AN EFFICIENT KEY UPDATE SCHEME FOR WIRELESS SENSOR NETWORKS

By

KAMINI B. PRAJAPATI

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

DECEMBER 2005

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of KAMINI B. PRAJAPATI find it satisfactory and recommend that it be accepted.

_____

Chair

_____

_____

ACKNOWLEDGEMENT

First, I would like to thank my advisor Professor Jabulani Nyathi for all his help, support and guidance in my research work. I would also like to thank Professor Murali Medidi and Professor Sirisha Medidi for being the committee members of this research.

Next, I would like to thank the Center for Teaching, Learning and Technology (CTLT) for awarding me Research Assistantships.

And last but not least, I would like to thank my parents and my husband for their encouragement and support.

AN EFFICIENT KEY UPDATE SCHEME FOR WIRELESS SENSOR NETWORKS

Abstract

by Kamini B. Prajapati, M.S.
Washington State University
December 2005

Chair: Jabulani Nyathi

Wireless sensors are highly resource constrained in terms of memory, power and processing capability. However, critical applications of these networks demand security features to be implemented. Some researchers have approached this problem and provided schemes like TinySec, TinyPK, Localized Encryption and Authentication Protocol (LEAP), Elliptical curve Cryptography (ECC), etc. TinySec is the most successful implementation till now. This research enhances the basic TinySec security by providing an efficient key update mechanism on top of TinySec. The simulation results show that the memory overhead for this scheme is 1.66% and the computational cost is minimal. There is no latency or bandwidth overhead.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENT……...iii

ABSTRACT……..iv

LIST OF TABLES……..vii

LIST OF FIGURES……viii

CHAPTERS

1. **INTRODUCTION……1**
   - **1.1 WSN Models (Topology)…….....2**
   - **1.2 Need for security in WSNs…….5**
   - **1.3 WSN Models (Key distribution)……....6**
   - **1.4 Components of Sensor Node……...……...8**
   - **1.5 Problem Statement……18**
   - **1.6 Conclusion…….……19**
2. **BACKGROUND WORK...……20**
   - **2.1 TinySec……20**
   - **2.2 Conclusion ……...24**
3. **RELATED WORK……...25**
   - **3.1 ECC……25**
   - **3.2 SPAWKU and SPAGKU……...29**
   - **3.3 LEAP……...33**
   - **3.4 TinyPK……37**
   - **3.5 Conclusion……..40**

4. PROPOSED SCHEME……..41

4.1 Basic Algorithm……....42

4.2 Cost Analysis…….43

4.3 Security Analysis……..43

4.4 Applicability to WSN Models (Topology)……..45

4.5 Conclusion……...……..48

5. IMPLMENTATION……..49

5.1 TOSSIM……49

5.2 Compiling and Executing an Application with TOSSIM……53

5.3 Algorithm Implementation and Results...…….53

5.4 Comparison with other schemes……62

5.5Conclusion…….63

6. CONCLUSION AND FUTURE WORK……..……...64

6.1 Implementation on Motes……...64

6.2 Enhancement of the Scheme……...64

7. BIBLIOGRAPHY……65

LIST OF TABLES

Table 1.1: Hardware characteristics of Motes...……..9

Table 1.2: Sensor Boards for Mica2 Motes…….11

Table 2.1: TinySec Overhead Summary……..24

Table 3.1: SPAWKU and SPAGKU - Memory requirements ……..32

Table 3.2: LEAP - RAM requirements as a function of number of neighbors…35

Table 3.3: TinyPK – Memory requirements for Diffie-Hellman key exchange..40

Table 5.1: Memory Overhead – Proposed Algorithm…….……...54

Table 5.2: Power measurement results – TinySec modes…….…….55

Table 5.3: CPU cycle results – TinySec modes……...……56

Table 5.4: CPU cycle measurement for key update task..…….57

Table 5.5: Power results – 10 node network……...58

Table 5.6: CPU cycle results – 10 node network……60

Table 5.7: Key Update Schemes - Comparison Summary…….63

LIST OF FIGURES

Figure 1.1: Network Models……...2

Figure 1.2: Hardware Block diagram for Sensor node……9

Figure 1.3: TinyOS component structure and communication…….13

Figure 1.4: Layered Model of TinyOS components……16

Figure 1.5: Typical TinyOS component graph for entire application……..17

Figure 1.6: TinyOS CRC packet format……..18

Figure 2.1: Block Diagram for SkipJack Encryption/Decryption…….21

Figure 2.2: TinySec – Auth packet format……..23

Figure 2.3: TinySec – AE packet format…….23

Figure 3.1: SPAWKU/SPAGKU Key Update packet format……...30

Figure 3.2: TinyPK – Execution time for first exponentiation (DH)……39

Figure 3.3: TinyPK – Execution time for second exponentiation (DH)……39

Figure 5.1: TOSSIM Architecture……...50

Figure 5.2: Screenshot of TinyViz for 10 node simulation…….62

Figure 5.3: Multi-Hop Network Simulation Results……..63

CHAPTER ONE

INTRODUCTION

The advancement in fields of wireless communication and electronics enabled the development of low cost, low power, multifunctional sensor nodes. These tiny sensor nodes, which consist of sensing, data processing, and communication components, leverage the idea of sensor networks. Sensor networks represent a significant improvement over traditional sensors, [1]. The term 'sensor network' refers to a heterogeneous system consisting of tiny sensors and actuators with general purpose computing elements, [2]. Typical application involves deploying hundreds or thousands of low-power, low-cost sensor nodes for a specified purpose, like habitat monitoring, burglar alarms, prognostic health management, battlefield management, etc. For the majority of applications, sensor networks are designed to be unattended for long periods after deployment and battery recharging or replacement may not be possible, [2].

