365,14300,7208,19392,12345,12345,12345,12345,12345,12345
0: Completed sending message
1: Dump: 1 105 3617,27171,42258,11129,0,0,0,0,0,0
4: Completed sending message
4: Mote Link 4<-->1 is complete.  Key: 3617
4: Dump: 4 1 1,49772,10639,60992,29143,24304,162,48158,54635,1921
1: Completed sending message
1: Processing Data 4 1
1: Dump: 1 1 1,49772,10639,60992,29143,24304,162,48158,54635,1921
1: Dump2: 1 1 1,49772,10639,60992,29143,24304,162,48158,54635,1921
1: Dump: 1 100 23461,38485,7612,6759,12345,12345,12345,12345,12345,12345
4: Completed sending message
0: Dump: 0 101 42028,38202,31412,46509,0,0,0,0,0,0
1: Completed sending message
0: Dump: 0 105 7982,42243,19085,34196,12345,12345,12345,12345,12345,12345
2: Completed sending message
1: Mote Link 1<-->0 is complete.  Key: 7982
1: Dump: 1 1 1,49772,10639,60992,29143,24304,162,48158,54635,1921
0: Completed sending message
0: Processing Data 1 1
1: Completed sending message
3: Sending Message to  1 val: 0
3: TIMER Message: 0
3: Dump: 3 0 0,46109,14862,13857,11903,7875,32699,48459,10402,4985
3: Dump2: 3 0 0,46109,14862,13857,11903,7875,32699,48459,10402,4985
3: Dump: 3 100 25807,35747,17778,51417,0,0,0,0,0,0
1: Dump: 1 101 47081,42604,11653,10846,0,0,0,0,0,0
3: Completed sending message
3: Dump: 3 102 21750,48026,30027,63712,12345,12345,12345,12345,12345,12345
1: Completed sending message
1: Dump: 1 103 18901,45749,21628,35303,12345,12345,12345,12345,12345,12345
3: Completed sending message
3: Dump: 3 104 49588,35747,17778,51417,0,0,0,0,0,0
1: Completed sending message
1: Dump: 1 105 25775,33420,25669,47582,0,0,0,0,0,0
3: Completed sending message
3: Mote Link 3<-->1 is complete.  Key: 25775
3: Dump: 3 0 0,46109,14862,13857,11903,7875,32699,48459,10402,4985
1: Completed sending message
1: Processing Data 3 0
1: Dump: 1 1 0,46109,14862,13857,11903,7875,32699,48459,10402,4985
3: Completed sending message
0: Processing Data 1 1
1: Completed sending message
10: Sending Message to  4 val: 1
10: TIMER Message: 0
10: Dump: 10 0 1,39295,52518,26004,9465,42543,45958,39124,52848,25400
10: Dump2: 10 0 1,39295,52518,26004,9465,42543,45958,39124,52848,25400
10: Dump: 10 100 10657,48287,34534,61972,0,0,0,0,0,0
4: Dump: 4 101 8299,22668,14329,59667,0,0,0,0,0,0
10: Completed sending message
10: Dump: 10 102 6552,36006,46815,49709,12345,12345,12345,12345,12345,12345
4: Completed sending message
4: Dump: 4 103 63072,10851,33388,49787,12345,12345,12345,12345,12345,12345
```

```
10: Completed sending message
10: Dump: 10 104 6738,48287,34534,61972,0,0,0,0,0,0
4: Completed sending message
4: Dump: 4 105 21048,6746,45653,62018,0,0,0,0,0,0
10: Completed sending message
10: Mote Link 10<-->4 is complete.  Key: 21048
10: Dump: 10 0 1,39295,52518,26004,9465,42543,45958,39124,52848,25400
4: Completed sending message
4: Processing Data 10 0
4: Dump: 4 1 1,39295,52518,26004,9465,42543,45958,39124,52848,25400
10: Completed sending message
1: Processing Data 4 1
1: Dump: 1 1 1,39295,52518,26004,9465,42543,45958,39124,52848,25400
4: Completed sending message
0: Processing Data 1 1
1: Completed sending message
12: Sending Message to  5 val: 1
12: TIMER Message: 0
12: Dump: 12 3 1,53536,63490,43590,3790,22487,58857,37264,31074,47247
12: Dump2: 12 3 1,53536,63490,43590,3790,22487,58857,37264,31074,47247
12: Dump: 12 100 11100,7411,29617,44333,0,0,0,0,0,0
5: Dump: 5 101 22776,16116,51840,12897,0,0,0,0,0,0
12: Completed sending message
12: Dump: 12 102 7013,11466,17288,40212,12345,12345,12345,12345,12345,12345
5: Completed sending message
5: Dump: 5 103 60479,12116,47499,33852,12345,12345,12345,12345,12345,12345
12: Completed sending message
12: Dump: 12 104 19124,7411,29617,44333,0,0,0,0,0,0
5: Completed sending message
5: Dump: 5 105 2482,8045,35250,46085,0,0,0,0,0,0
12: Completed sending message
12: Mote Link 12<-->5 is complete.  Key: 2482
12: Dump: 12 3 1,53536,63490,43590,3790,22487,58857,37264,31074,47247
5: Completed sending message
5: Processing Data 12 3
5: Dump: 5 5 1,53536,63490,43590,3790,22487,58857,37264,31074,47247
12: Completed sending message
2: Processing Data 5 5
2: Dump: 2 1 1,53536,63490,43590,3790,22487,58857,37264,31074,47247
2: Dump2: 2 1 1,53536,63490,43590,3790,22487,58857,37264,31074,47247
2: Dump: 2 100 64968,38921,21383,54430,0,0,0,0,0,0
5: Completed sending message
0: Dump: 0 101 15557,38202,31412,46509,0,0,0,0,0,0
2: Completed sending message
0: Completed sending message
7: Sending Message to  3 val: 1
7: TIMER Message: 0
7: Dump: 7 4 1,63570,25896,20009,6187,46127,64558,27680,23609,15371
7: Dump2: 7 4 1,63570,25896,20009,6187,46127,64558,27680,23609,15371
7: Dump: 7 100 64623,27822,23845,15923,0,0,0,0,0,0
3: Dump: 3 101 54402,48026,30027,63712,12345,12345,12345,12345,12345,12345
7: Completed sending message
7: Dump: 7 102 52310,23703,27932,3594,12345,12345,12345,12345,12345,12345
3: Completed sending message
3: Dump: 3 103 27997,39047,25402,33865,0,0,0,0,0,0
7: Completed sending message
7: Dump: 7 104 49604,27822,23845,15923,0,0,0,0,0,0
3: Completed sending message
3: Dump: 3 105 63345,43198,21251,46192,12345,12345,12345,12345,12345,12345
7: Completed sending message
7: Mote Link 7<-->3 is complete.  Key: 63345
7: Dump: 7 4 1,63570,25896,20009,6187,46127,64558,27680,23609,15371
3: Completed sending message
3: Processing Data 7 4
3: Dump: 3 1 1,63570,25896,20009,6187,46127,64558,27680,23609,15371
7: Completed sending message
1: Processing Data 3 1
1: Dump: 1 1 1,63570,25896,20009,6187,46127,64558,27680,23609,15371
```

```
3: Completed sending message
0: Processing Data 1 1
1: Completed sending message
14: Sending Message to  6 val: 4
14: TIMER Message: 0
14: Dump: 14 4 4,5437,803,12055,30591,51119,46594,21848,33765,16022
14: Dump2: 14 4 4,5437,803,12055,30591,51119,46594,21848,33765,16022
14: Dump: 14 100 21625,33211,14890,23809,0,0,0,0,0,0
6: Dump: 6 101 17920,0,0,0,0,0,0,0,0,0
14: Completed sending message
14: Dump: 14 102 25664,45442,2579,27960,12345,12345,12345,12345,12345,12345
6: Completed sending message
6: Dump: 6 103 16494,13624,57236,6853,12345,12345,12345,12345,12345,12345
14: Completed sending message
14: Dump: 14 104 51786,33211,14890,23809,0,0,0,0,0,0
6: Completed sending message
6: Dump: 6 105 42289,1281,61357,11004,0,0,0,0,0,0
14: Completed sending message
14: Mote Link 14<-->6 is complete.  Key: 42289
14: Dump: 14 4 4,5437,803,12055,30591,51119,46594,21848,33765,16022
6: Completed sending message
6: Processing Data 14 4
6: Dump: 6 4 4,5437,803,12055,30591,51119,46594,21848,33765,16022
6: Dump2: 6 4 4,5437,803,12055,30591,51119,46594,21848,33765,16022
6: Dump: 6 100 31439,16506,13584,57284,12345,12345,12345,12345,12345,12345
14: Completed sending message
2: Dump: 2 101 57912,43056,25534,58535,12345,12345,12345,12345,12345,12345
6: Completed sending message
6: Dump: 6 102 19190,28739,1321,61437,0,0,0,0,0,0
2: Completed sending message
2: Dump: 2 103 36419,32539,36262,30933,0,0,0,0,0,0
6: Completed sending message
6: Dump: 6 104 27346,16506,13584,57284,12345,12345,12345,12345,12345,12345
2: Completed sending message
2: Dump: 2 105 26605,20258,48543,18668,12345,12345,12345,12345,12345,12345
6: Completed sending message
6: Mote Link 6<-->2 is complete.  Key: 26605
6: Dump: 6 4 4,5437,803,12055,30591,51119,46594,21848,33765,16022
2: Completed sending message
2: Processing Data 6 4
2: Dump: 2 4 4,5437,803,12055,30591,51119,46594,21848,33765,16022
2: Dump2: 2 4 4,5437,803,12055,30591,51119,46594,21848,33765,16022
2: Dump: 2 100 6925,17802,63620,37521,0,0,0,0,0,0
6: Completed sending message
0: Dump: 0 101 62866,35348,26010,43651,7982,7982,7982,7982,7982,7982
2: Completed sending message
0: Completed sending message
13: Sending Message to  6 val: 5
13: TIMER Message: 0
13: Dump: 13 3 5,59953,49065,5273,21232,56878,55191,50405,57857,45001
13: Dump2: 13 3 5,59953,49065,5273,21232,56878,55191,50405,57857,45001
13: Dump: 13 100 13401,4976,23854,49562,0,0,0,0,0,0
6: Dump: 6 101 22878,28739,1321,61437,0,0,0,0,0,0
13: Completed sending message
13: Dump: 13 102 1120,9033,27927,61859,12345,12345,12345,12345,12345,12345
6: Completed sending message
6: Dump: 6 103 21626,7440,36804,47717,12345,12345,12345,12345,12345,12345
13: Completed sending message
13: Dump: 13 104 36654,4976,23854,49562,0,0,0,0,0,0
6: Completed sending message
6: Dump: 6 105 1983,11561,49149,35420,0,0,0,0,0,0
13: Completed sending message
13: Mote Link 13<-->6 is complete.  Key: 1983
13: Dump: 13 3 5,59953,49065,5273,21232,56878,55191,50405,57857,45001
6: Completed sending message
6: Processing Data 13 3
6: Dump: 6 5 5,59953,49065,5273,21232,56878,55191,50405,57857,45001
13: Completed sending message
```

```
2: Processing Data 6 5
2: Dump: 2 5 5,59953,49065,5273,21232,56878,55191,50405,57857,45001
2: Dump2: 2 5 5,59953,49065,5273,21232,56878,55191,50405,57857,45001
2: Dump: 2 100 65278,40549,24431,52558,12345,12345,12345,12345,12345,12345
6: Completed sending message
0: Dump: 0 101 35801,38202,31412,46509,0,0,0,0,0,0
2: Completed sending message
0: Completed sending message
8: Sending Message to  3 val: 5
8: TIMER Message: 0
8: Dump: 8 7 5,29857,26041,18313,1001,35625,35492,35262,36746,33762
8: Dump2: 8 7 5,29857,26041,18313,1001,35625,35492,35262,36746,33762
8: Dump: 8 100 39730,43666,51666,3922,0,0,0,0,0,0
3: Dump: 3 101 56874,39047,25402,33865,0,0,0,0,0,0
8: Completed sending message
8: Dump: 8 102 43787,39595,63979,16235,12345,12345,12345,12345,12345,12345
3: Completed sending message
3: Dump: 3 103 26711,37523,30482,44057,12345,12345,12345,12345,12345,12345
8: Completed sending message
8: Dump: 8 104 58230,43666,51666,3922,0,0,0,0,0,0
3: Completed sending message
3: Dump: 3 105 22065,41642,18219,39968,0,0,0,0,0,0
8: Completed sending message
8: Mote Link 8<-->3 is complete.  Key: 22065
8: Dump: 8 7 5,29857,26041,18313,1001,35625,35492,35262,36746,33762
3: Completed sending message
3: Processing Data 8 7
3: Dump: 3 5 5,29857,26041,18313,1001,35625,35492,35262,36746,33762
8: Completed sending message
1: Processing Data 3 5
1: Dump: 1 5 5,29857,26041,18313,1001,35625,35492,35262,36746,33762
3: Completed sending message
0: Processing Data 1 5
1: Completed sending message
11: Sending Message to  5 val: 4
11: TIMER Message: 4
11: Dump: 11 6 4,33008,53157,20751,31826,9956,37760,59717,7375,59346
5: Processing Data 11 6
5: Dump: 5 4 4,33008,53157,20751,31826,9956,37760,59717,7375,59346
11: Completed sending message
2: Processing Data 5 4
2: Dump: 2 5 4,33008,53157,20751,31826,9956,37760,59717,7375,59346
2: Dump2: 2 5 4,33008,53157,20751,31826,9956,37760,59717,7375,59346
2: Dump: 2 100 37295,16579,61974,34741,0,0,0,0,0,0
5: Completed sending message
0: Dump: 0 101 27615,35348,26010,43651,7982,7982,7982,7982,7982,7982
2: Completed sending message
9: Sending Message to  4 val: 3
9: TIMER Message: 4
9: Dump: 9 6 3,45672,51591,15961,49644,11919,57408,28119,26352,28834
0: Completed sending message
4: Processing Data 9 6
4: Dump: 4 3 3,45672,51591,15961,49644,11919,57408,28119,26352,28834
9: Completed sending message
1: Processing Data 4 3
1: Dump: 1 5 3,45672,51591,15961,49644,11919,57408,28119,26352,28834
1: Dump2: 1 5 3,45672,51591,15961,49644,11919,57408,28119,26352,28834
1: Dump: 1 100 1774,11459,30873,53293,12345,12345,12345,12345,12345,12345
4: Completed sending message
0: Dump: 0 101 31456,38202,31412,46509,0,0,0,0,0,0
1: Completed sending message
0: Completed sending message
10: Sending Message to  4 val: 3
10: TIMER Message: 4
10: Dump: 10 7 3,6766,56077,18882,31829,5991,49435,32238,5133,50999
4: Processing Data 10 7
4: Dump: 4 3 3,6766,56077,18882,31829,5991,49435,32238,5133,50999
10: Completed sending message
```