Sensors being primarily wireless devices, the sensor networks seem to have a close resemblance to the typical wireless networks, except for the fact that sensor nodes are highly resource constrained with very limited memory, power and computational capabilities. However, this resource constrained nature of sensor nodes influence the network design and behavior to such an extent that sensor networks have significant differences when compared to typical wireless networks. Most importantly, any protocol for sensor networks should be designed keeping the constrained resources in mind. Secondly, because of low power, the transmission range of sensor nodes is limited leading to multi-hop transmission/communication

pattern. Further, life of low-powered sensor nodes is short, and so a large number of nodes are densely deployed in a network to keep it functional for longer time, [3]. Sensor networks employ some techniques for in-network processing to save the resources and improve the processing time. One of them is passive participation which refers to taking the action based on overheard traffic for optimized usage of resources. The other is data aggregation, in which some intermediate nodes process the readings coming form multiple nodes and forward a single message to the base station or the gateway instead of forwarding every reading. This helps in eliminating redundancy, minimizing the number of transmissions, and thus saving energy.

1.1 Wireless Sensor Network Models based on Network Topology

This section classifies the sensor networks based on their configuration and types of nodes. Primarily they can be classified into two types as shown in figure 1.1:

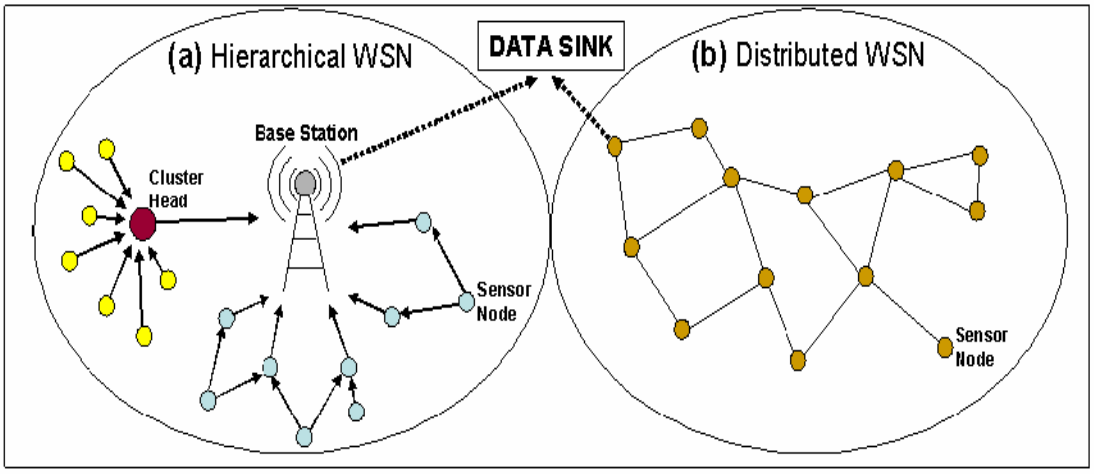


Figure 1.1: Network Models – Hierarchical and Distributed Wireless Sensor Networks, [4]

1.1.1 Hierarchical/ Infrastructure based wireless sensor networks

In these networks, there is a hierarchy of nodes in terms of resources and functions. The most powerful element is the Base Station. Base station is a powerful data processing and storage unit which collects sensor readings, perform costly operations and manage the network. It is usually the gateway to another network, or an access point for human interface. Transmission power of base station is usually enough to reach all sensor nodes, [4].

The next level of sensor nodes is called group heads or cluster heads. The inclusion of these nodes is optional depending on the network size and the application. These nodes have better resources compared to the sensor nodes which form the lowest level of this model. Cluster heads are responsible for intermediate data processing/in network processing, data aggregation, e.g. collect and process the readings of group nodes and send a single reading to the base station. The base station, in turn, performs computation on readings from multiple cluster heads.

The sensor nodes i.e. nodes with least resources form the majority of the network. They provide the readings for the parameters being sensed by the network. Since their transmission range is limited, they significantly depend on the ad-hoc communication for reaching distant nodes and the base station. Thus, the communication pattern, [2] in these networks is commonly of the following three types:

- **One-to-Many**

This can be further classified into two categories:

- *Broadcast* – A message sent by the base station to all the nodes in the network. It is also referred to as network-wise communication. e.g. control information like routing beacons, etc.
- *Multicast* – This refers to a message sent by a cluster head to all the group nodes. It is also called group-wise communication.

- **Many-to-One**

Many-to-One refer to messages sent by multiple sensor nodes to the base station or to the cluster heads. Usually, these messages are the data readings as sensed by the node. Multi-hop communication approach is followed to reach the desired node.

- **One-to-One**

This is link-wise or pair-wise communication between two sensor nodes. Neighboring nodes send localized messages to discover each other and for mutual coordination. This is also called *unicast* transmission.

1.1.2 Distributed Wireless sensor networks

The communication paradigm of distributed wireless sensor networks is similar to wireless ad hoc networks, where network nodes self-organize in an ad hoc fashion. A group of wireless nodes form a network without any fixed and centralized infrastructure. Network topology is not known prior to deployment, and sensor nodes are randomly scattered over the target area. After deployment, each sensor node scans its radio coverage area to figure out its neighbors, [4]. When two nodes wishing to communicate are relatively far apart, intermediate nodes forward packets along a multi-hop wireless route. The network nodes rely on peers for all or most of the services needed and for basic needs of communications. Due to the lack of centralized

control and management, nodes rely on fully distributed and self-organizing protocols to coordinate their activities, [3].

Unlike the wireless ad-hoc networks, sensor nodes are stationary. However, the topology can still change frequently due to high probability of node failure. Also, the number of nodes in sensor networks is much larger than that of wireless ad hoc networks and the nodes are densely deployed.

In these networks all the nodes have the same capabilities. These networks follow similar communication patterns to that of hierarchical networks, namely One-to-Many, Many-to-One, and One-to-One as previously described.

1.2 Need for security in Wireless Sensor Networks

Sensor networks can facilitate large-scale, real-time data processing in complex environments. They interact closely with their physical environment and with people. Their applications involve protecting and monitoring critical military, environmental, safety-critical or domestic infrastructures and resources.

Security is important in sensor networks for the following reasons:

- Since sensor networks actively monitor their surroundings, it often easy to deduce information other than the data monitored. Such unwanted information leakage often results in privacy breaches of the people in the environment.
- Wireless communication employed by the sensor networks facilitates eavesdropping and packet injection by an adversary.

The combination of these factors demand security for sensor networks to ensure operation safety, secrecy of sensitive data, and privacy for people in sensor environment. Without proper security mechanisms, networks will be confined to

limited controlled environment. This will restrict their application scope. Therefore, in order to monitor and protect safety-critical resources and structures, security is needed in wireless sensor networks.

1.3 Wireless sensor network Models based on Key Distribution

Secure sensor networks can be categorized based on key distribution within the network. In the following subsections I briefly describe the three main distribution categories.

1.3.1 Network-wide shared key

In these networks, all the nodes in a particular network share the same key for secure communication/security provision. The advantage is that key distribution is easy and keys can be pre-loaded into the sensor nodes prior to deployment. As only one key need to be stored, the memory overhead is less. Security operations are fast since the issue of selection from multiple keys does not arise. This scheme is scalable and suitable for large networks. However, the major disadvantage of this keying mechanism is that compromise of a single node reveals the key and thus, the entire network can become insecure, [5].

1.3.2 Pair-wise Shared Key

In this keying mechanism, every node of a network shares a unique key with every other node in the network, hence the name pair-wise, e.g. the base station has a pair-wise key with every node in the network.

There is another variant of this keying mechanism which is slightly different from pair-wise, and is commonly referred to as link-wise keying mechanism. In link-wise structure, every node shares a unique key with only its first hop neighbors or the

nodes in its transmission range. The link-wise scheme is suitable for small to medium sized distributed networks where two distant nodes have to rely on multi-hop communication anyways. However, the disadvantage is that the message needs to be decrypted and re-encrypted at each intermediate node causing wastage of scarce computational resources.

The pair-wise network is most resilient when it comes to node capture attacks, [5]. A compromised node can only decrypt traffic addressed to it. The main issues with these networks are memory requirements and scalability. If there are N nodes in a network, a node needs to store N-1 keys. This number can be high for densely deployed sensor networks. The link-wise structure does provide reduced memory needs, however as mentioned earlier, power consumption increases. For sending a message to a particular node, the node has to select the appropriate key from N-1 keys. This selection time may increase for larger N. Also, the key distribution is more challenging in this case, especially for distributed networks. Further, passive participation and local broadcast are incompatible with this mechanism as a node cannot decrypt and authenticate message not addressed to it.

1.3.3 Group-wise shared key

In this mechanism, the nodes belonging to a group share the same key whereas the nodes in different groups have different keys. For the application of this key structure, it becomes necessary to divide the network into groups, each having one key. The cluster heads or group heads will be the point of interaction between different groups. This structure is more suitable for hierarchical networks. However, it can be applied to distributed networks as well.

This network has an intermediate level of resilience between the network-wide and pair-wise structures. In case of compromised node attacks, there is graceful degradation as the adversary can decrypt and inject traffic only for particular group nodes and the confidentiality of other group's messages is retained. The impact would be more if the compromised node is a group head or cluster head. The benefit of this scheme is that it enables passive participation and local broadcast and this helps in resource conservation. Also, the number of keys stored by each node is less, usually one for all group nodes except for group head who need to store the keys for other neighboring group heads. The drawback is that for inter group messages, the group head has to decrypt them first and then re-encrypt them with suitable key for neighboring group, leading to extra resource consumption.

1.4 Components of a Sensor node

Researchers at UC Berkeley were the first to come up with the concept of intelligent wireless sensors and sensor networks. They have developed sensor devices called 'motes' and an operating system called 'TinyOS' which is specifically designed to run on the motes, [6]. The term 'mote' is used to refer the sensor node as whole; however, it's the hardware component of the sensor node. The software component includes the TinyOS operating system and various applications and protocols developed on top of it.

1.4.1 Motes

A mote is essentially a microcontroller along with a number of sensors attached to it. Some motes have integrated sensors. These motes are very small in size due to the nature and purpose of sensor networks applications. Some examples of motes are

Telos, Rene, Mica and Mica2. Table 1.1 gives the hardware and radio specifications for some of these motes. Mica2 is the most representative of all and the majority of the research done today in the field of wireless sensor networks use Mica2 motes.

Table 1.1: Hardware characteristics of motes, [6]

Mote Type	**Renee**	**Mica**	**Mica2**	**Mica2Dot**
Microcontroller				
Type	Atmega163	Atmega128	Atmega128	Atmega128
CPU Clock (MHz)	4	4	8	4
Program Memory (KB)	16	128	128	128
RAM (KB)	1	4	4	4
Non-volatile Storage				
Size (KB)	32	512		
Radio Communication				
Radio	RFM TR1000		Chipcon CC1000	
Frequency	916		916 / 433	
Transmit Power Control	Programmable resistor potentiometer		Programmable via CC1000 registers	
Encoding	SecDed (Software)		Manchester (Hardware)	

Mica2 Components

Mica2 consists of five hardware blocks as shown in figure 1.2 below.

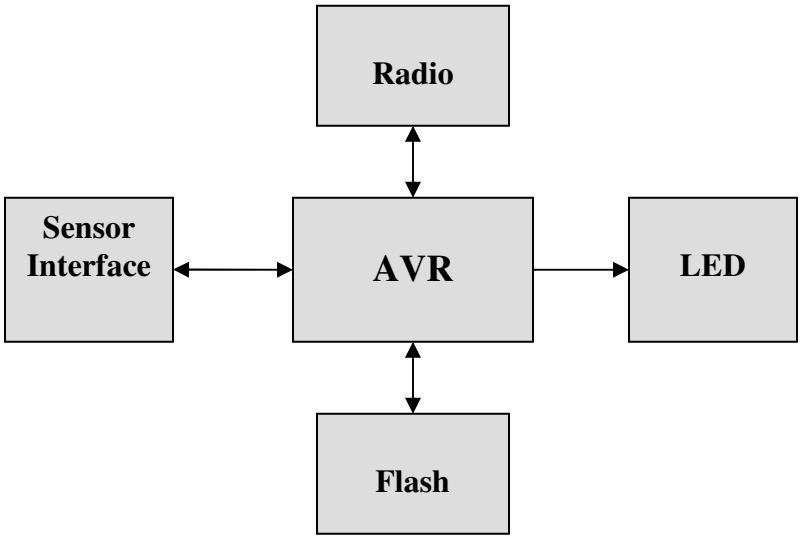


Figure 1.2 – Hardware Block diagram for sensor node, [6]