```
1: Processing Data 4 3
1: Dump: 1 5 3,6766,56077,18882,31829,5991,49435,32238,5133,50999
1: Dump2: 1 5 3,6766,56077,18882,31829,5991,49435,32238,5133,50999
1: Dump: 1 100 4994,1563,11561,31565,12055,12055,12055,12055,12055,12055
4: Completed sending message
0: Dump: 0 101 53918,42243,19085,34196,12345,12345,12345,12345,12345,12345
1: Completed sending message
0: Completed sending message
12: Sending Message to  5 val: 6
12: TIMER Message: 4
12: Dump: 12 2 6,41415,6604,31187,47601,10656,6411,30817,47757,12120
5: Processing Data 12 2
5: Dump: 5 6 6,41415,6604,31187,47601,10656,6411,30817,47757,12120
12: Completed sending message
2: Processing Data 5 6
2: Dump: 2 6 6,41415,6604,31187,47601,10656,6411,30817,47757,12120
2: Dump2: 2 6 6,41415,6604,31187,47601,10656,6411,30817,47757,12120
2: Dump: 2 100 43918,13445,6810,18084,12345,12345,12345,12345,12345,12345
5: Completed sending message
0: Dump: 0 101 55757,38202,31412,46509,0,0,0,0,0,0
2: Completed sending message
0: Completed sending message
7: Sending Message to  3 val: 6
7: TIMER Message: 4
7: Dump: 7 7 6,16608,1465,36619,35430,32944,38164,48732,59596,17900
3: Processing Data 7 7
3: Dump: 3 6 6,16608,1465,36619,35430,32944,38164,48732,59596,17900
7: Completed sending message
1: Processing Data 3 6
1: Dump: 1 6 6,16608,1465,36619,35430,32944,38164,48732,59596,17900
1: Dump2: 1 6 6,16608,1465,36619,35430,32944,38164,48732,59596,17900
1: Dump: 1 100 40476,3362,15195,22441,12345,12345,12345,12345,12345,12345
3: Completed sending message
0: Dump: 0 101 16126,35348,26010,43651,7982,7982,7982,7982,7982,7982
1: Completed sending message
0: Completed sending message
14: Sending Message to  6 val: 7
14: TIMER Message: 4
14: Dump: 14 7 7,61935,55938,35928,8684,27277,64579,49626,47848,19596
6: Processing Data 14 7
6: Dump: 6 7 7,61935,55938,35928,8684,27277,64579,49626,47848,19596
14: Completed sending message
2: Processing Data 6 7
2: Dump: 2 7 7,61935,55938,35928,8684,27277,64579,49626,47848,19596
2: Dump2: 2 7 7,61935,55938,35928,8684,27277,64579,49626,47848,19596
2: Dump: 2 100 40489,24519,52254,64421,0,0,0,0,0,0
6: Completed sending message
0: Dump: 0 101 54822,47661,21923,39610,12055,12055,12055,12055,12055,12055
2: Completed sending message
0: Completed sending message
13: Sending Message to  6 val: 7
13: TIMER Message: 4
13: Dump: 13 6 7,30565,38152,16859,63604,39715,23949,49368,60027,48957
6: Processing Data 13 6
6: Dump: 6 7 7,30565,38152,16859,63604,39715,23949,49368,60027,48957
13: Completed sending message
2: Processing Data 6 7
2: Dump: 2 7 7,30565,38152,16859,63604,39715,23949,49368,60027,48957
2: Dump2: 2 7 7,30565,38152,16859,63604,39715,23949,49368,60027,48957
2: Dump: 2 100 19011,59158,44469,14351,12345,12345,12345,12345,12345,12345
6: Completed sending message
0: Dump: 0 101 24799,42243,19085,34196,12345,12345,12345,12345,12345,12345
2: Completed sending message
0: Completed sending message
8: Sending Message to  3 val: 5
8: TIMER Message: 4
8: Dump: 8 3 5,38460,45198,65002,26402,17083,2437,40945,41748,56030
3: Processing Data 8 3
```

```
3: Dump: 3 6 5,38460,45198,65002,26402,17083,2437,40945,41748,56030
8: Completed sending message
1: Processing Data 3 6
1: Dump: 1 5 5,38460,45198,65002,26402,17083,2437,40945,41748,56030
1: Dump2: 1 5 5,38460,45198,65002,26402,17083,2437,40945,41748,56030
1: Dump: 1 100 11660,31239,54545,39732,12055,12055,12055,12055,12055,12055
3: Completed sending message
0: Dump: 0 101 9540,47661,21923,39610,12055,12055,12055,12055,12055,12055
1: Completed sending message
0: Completed sending message
11: Sending Message to  5 val: 1
11: TIMER Message: 3
11: Dump: 11 6 1,13566,47020,41245,35967,54971,25395,6186,60948,4717
5: Processing Data 11 6
5: Dump: 5 6 1,13566,47020,41245,35967,54971,25395,6186,60948,4717
11: Completed sending message
2: Processing Data 5 6
2: Dump: 2 7 1,13566,47020,41245,35967,54971,25395,6186,60948,4717
2: Dump2: 2 7 1,13566,47020,41245,35967,54971,25395,6186,60948,4717
2: Dump: 2 100 29836,39560,22153,56962,0,0,0,0,0,0
5: Completed sending message
0: Dump: 0 101 25712,35348,26010,43651,7982,7982,7982,7982,7982,7982
2: Completed sending message
9: Sending Message to  4 val: 6
9: TIMER Message: 3
9: Dump: 9 6 6,49787,10657,60956,29039,24448,578,47582,57067,4225
0: Completed sending message
4: Processing Data 9 6
4: Dump: 4 6 6,49787,10657,60956,29039,24448,578,47582,57067,4225
9: Completed sending message
1: Processing Data 4 6
1: Dump: 1 6 6,49787,10657,60956,29039,24448,578,47582,57067,4225
1: Dump2: 1 6 6,49787,10657,60956,29039,24448,578,47582,57067,4225
1: Dump: 1 100 12283,32489,56525,34956,12345,12345,12345,12345,12345,12345
4: Completed sending message
0: Dump: 0 101 64437,38202,31412,46509,0,0,0,0,0,0
1: Completed sending message
0: Completed sending message
Simulation of 15 motes completed.
```

2.4. Base Tree Power Profile

Results from 180 seconds of simulation with 15 mote sensors

```
maxseen 14
Mote 0, cpu total: 2218.976819
Mote 0, radio total: 3652.104705
Mote 0, adc total: 0.000000
Mote 0, leds total: 2898.072327
Mote 0, sensor total: 0.000000
Mote 0, eeprom total: 0.000000
Mote 0, cpu_cycle total: 0.000000
Mote 0, Total energy: 8769.153851

Mote 1, cpu total: 2218.976819
Mote 1, radio total: 3626.642352
Mote 1, adc total: 0.000000
Mote 1, leds total: 2498.434462
Mote 1, sensor total: 0.000000
Mote 1, eeprom total: 0.000000
Mote 1, cpu_cycle total: 0.000000
Mote 1, Total energy: 8344.053632

Mote 2, cpu total: 2218.976819
Mote 2, radio total: 3609.511452
Mote 2, adc total: 0.000000
Mote 2, leds total: 2548.352690
Mote 2, sensor total: 0.000000
Mote 2, eeprom total: 0.000000
Mote 2, cpu_cycle total: 0.000000
Mote 2, Total energy: 8376.840962

Mote 3, cpu total: 2218.976819
Mote 3, radio total: 3629.881890
Mote 3, adc total: 0.000000
Mote 3, leds total: 1973.682531
Mote 3, sensor total: 0.000000
Mote 3, eeprom total: 0.000000
Mote 3, cpu_cycle total: 0.000000
Mote 3, Total energy: 7822.541240

Mote 4, cpu total: 2218.976819
Mote 4, radio total: 3715.867709
Mote 4, adc total: 0.000000
Mote 4, leds total: 2226.251559
Mote 4, sensor total: 0.000000
Mote 4, eeprom total: 0.000000
Mote 4, cpu_cycle total: 0.000000
Mote 4, Total energy: 8161.096087

Mote 5, cpu total: 2218.976819
Mote 5, radio total: 3634.872872
Mote 5, adc total: 0.000000
Mote 5, leds total: 2190.444029
Mote 5, sensor total: 0.000000
Mote 5, eeprom total: 0.000000
Mote 5, cpu_cycle total: 0.000000
Mote 5, Total energy: 8044.293721

Mote 6, cpu total: 2218.976819
Mote 6, radio total: 3761.738920
Mote 6, adc total: 0.000000
Mote 6, leds total: 2174.156711
Mote 6, sensor total: 0.000000
Mote 6, eeprom total: 0.000000
Mote 6, cpu_cycle total: 0.000000
Mote 6, Total energy: 8154.872450

Mote 7, cpu total: 2218.976819
Mote 7, radio total: 3672.689758
Mote 7, adc total: 0.000000
Mote 7, leds total: 1578.147064
Mote 7, sensor total: 0.000000
Mote 7, eeprom total: 0.000000
Mote 7, cpu_cycle total: 0.000000
Mote 7, Total energy: 7469.813641