The main block of Mica2 is the microcontroller or processor - Atmel Atmega128 AVR. AVR is an 8-Bit Harvard architecture, with separate instruction and data memory. AVR micro controllers provide several sleep modes. The purpose of these modes is to provide a way of suspending program execution when necessary, thereby reducing power consumption. The microcontroller unit (MCU) is responsible for control of the sensors and the execution of communication protocols and signal processing algorithms on the gathered sensor data. The microcontroller interfaces with Radio, LEDS, Flash Memory and Sensor board/Programming interface, [6].

- **LEDS** - Three Programmable LEDs are connected to the AVR in the Mica2 motes. These may be used for status and output of digital values.
- **Flash Memory** - A 512KB Serial Flash memory chip is attached to one of the AVR's UART ports to allow permanent storage and data logging in the motes.
- **Radio** - The radio used is a low-power, single-chip UHF transceiver from Chipcom called CC1000. The CC1000 is designed for very low power and very low voltage wireless applications. The circuit is mainly intended for frequency bands at 315, 433, 868 and 915 MHz, but can easily be programmed for operation at other frequencies in the 300-1000 MHz range. The main operating parameters of CC1000 can be programmed via a serial bus, thus making CC1000 a very easy to use transceiver. In general radio can operate in four distinct modes of operation: Transmit, Receive, Idle, and Sleep (Off).

  The features of mica2 radio can be summarized as:

  - Frequency selectable from 300-1000 MHz
  - Frequency Shift Keying modulation with data rates up to 19.2 Kbps

- Hardware based Manchester encoding
- Integrated bit synchronizer
- -110 dBm sensitivity
- selectable power states
- digital control interface using special function register

- **Sensing Hardware** - The modular design of the motes allows a wide range of analog and digital sensors to be attached to the mote. However, the reference sensor board for the mica platform is the “Mica Sensorboard". A variety of these sensor boards are available. They allow for a range of different sensing modalities as well as interface to external sensor via prototyping areas or screw terminals. Table 1.2 gives a list of sensor boards available for Mica2.

Table 1.2 – Sensor Boards for Mica2 motes, [6]

Part Number	Motes Supported	Sensors and Features
MTS101CA	MICA, MICA2	Light, Temperature, Prototype Area
MTS300CA	MICA, MICA2	Light, Temperature, Acoustic, and Sounder
MTS310CA	MICA, MICA2	Light, Temperature, Acoustic, Sounder, 2-Axis Accelerometer(ADXL202), and 2-Axis Magnetometer
MDA300CA	MICA	Light, Humidity, General Purpose Interface for External Sensors
MDA500CA	MICA2DOT	General Purpose Interface

1.4.2 TinyOS

TinyOS is a small event-driven, component based operating system, designed specifically for supporting the concurrency intensive operations required by networked sensors with minimum hardware requirements, [7]. The TinyOS framework contains numerous pre-built sensor applications and algorithms e.g. multi-hop ad-hoc routing and supports different sensor node platforms. The design of

TinyOS is based on the specific sensor network characteristics: small physical size, low-power consumption, concurrency-intensive operation, multiple flows, limited physical parallelism and controller hierarchy, diversity in design and usage, and robust operation to facilitate the development of reliable distributed applications, [8]. TinyOS is optimized in terms of memory usage and energy efficiency. It provides defined interfaces between the components which reside in neighboring layers.

1.4.2.1 TinyOS Design

A complete system configuration in TinyOS consists of a tiny scheduler and a graph of components**.**

- **Components -** There are two types of components in TinyOS: Modules and Configurations. Modules provide application code, implementing one or more interface. Configurations are used to assemble other components together, connecting interfaces used by components to interfaces provided by others. As shown in Figure 1.3, a component provides and uses interfaces. These interfaces are the only point of access to the component, and are bi-directional. An interface declares a set of functions called commands that the interface provider must implement and another

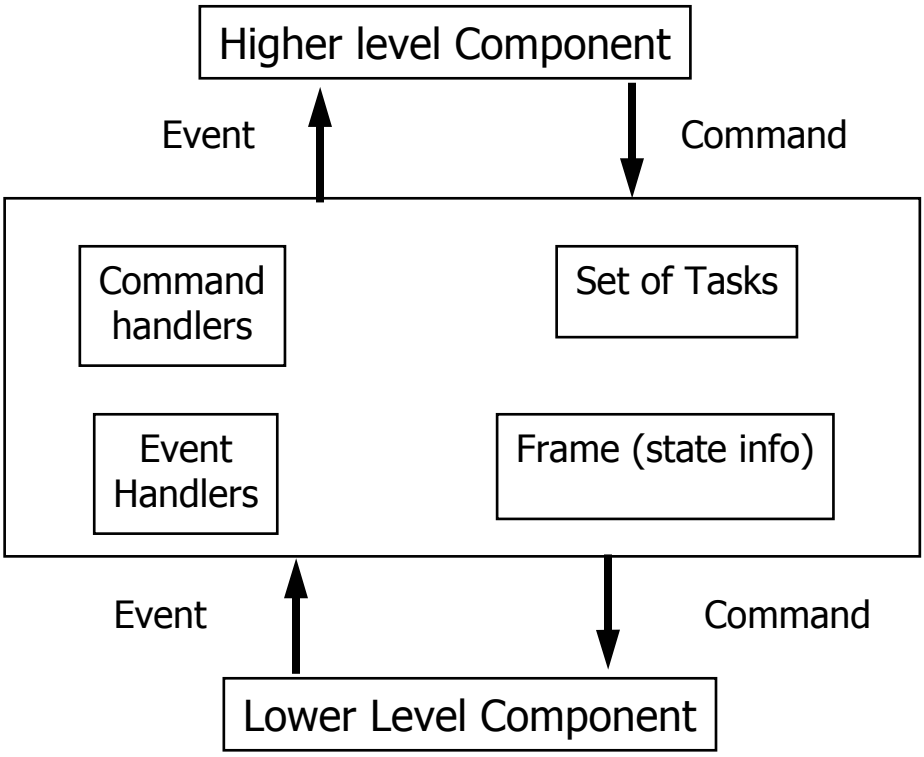


Figure 1.3: TinyOS Component Structure and Communication

set of functions called events that the interface user must implement. For a component to call the commands in an interface, it must implement the events of that interface. A single component may use or provide multiple interfaces and multiple instances of the same interface. The events give rise to tasks which are non-critical and they handle computation and processing associated with the events.

Thus, a component has four interrelated parts:

- a set of command handlers
- a set of event handlers
- an encapsulated fixed-size frame
- a bundle of simple tasks

Tasks, commands, and handlers execute in the context of the frame and operate on its state.

- **Scheduler -** The design of the TinyOS Kernel is based on a two (2) level scheduling structure consisting of events and tasks.
    - **Events**: Events are intended to do a small amount of processing (e.g. Timer interrupts, ADC interrupts) and can preempt (i.e. interrupt) longer running tasks.
    - **Tasks:** Tasks are intended to do a larger amount of processing and are not time critical (e.g. computing an average on an array). Tasks always run to completion with respect to other Tasks. This "run to completion" property of tasks is very important and implies that a TinyOS system application only needs a single stack.

The scheduler support Concurrency Model. TinyOS executes only one program consisting of selected system components and custom components needed for a single application. There are two threads of execution: tasks and hardware event handlers, [8]. Tasks are functions whose execution can be deferred. Tasks execute asynchronously with respect to events, thereby, simulating concurrency within each component. However, tasks must never block or spin wait or they will prevent progress in other components. Context Switching is not possible with TinyOS because it utilizes a single stack.

1.4.2.2 Active Message

In TinyOS, legacy communication (TCP/IP, sockets, routing protocols like OSPF,etc) cannot be used because they require intensive bandwidth and are centered on “stop and wait” semantics, [6,10]. The socket/TCP/IP uses too much memory for buffering and threads. Further data are buffered in network stack until application threads read

it and application threads are blocked until data is available. Sensor networks need to follow real time constraints and have low processing overhead. The Active Message (AM) types are similar to port numbers in TCP/IP. The AM type specifies the appropriate handler function to extract and interpret the message on the receiver.

The Active messaging layer is responsible for:

- Integrating communication and computation
- Matching communication primitives to hardware capabilities
- Providing a distributed event model where networked nodes send events to each other

Message contains a user-level handler which is invoked on arrival at the receiver and the data payload passed as argument. Message handlers are executed quickly to prevent network congestion and provide adequate performance. Event-centric nature enables network communication to overlap with sensor-interaction.

Active Message and TinyOS form "Tiny Active Messages" that support three basic primitives: best effort message transmission, addressing and dispatch, [10]. With Active Message every message contains the name of an event handler; the sender declares buffer storage in a frame, names a handler, requests transmission and does completion signal. On the other side receiver's event handler is fired automatically in a target node. So there are no blocked or waiting threads on the receiver and we have a single buffering.

1.4.2.3 Layered model of TinyOS components

A layered model of TinyOS components is shown in figure 1.4 reproduced from [6]. The hardware abstraction layer maps the physical hardware into the component

model. It converts the hardware interrupts to appropriate signaling events and exports the commands to set/reset the individual pin/bus line of hardware component for which it provides abstraction. The next layer represents the communication, sensing and acting component stack. Each of these stacks can have multiple components arranged hierarchically.

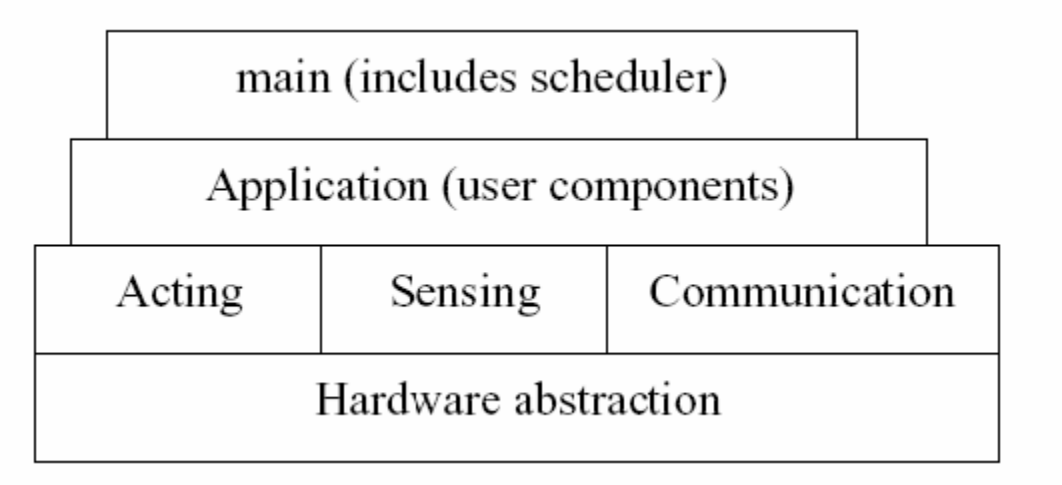


Figure 1.4 – Layered model of TinyOS components, [6]

The application layer has a stack of user defined components for that particular application. And finally at the top, there is the main component which is executed first in any TinyOS application. It initializes the hardware, scheduler, and the application.