Mote 8, cpu total: 2218.976819
Mote 8, radio total: 3785.418183
Mote 8, adc total: 0.000000
Mote 8, leds total: 1532.638110
Mote 8, sensor total: 0.000000
Mote 8, eeprom total: 0.000000
Mote 8, cpu_cycle total: 0.000000
Mote 8, Total energy: 7537.033112

Mote 9, cpu total: 2202.844661
Mote 9, radio total: 3757.922072
Mote 9, adc total: 0.000000
Mote 9, leds total: 1618.119804
Mote 9, sensor total: 0.000000
Mote 9, eeprom total: 0.000000
Mote 9, cpu_cycle total: 0.000000
Mote 9, Total energy: 7578.886537

Mote 10, cpu total: 2150.746170
Mote 10, radio total: 3616.119656
Mote 10, adc total: 0.000000
Mote 10, leds total: 1774.790384
Mote 10, sensor total: 0.000000
Mote 10, eeprom total: 0.000000
Mote 10, cpu_cycle total: 0.000000
Mote 10, Total energy: 7541.656211

Mote 11, cpu total: 2150.746170
Mote 11, radio total: 3594.948287
Mote 11, adc total: 0.000000
Mote 11, leds total: 1722.899798
Mote 11, sensor total: 0.000000
Mote 11, eeprom total: 0.000000
Mote 11, cpu_cycle total: 0.000000
Mote 11, Total energy: 7468.594255

Mote 12, cpu total: 2150.746170
Mote 12, radio total: 3665.030092
Mote 12, adc total: 0.000000
Mote 12, leds total: 1940.712262
Mote 12, sensor total: 0.000000
Mote 12, eeprom total: 0.000000
Mote 12, cpu_cycle total: 0.000000
Mote 12, Total energy: 7756.488524

Mote 13, cpu total: 2150.746170
Mote 13, radio total: 3654.548687
Mote 13, adc total: 0.000000
Mote 13, leds total: 1605.464870
Mote 13, sensor total: 0.000000
Mote 13, eeprom total: 0.000000
Mote 13, cpu_cycle total: 0.000000
Mote 13, Total energy: 7410.759727

Mote 14, cpu total: 2150.746170
Mote 14, radio total: 3669.187537
Mote 14, adc total: 0.000000
Mote 14, leds total: 1648.371681
Mote 14, sensor total: 0.000000
Mote 14, eeprom total: 0.000000
Mote 14, cpu_cycle total: 0.000000
Mote 14, Total energy: 7468.305388
```

2.5. New Tree Power Profile

Results from 180 seconds of simulation with 15 mote sensors

```
maxseen 14
Mote 0, cpu total: 2203.959851
Mote 0, radio total: 3696.191106
Mote 0, adc total: 0.000000
Mote 0, leds total: 2222.547375
Mote 0, sensor total: 0.000000
Mote 0, eeprom total: 0.000000
Mote 0, cpu_cycle total: 0.000000
Mote 0, Total energy: 8122.698332

Mote 1, cpu total: 2203.959851
Mote 1, radio total: 3732.661747
Mote 1, adc total: 0.000000
Mote 1, leds total: 2052.241901
Mote 1, sensor total: 0.000000
Mote 1, eeprom total: 0.000000
Mote 1, cpu_cycle total: 0.000000
Mote 1, Total energy: 7988.863498

Mote 2, cpu total: 2203.959851
Mote 2, radio total: 3700.232011
Mote 2, adc total: 0.000000
Mote 2, leds total: 2269.787883
Mote 2, sensor total: 0.000000
Mote 2, eeprom total: 0.000000
Mote 2, cpu_cycle total: 0.000000
Mote 2, Total energy: 8173.979745

Mote 3, cpu total: 2203.959851
Mote 3, radio total: 3762.596357
Mote 3, adc total: 0.000000
Mote 3, leds total: 1986.239145
Mote 3, sensor total: 0.000000
Mote 3, eeprom total: 0.000000
Mote 3, cpu_cycle total: 0.000000
Mote 3, Total energy: 7952.795352

Mote 4, cpu total: 2203.943069
Mote 4, radio total: 3690.571027
Mote 4, adc total: 0.000000
Mote 4, leds total: 2202.636157
Mote 4, sensor total: 0.000000
Mote 4, eeprom total: 0.000000
Mote 4, cpu_cycle total: 0.000000
Mote 4, Total energy: 8097.150252

Mote 5, cpu total: 2203.943069
Mote 5, radio total: 3671.221505
Mote 5, adc total: 0.000000
Mote 5, leds total: 2089.842176
Mote 5, sensor total: 0.000000
Mote 5, eeprom total: 0.000000
Mote 5, cpu_cycle total: 0.000000
Mote 5, Total energy: 7965.006750

Mote 6, cpu total: 2203.943069
Mote 6, radio total: 3762.733678
Mote 6, adc total: 0.000000
Mote 6, leds total: 2468.257579
Mote 6, sensor total: 0.000000
Mote 6, eeprom total: 0.000000
Mote 6, cpu_cycle total: 0.000000
Mote 6, Total energy: 8434.934325

Mote 7, cpu total: 2197.774589
Mote 7, radio total: 3668.898956
Mote 7, adc total: 0.000000
Mote 7, leds total: 1274.993334
Mote 7, sensor total: 0.000000
Mote 7, eeprom total: 0.000000
Mote 7, cpu_cycle total: 0.000000
Mote 7, Total energy: 7141.666880

Mote 8, cpu total: 2197.774589
Mote 8, radio total: 3723.208888
Mote 8, adc total: 0.000000
Mote 8, leds total: 1749.073120
Mote 8, sensor total: 0.000000
Mote 8, eeprom total: 0.000000
Mote 8, cpu_cycle total: 0.000000
Mote 8, Total energy: 7670.056597

Mote 9, cpu total: 2197.774589
Mote 9, radio total: 3720.864808
Mote 9, adc total: 0.000000
Mote 9, leds total: 1551.275613
Mote 9, sensor total: 0.000000
Mote 9, eeprom total: 0.000000
Mote 9, cpu_cycle total: 0.000000
Mote 9, Total energy: 7469.915010

Mote 10, cpu total: 2197.774589
Mote 10, radio total: 3632.346702
Mote 10, adc total: 0.000000
Mote 10, leds total: 1714.697507
Mote 10, sensor total: 0.000000
Mote 10, eeprom total: 0.000000
Mote 10, cpu_cycle total: 0.000000
Mote 10, Total energy: 7544.818798

Mote 11, cpu total: 2197.774589
Mote 11, radio total: 3631.745911
Mote 11, adc total: 0.000000
Mote 11, leds total: 1663.282477
Mote 11, sensor total: 0.000000
Mote 11, eeprom total: 0.000000
Mote 11, cpu_cycle total: 0.000000
Mote 11, Total energy: 7492.802977

Mote 12, cpu total: 2197.774589
Mote 12, radio total: 3575.857780
Mote 12, adc total: 0.000000
Mote 12, leds total: 1568.121608
Mote 12, sensor total: 0.000000
Mote 12, eeprom total: 0.000000
Mote 12, cpu_cycle total: 0.000000
Mote 12, Total energy: 7341.753977

Mote 13, cpu total: 2197.774589
Mote 13, radio total: 3730.243473
Mote 13, adc total: 0.000000
Mote 13, leds total: 1943.839616
Mote 13, sensor total: 0.000000
Mote 13, eeprom total: 0.000000
Mote 13, cpu_cycle total: 0.000000
Mote 13, Total energy: 7871.857679

Mote 14, cpu total: 2197.774589
Mote 14, radio total: 3747.671901
Mote 14, adc total: 0.000000
Mote 14, leds total: 1954.747879
Mote 14, sensor total: 0.000000
Mote 14, eeprom total: 0.000000
Mote 14, cpu_cycle total: 0.000000
Mote 14, Total energy: 7900.194369
```

APPENDIX III
Source Code

3.1. Network Test Application

3.1.1. NetTest.nc

```
includes IntMsg;

configuration NetTest {
}

implementation {
        components Main, TimerC, RandomLFSR, GenericComm, IntToLeds,  NetTestM;