1.4.2.4 TinyOS component graph for an application

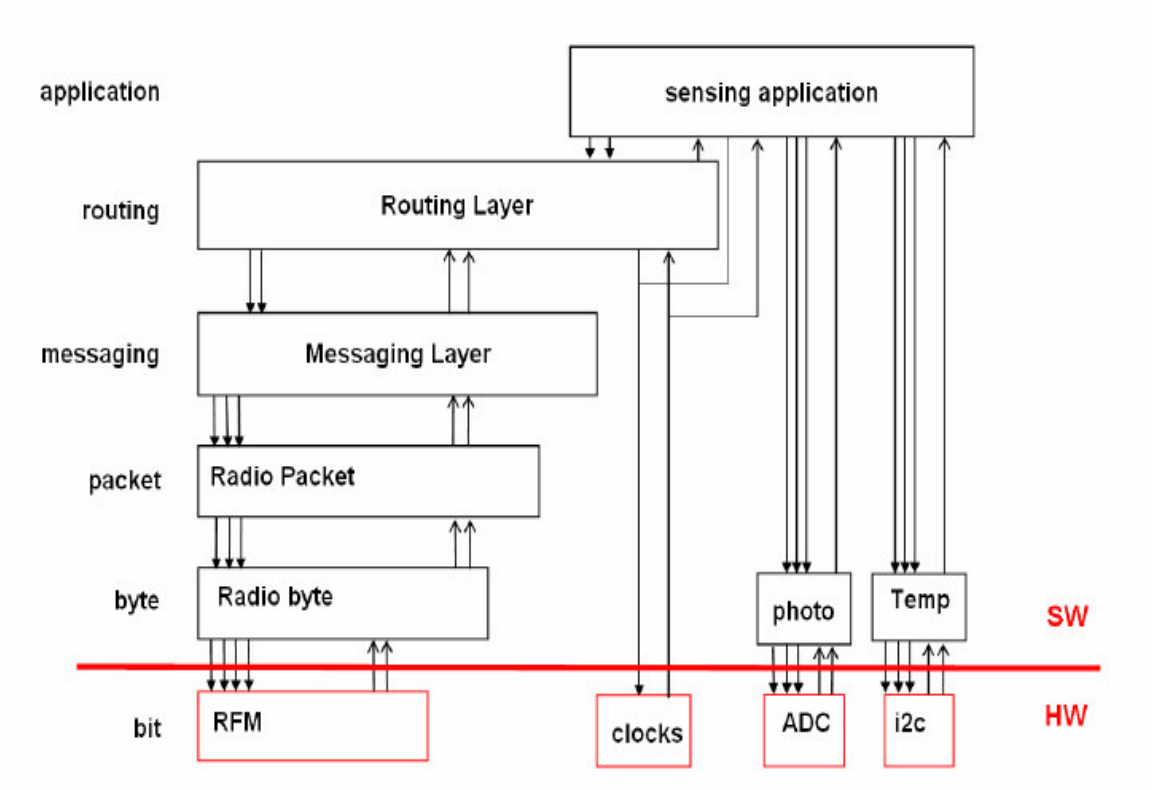


Figure 1.5 – Typical TinyOS component graph for entire application, [6]

Figure 1.5 represents a complete application. The lowest layer of components directly corresponds to the hardware of the system. They simply map the physical hardware into the software based component model. The user application sits at the top of the hierarchy issuing commands down into the lower level components and responding to events propagating up from the system components. During execution, all events are directly or indirectly triggered from the propagation of hardware events up through the component graph. This comes directly from the state machine based programming model, where state changes are the result of changes on the input pins.

1.4.2.5 TinyOS Programming Language

The programming language used for TinyOS is 'nesC' which is a new language for programming structured component-based applications. It is primarily intended for embedded systems such as sensor networks. It has a C like syntax and it supports TinyOS concurrency model, as well as mechanisms for structuring, naming, and linking together software components into robust network embedded systems.

1.4.2.6 TinyOS packet format

Dest (2)	AM (1)	Len (1)	Grp (1)	Data (0..29)	CRC (2)

Figure 1.6 –TinyOS CRC Packet Format, [5]

Figure 1.6 shows a typical TinyOS packet format structure. The first field is the destination address and is two bytes long. AM represents the Active Message Handler type and is one byte long. The third field gives the length of the data payload. The group field is like the network ID and is one byte long. The data payload can be any number of bytes with a maximum limit of 29 bytes. TinyOS sender computes 16-bit Cyclic Redundancy Check (CRC) over the packet to detect transmission errors.

1.5 Problem Statement

As discussed in the previous sections, sensor networks are resource constrained. However, their mission-critical applications need security features. Work has been underway to explore adding security features without straining the very limited resources. TinySec [5] provides the basic security features of Authentication and Encryption, and is one of the successful security protocols adopted in sensor networks. In this study we propose an efficient key update scheme for TinySec keys.

This scheme is not compute intensive, does not add significantly to storage requirements and is applicable to the majority of the sensor network architectures.

1.6 Conclusion

This chapter introduced the wireless sensor networks and gave their types based on network topology and key distribution structure. The hardware and software feature of sensor nodes gives an overview of the application development and execution for the sensor networks.

CHAPTER TWO

BACKGROUND WORK

The background work for this thesis is mainly the link layer security mechanism called TinySec. TinySec was proposed by researchers at UC Berkeley. TinySec is a lightweight and an efficient link-layer security protocol that is adapted to the sensor networks, [5]. It was designed with the goals of achieving the basic security without causing excessive overhead for the resource constrained sensor networks. Further, being a link layer protocol, it is transparent to all TinyOS applications, and thus, has the scope of widespread deployment.

2.1 TinySec

2.1.1 Security Features of TinySec

It provides three basic security features: Access Control, Message Integrity and Message Confidentiality, [5]. Access control and Message integrity are provided by means of Message Authentication Code (MAC). Unauthorized parties are prevented from participating in network communication by means of access control. The nodes can identify the traffic coming from illegitimate nodes and reject it. Message integrity ensures that the message is not tampered with or modified in transit. Both these features are provided by message authentication code (MAC). MAC is the checksum computed on message using a key i.e. cryptographically. The sender computes a MAC over the packet, and sends it with the packet. On receiving the packet, the receiver, re-computes the checksum and compares it with the original MAC; if they are the same it accepts the packet else rejects it. Since the MAC is computed using

the shared key, it provides authentication of the sender. Verification of the MAC provides message integrity.