        Main.StdControl -> TimerC.StdControl;
        Main.StdControl -> NetTestM.StdControl;
        Main.StdControl -> GenericComm.Control;
        Main.StdControl -> IntToLeds.StdControl;

        NetTestM.IntOutput -> IntToLeds.IntOutput;
        NetTestM.SendMsg -> GenericComm.SendMsg[AM_INTMSG];
        NetTestM.ReceiveMsg -> GenericComm.ReceiveMsg[AM_INTMSG];

        NetTestM.Random -> RandomLFSR.Random;
         NetTestM.Timer -> TimerC.Timer[unique("Timer")];
}
```

3.1.2. NetTestM.nc

```
includes IntMsg;

module NetTestM {

        provides {
                 interface StdControl;
}

        uses {
                interface SendMsg;
                interface ReceiveMsg;
                interface IntOutput;
                interface Random;
                interface Timer;
        }
}

implementation {

struct TOS_Msg data;
uint16_t receiveFlag;
uint16_t leftNode;
uint16_t rightNode;

//// Interface /////
////////////////////
command result_t StdControl.init() {
        receiveFlag = 0;
        return SUCCESS;
}

command result_t StdControl.start() {
        return call Timer.start(TIMER_REPEAT, 3000);
}

command result_t StdControl.stop() {
        return call Timer.stop();
}

////////////////////

event result_t IntOutput.outputComplete(result_t success) {
        return success;
}

event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr m) {
        IntMsg *message = (IntMsg *)m->data;
        IntMsg *message2 = (IntMsg *)data.data;
        uint16_t dest;
        uint16_t maxVal;
        receiveFlag = 1;
        dbg(DBG_USR1, "Received Message: (Node,value) %d,%d\n",message->src,message->val);

        if(TOS_LOCAL_ADDRESS !=0)
        {
                message2->val = message->val;
                message2->src = TOS_LOCAL_ADDRESS;
                dest = TOS_LOCAL_ADDRESS-1;
                dbg(DBG_USR1, "Sent Message: (Node,value) %d,%d\n",message2->src,message2->val);
                call SendMsg.send(dest, sizeof(IntMsg), &data);
        }

        call IntOutput.output(message->val); //Put data to LEDS
        return m;
}

event result_t SendMsg.sendDone(TOS_MsgPtr msg, result_t success) {
        return SUCCESS;
}

event result_t Timer.fired() {
        uint16_t dest;
        uint16_t randomvalue;
        IntMsg *message = (IntMsg *)data.data;

        if(receiveFlag == 0)
        {
                randomvalue  = call Random.rand() % 8; //Random value from 0-7
                message->val = randomvalue;
                message->src = TOS_LOCAL_ADDRESS;
                dest = TOS_LOCAL_ADDRESS-1;

                call IntOutput.output(randomvalue);
                dbg(DBG_USR1, "Sent Message: (Node,value) %d,%d\n",message->src,message->val);

                call SendMsg.send(dest, sizeof(IntMsg), &data);
        }
        return SUCCESS;
}
}
```

3.2. Basic Tree Application

3.2.1. Tree.nc

```
includes IntMsg;

configuration Tree {
}

implementation {
        components Main, TimerC, RandomLFSR, GenericComm, IntToLeds,  TreeM;

        Main.StdControl -> TimerC.StdControl;
        Main.StdControl -> TreeM.StdControl;
        Main.StdControl -> GenericComm.Control;
        Main.StdControl -> IntToLeds.StdControl;

        TreeM.IntOutput -> IntToLeds.IntOutput;

        TreeM.SendMsg -> GenericComm.SendMsg[AM_INTMSG];
        TreeM.ReceiveMsg -> GenericComm.ReceiveMsg[AM_INTMSG];

        TreeM.Random -> RandomLFSR.Random;
        TreeM.Timer -> TimerC.Timer[unique("Timer")];
}
```

3.2.2. TreeM.nc

```
includes IntMsg;

module TreeM {
        provides {
                interface StdControl;
        }
        uses {
                interface SendMsg;
                interface ReceiveMsg;
                interface IntOutput;
                interface Random;
                interface Timer;
        }
}

implementation {

struct TOS_Msg data;
uint16_t receiveFlag;
uint16_t leftNode;
uint16_t rightNode;

//// Interface /////
////////////////////
command result_t StdControl.init() {
        receiveFlag = 0;
        return SUCCESS;
}

command result_t StdControl.start() {
        return call Timer.start(TIMER_REPEAT, 3000);
}

command result_t StdControl.stop() {
        return call Timer.stop();
}

////////////////////
event result_t IntOutput.outputComplete(result_t success) {
        dbg(DBG_USR1, "Output Complete %d\n", success);
        return success;
}

event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr m) {
        IntMsg *message = (IntMsg *)m->data;
        IntMsg *message2 = (IntMsg *)data.data;
        uint16_t dest;
        uint16_t maxVal;
        receiveFlag = 1;
        dbg(DBG_USR1, "Received Message: (Node,value) %d,%d\n",message->src,message->val);

        if((TOS_LOCAL_ADDRESS*2)+1 == message->src)
        {
                leftNode = message->val;
        }
        if((TOS_LOCAL_ADDRESS*2)+2 == message->src)
        {
                rightNode = message->val;
        }

        if(leftNode > rightNode)
        {
                maxVal = leftNode;
        }
        else
        {
                maxVal = rightNode;
        }

        if(TOS_LOCAL_ADDRESS !=0)
        {
                message2->val = maxVal;
                message2->src = TOS_LOCAL_ADDRESS;
                dest = (TOS_LOCAL_ADDRESS-1)/2;
                call SendMsg.send(dest, sizeof(IntMsg), &data);
        }

        call IntOutput.output(maxVal); //Put data to LEDS
        return m;
}

event result_t SendMsg.sendDone(TOS_MsgPtr msg, result_t success) {

        dbg(DBG_USR1, "Send Done\n");
        return SUCCESS;
}

event result_t Timer.fired() {
        uint16_t dest;
        uint16_t randomvalue;
        IntMsg *message = (IntMsg *)data.data;

        if(receiveFlag == 0)
        {
                randomvalue  = call Random.rand() % 8; //Random value from 0-7
                message->val = randomvalue;
                message->src = TOS_LOCAL_ADDRESS;
                dest = (TOS_LOCAL_ADDRESS-1)/2;

                call IntOutput.output(randomvalue);
                call SendMsg.send(dest, sizeof(IntMsg), &data);
                dbg(DBG_USR2, "Timer Fired: RandomNumber: %i\n", randomvalue);
        }
        return SUCCESS;
}
}
```

3.3. Encrypted Tree Application

3.3.1. TreeTest.nc

```
includes key;
includes PageEEPROM;

configuration KeyTest {
}

implementation {
        components Main, TimerC, RandomLFSR, GenericComm, IntToLeds,  KeyTestM, PageEEPROMC;

        Main.StdControl -> TimerC.StdControl;
        Main.StdControl -> KeyTestM.StdControl;
        Main.StdControl -> GenericComm.Control;
        Main.StdControl -> IntToLeds.StdControl;

        Main.StdControl -> PageEEPROMC;

        KeyTestM.Flash -> PageEEPROMC.PageEEPROM[unique("PageEEPROM")];
        KeyTestM.IntOutput -> IntToLeds.IntOutput;

        KeyTestM.SendMsg -> GenericComm.SendMsg[AM_KEYMSG];
        KeyTestM.ReceiveMsg -> GenericComm.ReceiveMsg[AM_KEYMSG];

        KeyTestM.Random -> RandomLFSR.Random;
        KeyTestM.Timer -> TimerC.Timer[unique("Timer")];
}
```

3.3.2. TreeTestM.nc

```
/**************************************************************************************
 * Linear Network test with point-to-point key security
 * Timothy W. Hnat
 * M.Eng Thesis
 * University of Louisville
 *************************************************************************************/

/**************************************************************************************
 * Include Section
 *************************************************************************************/
includes crc;

/**************************************************************************************
 * Module Definition
 *
 * Provides a Stdcontrol interface
 * Uses SendMsg, ReceiveMsg, IntOutput, Random, and Timer objects
 *************************************************************************************/
module KeyTestM {

  provides {
    interface StdControl;
  }

  uses {
    interface SendMsg;
    interface ReceiveMsg;
    interface IntOutput;
    interface Random;
    interface Timer;
    interface PageEEPROM as Flash;
  }
}

/**************************************************************************************
```

```
 * Implementation section
 ***********************************************************************************/
implementation {

/***********************************************************************************
 * Defines for key establishment procedures and message passing
 ***********************************************************************************/
#define MAXSENSORS 100
#define SETUP1 100
#define SETUP2 101
#define SETUP3 102
#define SETUP4 103
#define SETUP5 104
#define SETUP6 105
#define DATA 0

/***********************************************************************************
 * Required function definations
 ***********************************************************************************/
void setupKey(uint8_t msgSrc);

/***********************************************************************************
 * Global variables
 ***********************************************************************************/
struct TOS_Msg data;
struct TOS_Msg backupdata;
struct TOS_Msg rdata;

uint16_t receiveFlag;
uint16_t leftNode;
uint16_t rightNode;
uint16_t masterKey;


/***********************************************************************************
 * IntMsg structure
 *
 * src: contains the source mote ID
 * val: contains the message type value
 * data[10]: ten 16-bit numbers for data (Encrypted for communications)
 ***********************************************************************************/
typedef struct IntMsg
{
        uint8_t src;
        uint8_t val;
        uint16_t data[10];
} IntMsg;

/***********************************************************************************
 * EncKeys structure
 *
 * key: copy oef newkey for use with the creation of a newkey
 * newkey: encryption key generated by the motes
 * mote: mote id
 * status: 1 if a newkey has been set, 0 utilized the masterkey
 * count: number of remaining valid message on the given encryption key
 * moteSum: contains a crc checksum
 * resend: set if a message needs to be resent because of key creations
 ***********************************************************************************/
  typedef struct EncKeys
  {
    uint16_t key;
    uint16_t newkey;
    uint16_t mote;
    uint8_t status;
    uint8_t count;
    uint16_t moteSum;
    uint8_t resend;
  } EncKeys;

  EncKeys moteList[MAXSENSORS];


/*************************************************************************************
 * function sendSecure
 * parameters:
 *  dest: destination of message
 *  size: size of current message
 *  ptrMsg: pointer to TOS_Msg
 *  status: status value for key generation protocol
 *
 * Secure send function which encrypts the data portion of the packet and if
 * necessary establishes a point-to-point key between motes
 *************************************************************************************/
void sendSecure(uint8_t dest, uint8_t size, struct TOS_Msg * ptrMsg, uint8_t status)
{
        uint8_t i;
        IntMsg *message = (IntMsg *) ptrMsg->data;
        IntMsg *message2 = (IntMsg *)backupdata.data;

        dbg(DBG_USR1, "Dump: %d\t%d\t%d,%d,%d,%d,%d,%d,%d,%d,%d,%d\n"
                ,message->src
                ,message->val
                ,message->data[0]
                ,message->data[1]
                ,message->data[2]
                ,message->data[3]
                ,message->data[4]
                ,message->data[5]
                ,message->data[6]
                ,message->data[7]
                ,message->data[8]
                ,message->data[9]);

        if(moteList[dest].status==0) //Check the status value for key determination
        {
                for(i=0; i<10; i++) //Encrypt the data with the masterKey
                {
                        message->data[i] = message->data[i] ^ masterKey;
                }
        }
        else
        {
                for(i=0; i<10; i++) //Encyrpt the data with the generated key
                {
                        message->data[i] = message->data[i] ^ moteList[dest].newkey;
                }
        }

        //check to see if the count has expired or is invalid.
        if(moteList[dest].count == 0 || moteList[dest].count > 5)
        {
                moteList[dest].resend = 1; //set the resend flag

                //make a backup copy of the message
                memcpy(&backupdata,ptrMsg,sizeof(TOS_Msg));

                if(moteList[dest].status==0) //Check the status value for key determination
                {
                        for(i=0; i<10; i++) //Encrypt the data with the masterKey
                        {
                                message2->data[i] = message2->data[i] ^ masterKey;
                        }
                }
                else
                {
                        for(i=0; i<10; i++) //Encyrpt the data with the generated key
                        {
                                message2->data[i] = message2->data[i] ^ moteList[dest].newkey;
                        }
                }

                dbg(DBG_USR1, "Dump2: %d\t%d\t%d,%d,%d,%d,%d,%d,%d,%d,%d,%d\n"
                        ,message2->src
                        ,message2->val
                        ,message2->data[0]
                        ,message2->data[1]
                        ,message2->data[2]
                        ,message2->data[3]
                        ,message2->data[4]
                        ,message2->data[5]
                        ,message2->data[6]
                        ,message2->data[7]
                        ,message2->data[8]
                        ,message2->data[9]);

                setupKey(dest); //Start the key generation process
        }
        else
        {
                //decrease the count of the valid uses of the current key
                moteList[dest].count--;
                //send the message with the current encryption.
                call SendMsg.send(dest, size, ptrMsg);
        }
}

/*********************************************************************************
 * function sendSecureKEY
 * parameters:
 *  dest: destination of message
 *  size: size of current message
 *  ptrMsg: pointer to TOS_Msg
 *  status: status value for key generation protocol
 *
 * Secure send function which encrypts the data portion of the packet
 * Used only for creation of the point-to-point key
 ********************************************************************************/
void sendSecureKEY(uint8_t dest, uint8_t size, struct TOS_Msg * ptrMsg, uint8_t status)
{
        uint8_t i;
        IntMsg *message = (IntMsg *) ptrMsg->data;

        //check the status bit to determine which key to use
        if(moteList[dest].status==0)
        {
                for(i=0; i<10; i++) //encrypt the data with the masterKey
                {
                        message->data[i] = message->data[i] ^ masterKey;
                }
        }
        else
        {
                for(i=0; i<10; i++) //encrypt the data with the custom key
                {
                        message->data[i] = message->data[i] ^ moteList[dest].key;
                }
        }
        call SendMsg.send(dest, size, ptrMsg); //send message with the current encyrption
}

/*********************************************************************************
 * function setupKey
 * parameters:
```

```
 *  msgSrc: mote ID
 *
 * First part of the key procedures.  Store the destination mode ID. Generate random
 * bytes.  Compute a CRC at these byte locations.
 ********************************************************************************/
void setupKey(uint8_t msgSrc)
{
        IntMsg *message = (IntMsg *)data.data;
        uint16_t crc;
        uint8_t returnValue;
        uint8_t dataptr[16];
        memset((void *)dataptr,0,16);

        moteList[msgSrc].mote = msgSrc; // Store mote ID

        message->data[0] = call Random.rand(); //Generate random bytes
        message->data[1] = call Random.rand();
        message->data[2] = call Random.rand();
        message->data[3] = call Random.rand();
        message->val = SETUP1; //Set message type
        message->src = TOS_LOCAL_ADDRESS; //Store current mote ID

        //Simulation Restriction for reading EEPROM
        returnValue = call Flash.read(0,0,dataptr,16);
        returnValue = call Flash.read(1,0,dataptr,16);
        returnValue = call Flash.read(2,0,dataptr,16);
        returnValue = call Flash.read(3,0,dataptr,16);

        crc = 0; //Compute CRC value
        crc = crcByte(crc,message->data[0]);
        crc = crcByte(crc,message->data[1]);
        crc = crcByte(crc,message->data[2]);
        crc = crcByte(crc,message->data[3]);

        moteList[msgSrc].moteSum = crc; //Store CRC for future reference

        dbg(DBG_USR1, "Dump: %d\t%d\t%d,%d,%d,%d,%d,%d,%d,%d,%d,%d\n"
                ,message->src
                ,message->val
                ,message->data[0]
                ,message->data[1]
                ,message->data[2]
                ,message->data[3]
                ,message->data[4]
                ,message->data[5]
                ,message->data[6]
                ,message->data[7]
                ,message->data[8]
                ,message->data[9]);

        sendSecureKEY(msgSrc, sizeof(IntMsg), &data,SETUP1); //Send bytes to other mote
}