Message confidentiality refers to keeping the data secret from the adversary, and this is achieved by means of encryption of the plaintext data. The authors decided to use SkipJack encryption algorithm operating in Cipher Block Chaining (CBC) mode. In this mode, encryption is initialized by an Initialization Vector (IV). The IV is changed for each message encrypted to provide semantic security, i.e. the same plaintext is encrypted differently each time. However, identical plaintext produces identical ciphertext for the same key and IV.

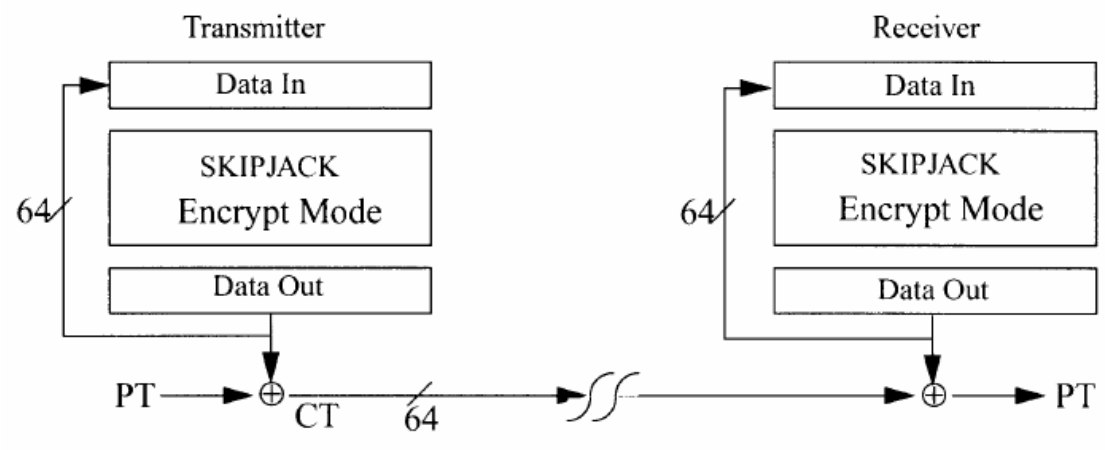


Figure 2.1: Block Diagram SkipJack encryption/Decryption, [11]

Ideally, IV should be unique for every message transmitted, and for this the length of IV should be as large as possible. But due to the resource constrained nature of sensor networks, an 8 byte long IV is selected. The structure of the IV is as follows:

dst | AM | *l* | src | ctr, where:

- **dst** is the destination address of the receiver,
- **AM** is the active message (AM) handler type,
- **l** is the length of the data payload

- **src** is the source address of the sender
- **ctr** is a 16 bit counter; The counter starts at 0, and the sender increases it by 1 after each message sent.

Thus, as we can see a node can send 2^{16} packets without reusing IV. Therefore, to achieve semantic security the key has to be updated. The designers acknowledge the IV reuse problem and suggest that a key update protocol be instituted to exchange new TinySec keys, however this is not their primary focus. We provide a key update scheme that has potential to fill this need and provide semantic security.

2.1.2 TinySec Security Modes

TinySec supports two different security options:

- **Authenticated encryption (AE)** - In this option, TinySec encrypts the data payload and authenticates the packet with a MAC. The MAC is computed over the encrypted data and the packet header.
- **Authentication only (Auth)** – In this option, MAC is computed over the data payload and the packet header.

To distinguish the packets for these modes, TinySec makes use of **first two bits** of the length field. Since, the default maximum data payload of TinyOS is 29 bytes, only the last five bits of the length field are used.

2.1.3 Packet formats for TinySec

As we have seen in the previous chapter (section 1.3.2.6), the default TinyOS packet contains six fields; destination address, AM type, length, group, data payload and CRC. Since TinySec supports two security modes, it has one packet format for each mode. In both these formats, the first three fields i.e. destination address, AM type

and length are same as that of TinyOS packet. Also, the data payload is 29 bytes. However, TinySec replaces the default CRC (2 bytes) of TinyOS with a MAC (4 bytes). Also, the group field is eliminated.

For TinySec AE, 2 bytes of Source address and 2 bytes of counter are added after the length field and before the data field as shown if Fig 2.3.

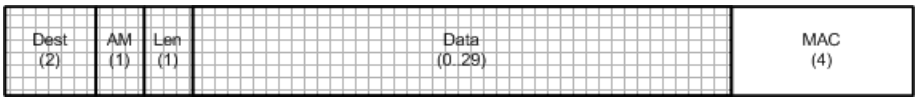


Figure 2.2: TinySec-Auth

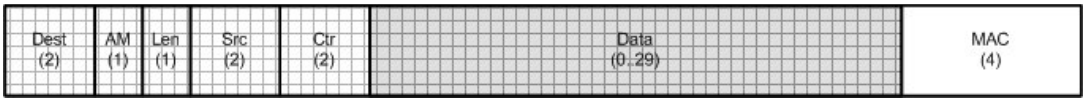


Figure 2.3: TinySec - AE

As we can see from the figures, the packet overhead is as follows:

TinySec - Auth only: +8 Bytes

TinySec -Auth + Encryption: +12 Bytes

Transmission of 29-byte plaintext and its cyclic redundancy check (CRC) requires a packet of 36 bytes. Transmission of that plaintext's ciphertext and MAC under TinySec requires a packet of 41 bytes. Thus, this additional security of TinySec comes at a cost of five extra bytes compared to the original format of TinyOS (Chapter 1, Figure 1.6)

2.1.4 Keying Mechanism

TinySec protocol can work with all the three keying mechanism as described in Chapter 1. There is no limitation for applying it with any keying mechanism.

2.1.5 Implementation and Results

The authors implemented TinySec on Mica, Mica2, and Mica2Dot platforms.