/*********************************************************************************
 * function validationRequest
 * parameters:
 *  msgSrc: mote ID
 *  *pktdata: pointer to data from packet
 *
 * Checksums the given bytes and returns the results
 ********************************************************************************/
void validationRequest(uint8_t msgSrc, uint16_t * pktdata)
{
        IntMsg *message = (IntMsg *)data.data;
        uint16_t crc;
        uint8_t returnValue;
        uint8_t dataptr[16];
        memset((void *)dataptr,0,16);
        moteList[msgSrc].mote = msgSrc; //Store mote ID

        //Simulation Restriction for reading EEPROM
        returnValue = call Flash.read(0,0,dataptr,16);
        returnValue = call Flash.read(1,0,dataptr,16);
        returnValue = call Flash.read(2,0,dataptr,16);
        returnValue = call Flash.read(3,0,dataptr,16);

        crc = 0; //Compute CRC value
        crc = crcByte(crc,pktdata[0]);
        crc = crcByte(crc,pktdata[1]);
        crc = crcByte(crc,pktdata[2]);
        crc = crcByte(crc,pktdata[3]);

        message->data[0] = crc; //Add CRC to data portion of packet
        message->val = SETUP2; //Set message type
        message->src = TOS_LOCAL_ADDRESS; //Store current mote ID

        dbg(DBG_USR1, "Dump: %d\t%d\t%d,%d,%d,%d,%d,%d,%d,%d,%d,%d\n"
                ,message->src
                ,message->val
                ,message->data[0]
                ,message->data[1]
                ,message->data[2]
                ,message->data[3]
                ,message->data[4]
                ,message->data[5]
                ,message->data[6]
                ,message->data[7]
                ,message->data[8]
                ,message->data[9]);

        sendSecureKEY(msgSrc, sizeof(IntMsg), &data,SETUP2); //Send bytes to other mote
}

/*************************************************************************************
 * function setupKey2
 * parameters:
 *  msgSrc: mote ID
 *  *pktdata: pointer to data from packet
 *
 * Verifies the checksum from the other mote, if they match send next message
 ************************************************************************************/
void setupKey2(uint8_t msgSrc, uint16_t * pktdata)
{
        IntMsg *message = (IntMsg *)data.data;

        if(moteList[msgSrc].moteSum == pktdata[0]) //Check checksums
        {
                message->val = SETUP3; //Set message type
                message->src = TOS_LOCAL_ADDRESS; //Store current mote ID

                dbg(DBG_USR1, "Dump: %d\t%d\t%d,%d,%d,%d,%d,%d,%d,%d,%d,%d\n"
                        ,message->src
                        ,message->val
                        ,message->data[0]
                        ,message->data[1]
                        ,message->data[2]
                        ,message->data[3]
                        ,message->data[4]
                        ,message->data[5]
                        ,message->data[6]
                        ,message->data[7]
                        ,message->data[8]
                        ,message->data[9]);

                //Send bytes to other mote
                sendSecureKEY(msgSrc, sizeof(IntMsg), &data,SETUP3);
        }
}

/*********************************************************************************
 * function validationRequest2
 * parameters:
 *  msgSrc: mote ID
 *
 * Store the destination mode ID. Generate random
 * bytes.  Compute a CRC at these byte locations.
 *********************************************************************************/
void validationRequest2(uint8_t msgSrc)
{
        IntMsg *message = (IntMsg *)data.data;
        uint16_t crc;
        uint8_t returnValue;
        uint8_t dataptr[16];
        memset((void *)dataptr,0,16);

        message->data[0] = call Random.rand(); //Generate random bytes
        message->data[1] = call Random.rand();
        message->data[2] = call Random.rand();
        message->data[3] = call Random.rand();
        message->val = SETUP4; //Set message type
        message->src = TOS_LOCAL_ADDRESS; //Store current mote ID

        //Simulation Restriction for reading EEPROM
        returnValue = call Flash.read(0,0,dataptr,16);
        returnValue = call Flash.read(1,0,dataptr,16);
        returnValue = call Flash.read(2,0,dataptr,16);
        returnValue = call Flash.read(3,0,dataptr,16);

        crc = 0; //Compute CRC value
        crc = crcByte(crc,message->data[0]);
        crc = crcByte(crc,message->data[1]);
        crc = crcByte(crc,message->data[2]);
        crc = crcByte(crc,message->data[3]);

        moteList[msgSrc].moteSum = crc; //Store crc value for future reference

        dbg(DBG_USR1, "Dump: %d\t%d\t%d,%d,%d,%d,%d,%d,%d,%d,%d,%d\n"
                ,message->src
                ,message->val
                ,message->data[0]
                ,message->data[1]
                ,message->data[2]
                ,message->data[3]
                ,message->data[4]
                ,message->data[5]
                ,message->data[6]
                ,message->data[7]
                ,message->data[8]
                ,message->data[9]);

        sendSecureKEY(msgSrc, sizeof(IntMsg), &data,SETUP4); //Send bytes to other node
}

/*********************************************************************************
 * function setupKey3
 * parameters:
 *  msgSrc: mote ID
 *  *pktdata: pointer to data from packet
 *
 * Computes CRC from received bytes and returns the result
 *********************************************************************************/
void setupKey3(uint8_t msgSrc, uint16_t * pktdata)
{
        IntMsg *message = (IntMsg *)data.data;
        uint16_t crc;
        uint8_t returnValue;
        uint8_t dataptr[16];
        memset((void *)dataptr,0,16);

        //Simulation Restriction for reading EEPROM
        returnValue = call Flash.read(0,0,dataptr,16);
        returnValue = call Flash.read(1,0,dataptr,16);
        returnValue = call Flash.read(2,0,dataptr,16);
        returnValue = call Flash.read(3,0,dataptr,16);

        crc = 0; //Computer CRC value
        crc = crcByte(crc,pktdata[0]);
        crc = crcByte(crc,pktdata[1]);
        crc = crcByte(crc,pktdata[2]);
        crc = crcByte(crc,pktdata[3]);

        message->data[0] = crc; //Add CRC to data
        message->val = SETUP5; //Set message type
        message->src = TOS_LOCAL_ADDRESS; //Store current mote ID

        dbg(DBG_USR1, "Dump: %d\t%d\t%d,%d,%d,%d,%d,%d,%d,%d,%d,%d\n"
                ,message->src
                ,message->val
                ,message->data[0]
                ,message->data[1]
                ,message->data[2]
                ,message->data[3]
                ,message->data[4]
                ,message->data[5]
                ,message->data[6]
                ,message->data[7]
                ,message->data[8]
                ,message->data[9]);

        sendSecureKEY(msgSrc, sizeof(IntMsg), &data,SETUP5); //Send bytes to other node
}

/*************************************************************************************
 * function validationRequest3
 * parameters:
 *  msgSrc: mote ID
 *  *pktdata: pointer to data from packet
 *
 * Verifies the crc values from both motes.  If they match, it generates a random key
 * and sends this to the other mote.  Status and count are set.
 ************************************************************************************/
void validationRequest3(uint8_t msgSrc, uint16_t * pktdata)
{
        IntMsg *message = (IntMsg *)data.data;

        if(moteList[msgSrc].moteSum == pktdata[0]) //Check CRC values
        {
                moteList[msgSrc].newkey = call Random.rand(); //Generation Random Key

                message->data[0] = moteList[msgSrc].newkey;
                message->val = SETUP6; //Set message type
                message->src = TOS_LOCAL_ADDRESS; //Store current mote ID

                dbg(DBG_USR1, "Dump: %d\t%d\t%d,%d,%d,%d,%d,%d,%d,%d,%d,%d\n"
                        ,message->src
                        ,message->val
                        ,message->data[0]
                        ,message->data[1]
                        ,message->data[2]
                        ,message->data[3]
                        ,message->data[4]
                        ,message->data[5]
                        ,message->data[6]
                        ,message->data[7]
                        ,message->data[8]
                        ,message->data[9]);

                sendSecureKEY(msgSrc,sizeof(IntMsg),&data,SETUP6); //Send bytes to other node
                moteList[msgSrc].status = 1; //Set status to 1 to indicate a random key
                moteList[msgSrc].count = 5; //Set number of uses to the key
                moteList[msgSrc].key=moteList[msgSrc].newkey;
        }
}

/*************************************************************************************
 * function setupKey4
 * parameters:
 *  msgSrc: mote ID
 *  *pktdata: pointer to data from packet
 *
 * Checks the incoming key and if necessary, resend the backed up message
 *************************************************************************************/
void setupKey4(uint8_t msgSrc, uint16_t * pktdata)
{
        uint16_t moteSum;

        moteSum = pktdata[0];
        if(moteSum > 0) //Check the incomming key
        {
                moteList[msgSrc].status = 1; //Set status to indicate the presence of a key
                moteList[msgSrc].count = 5; //Set the number of uses of the key
                moteList[msgSrc].newkey = pktdata[0]; //Store the incoming key
                dbg(DBG_USR1, "Mote Link %d<-->%d is complete.  Key: %d\n"
                        ,TOS_LOCAL_ADDRESS, msgSrc,pktdata[0]);

                if(moteList[msgSrc].resend == 1) //Resend the message if necessary
                {
                        sendSecure((TOS_LOCAL_ADDRESS-1)/2, sizeof(IntMsg), &backupdata,DATA);
                        moteList[msgSrc].resend = 0; //Reset the resend flag
                }
                moteList[msgSrc].key=pktdata[0]; //Save the old key
        }
}


/*************************************************************************************
 * StdControl.init()
 * The initialization function for the sensors
 *************************************************************************************/
command result_t StdControl.init() {
        uint8_t i;
        receiveFlag = 0;
        masterKey = 12345; //Set Master Key

        for(i=0; i<MAXSENSORS; i++) //Set all elements of moteList to 0
        {
                moteList[i].key=0;
                moteList[i].mote=0;
                moteList[i].status=0;
                moteList[i].count=0;
        }
        dbg(DBG_USR1, "Starting Node\n");
        return SUCCESS;
}

/*************************************************************************************
 * StdControl.start()
 * This starts all the necessary components of the system.
 *************************************************************************************/
command result_t StdControl.start() {
        dbg(DBG_USR3, "Starting Timer\n");
        return call Timer.start(TIMER_REPEAT, 10000);
}

/************************************************************************************
 * StdControl.start()
 * This stop all the necessary components of the system.
 ***********************************************************************************/
command result_t StdControl.stop() {
        return call Timer.stop();
}

event result_t Flash.syncDone(result_t success) {
        uint8_t readData[16];
        if(success==SUCCESS)
        {
                call Flash.read(0,0,readData,16);
        }
        return SUCCESS;
}

event result_t Flash.flushDone(result_t success) { return SUCCESS;}
event result_t Flash.eraseDone(result_t success) { return SUCCESS;}
event result_t Flash.computeCrcDone(result_t success, uint16_t crc) { return SUCCESS;}

event result_t Flash.readDone(result_t success) {
        return SUCCESS;
}

event result_t Flash.writeDone(result_t success ){
        if(success == SUCCESS)
        {
                call Flash.syncAll();
        }
        return SUCCESS;
}

/************************************************************************************
 * IntOutput.outputComplete()
 * Indicates the LED output is successful
 ***********************************************************************************/
event result_t IntOutput.outputComplete(result_t success) {
        return success;
  }

/************************************************************************************
 * ReceiveMsg.receive()
 * Handles all of the incoming message.  Routing all message to the appropiate
 * locations.
 ***********************************************************************************/
event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr m) {
        uint8_t i,dest;
        IntMsg *message = (IntMsg *)m->data;
        IntMsg *message2 = (IntMsg *)rdata.data;

        uint8_t statusValue,messageSource;
        uint16_t maxVal;

        messageSource = message->src; //Set message source ID
        statusValue = message->val; //Set message status value

        //Check the status bit to determine which key to utilized
        if(moteList[message->src].status==0)
        {
                for(i=0; i<10; i++) //Use master key
                {
                       message->data[i] = message->data[i] ^ masterKey;
                }
```

```
}
else
{
        for(i=0; i<10; i++) //Use generated point-to-point key
        {
                message->data[i] = message->data[i] ^ moteList[message->src].key;
        }
}

if(message->val < SETUP1) //Check for lack of setup messages
{
        receiveFlag = 1;
        if(moteList[message->src].status==1) //Fix encryption for this data
        {
                for(i=0; i<10; i++)
                {
                        message->data[i] =
                                message->data[i] ^
                                moteList[message->src].key ^
                                moteList[message->src].newkey;
                }
        }
        message2->val = message->val; //Save copy of message
        message2->src = message->src;
        for(i=0; i<10; i++)
                message2->data[i] = message->data[i];
}

if(statusValue == SETUP1) //Setup Key Procedure
{
        validationRequest(messageSource,message->data);
}
else if(statusValue == SETUP2)
{
        setupKey2(messageSource,message->data);
}
else if(statusValue == SETUP3)
{
        validationRequest2(messageSource);
}
else if(statusValue == SETUP4)
{
        setupKey3(messageSource,message->data);
}
else if(statusValue == SETUP5)
{
        validationRequest3(messageSource,message->data);
}
else if(statusValue == SETUP6)
{
        setupKey4(messageSource,message->data);
}
else //All other messages
{
        dbg(DBG_USR3, "Processing Data\t%d\t%d\n",message->src,message->val);

        call Timer.stop();

        receiveFlag = 1;
        if((TOS_LOCAL_ADDRESS*2)+1 == message->src)
        {
                leftNode = message->data[0];
        }
        if((TOS_LOCAL_ADDRESS*2)+2 == message->src)
        {
                rightNode = message->data[0];
        }
```

```
                if(leftNode > rightNode)
                {
                        maxVal = leftNode;
                }
                else
                {
                        maxVal = rightNode;
                }

                if(TOS_LOCAL_ADDRESS!=0)
                {
                        message2->val = maxVal;
                        message2->src=TOS_LOCAL_ADDRESS;
                        dest = (TOS_LOCAL_ADDRESS-1)/2;
                        sendSecure(dest,sizeof(IntMsg),&rdata,DATA);
                }
        call IntOutput.output(maxVal); //Put Data on LEDs
        }
        return m;
  }

/*************************************************************************************
 * SendMsg.sendDone()
 *
 * Reports the completion of the send message.
 ************************************************************************************/
event result_t SendMsg.sendDone(TOS_MsgPtr msg, result_t success) {
        dbg(DBG_USR2, "Completed sending message\n");
        return SUCCESS;
}

/*************************************************************************************
 * Timer.fired()
 *
 * Executes at intervals.  This provides test data on the network.
 ************************************************************************************/
event result_t Timer.fired() {
        uint16_t dest;
        uint16_t randomvalue;
        IntMsg *message = (IntMsg *)backupdata.data;

        //check receive flag to limit transmission from nodes that have received data
        if(receiveFlag == 0)
        {
                randomvalue  = call Random.rand() % 8; //Random value from 0-7
                message->val = randomvalue;
                message->src = TOS_LOCAL_ADDRESS;
                message->data[0] = call Random.rand() % 8;
                message->data[1] = call Random.rand();
                message->data[2] = call Random.rand();
                message->data[3] = call Random.rand();
                message->data[4] = call Random.rand();
                message->data[5] = call Random.rand();
                message->data[6] = call Random.rand();
                message->data[7] = call Random.rand();
                message->data[8] = call Random.rand();
                message->data[9] = call Random.rand();
                dest = (TOS_LOCAL_ADDRESS-1)/2;

                dbg(DBG_USR3, "Sending Message to \t%d\tval: %d\n", dest,message->data[0]);

                call IntOutput.output(message->data[0]);
                dbg(DBG_USR1, "TIMER Message: %d\n", moteList[dest].count);
                sendSecure(dest, sizeof(IntMsg), &backupdata,DATA); //Send message
        }
        return SUCCESS;
}
}
```