They analytically estimated the costs and also experimentally measured TinySec's performance costs using a variety of microbenchmarks and macrobenchmarks. Their results could be summarized as follows:

Table 2.1: TinySec Overhead Summary

	Packet Size Increase	**Latency Overhead**	**BW Overhead**	**Energy overhead**
CRC (no TinySec)	-	-	-	-
Tinysec - Auth	1.5 %	1.7 %	Negligible	3 %
TinySec - AE	8 %	7.3 %	6 % less thrpt	10 %

TinySec's implementation requires 728 bytes of RAM and 7146 bytes of program space, [5]. TinyOS was required to be modified for implementing TinySec. A two-level priority scheduler was employed in which cryptographic operations were given higher priority and other tasks ran at low priority. They used network wide shared key structure for their implementations and experiments.

2.2 Conclusion

Thus, as we see TinySec was the first attempt to develop security protocol at link layer with detailed specifications. The authors implemented it successfully and it is being used by many researchers as a security platform e.g. companies like SRI, Bosch, BBN, UMass, Intel.

CHAPTER THREE

RELATED WORK

Recently, some researchers have tried to provide key update mechanisms for wireless sensor networks, particularly for MICA2 motes. The approaches of each of these schemes are different and provide a good comparative analysis with respect to resources, complexity, security provisions. There are four main schemes; ECC [12], SPAWKU and SPAGKU [13], LEAP [14] TinyPK [15]. Each of these schemes is described below in detail with its advantages and disadvantages.

3.1 Elliptical Curve Cryptography (ECC)

The authors of [12] emphasize that public key cryptography is viable on sensor nodes, especially Mica2. It can be useful for infrequent distribution of shared secret like the network wide shared key used in TinySec. TinySec IV is 4-byte long, therefore, after 2^{32} packets, it will be reused. This bound may be insufficient for embedded networks whose lifespans demands long-lasting security. Public key infrastructure can help these types of networks to securely re-key themselves.

To address this problem, Malan et al experimented with two public key methods:

- Diffie-Hellman Key exchange based on DLP
- Elliptical Curve Cryptography based on ECDLP

3.1.1 DLP / Diffie-Hellman scheme of key exchange:

Diffie-Hellman is a popular way of key exchange in asymmetric cryptography. In this, two communicating parties, say Alice and Bob, agree on a prime number **p** and a primitive root **g** (i.e. a number between 1 to p-1, also called the generator or base).

Alice chooses a secret integer (private key) **A**, and computes her public key $\mathbf{g^A \bmod p}$. Bob does the same thing, selects an integer **B**, computes $\mathbf{g^B \bmod p}$ Now, both send their public keys to each other. They compute their shared secret using the formula $\mathbf{g^{AB} \bmod p}$. (Since, $\mathbf{(g^A \bmod p)^B \bmod p = (g^B \bmod p)^A \bmod p = g^{AB} \bmod p}$)
Once Alice and Bob compute the shared secret they can use it as an encryption key, known only to them, for sending messages across the same open communications channel.

According to the authors, to generate TinySec key (80 bits), the prime number (**p**) used should be 1024 bits long, and the exponent (private key of each node) used should be 160 bits long.

3.1.1.1 Implementation and Results:

They computed different values for $2^x \bmod p$, x being a 160 bit integer and p being 1024 bit long prime number. Their results measured through instrumentation showed that the time for calculating one exponentiation was 54.9 seconds, and the energy consumed was 1.185 Joules. Two such calculations are required for complete operation. Also, the Memory overhead is ≈11.3 KB of ROM, and 1KB of RAM.
Their argument is that this is too much overhead for resource constrained Mica2 motes. Also, 1024 bits have to be transmitted in more than one TinyOS packets.
(Max payload for one packet = 29 bytes = 29 x 8 = 232 bits, so it would require around 5 packets for transmission of public key -1024 bit long)

3.1.2 ECDLP / Elliptical Curve Cryptography based key distribution:

Of all the cryptosystems known today, Elliptical curve cryptography provides the highest strength-per-bit. As an example, the strength given by 1024 bit RSA keys, is provided by just 163 bit keys in ECC, [12]. The basic scheme works as follows:

Two parties (Alice and Bob) wishing to do a secure communication agree on an Elliptical curve **E** and a generator point **G** on this curve. **E** is defined over $\mathbf{F_q}$, where $\mathbf{F_q}$ is a finite field containing **q** elements. However, in practice, **q** is typically a power of 2 (2^m) or an odd prime number p.

Then, each party selects a random number, **k** (private key of that party). It then computes **k * G** which is its public key and sends that key to another party. The security assumption here is that it is hard to compute the private key k given the public key kG due to the complexity of elliptical curves. The shared secret is then computed by each party, which is the product of one's private key with another's public key. In short,

- Alice chooses k_A, sends $k_A * G$ to Bob.
- Bob chooses k_B, sends $k_B * G$ to Alice.
- Both agree on shared secret = $k_A * k_B * G$ for future communications.

The authors selected the Elliptical curve defined over the field $q = 2^m$ since it allowed the implementation of space and time efficient algorithms. It is also particularly good for hardware implementations. The public key was 163 bits long.

3.1.2.1 Implementation and Results:

Their first implementation attempt, EccM 1.0, was a failure. The module caused resetting of the mote due to stack overflow condition. Their second implementation, EccM 2.0, was a Java based code and they successfully accomplished it.

The time required to compute the public/private key pair was around 34.161 seconds, and the time required to calculate the shared secret, given one's private key and another's public key was 34.173 seconds, [12]. Thus, the approximate computation time required for total key derivation was about one minute and 8 seconds per node. The energy consumed was around 0.9 Joules. The code space (ROM) required was about 34.3 KB, and it consumed around 1 KB of SRAM. (public key is transmitted in two 22 byte payloads = 22 x 8 x 2 = 352 bits).

3.1.3 Analysis:

The authors of ECC did very good work of precisely measuring the time, energy and memory requirements by instrumentation on Mica2 motes. They did this for computing TinySec overhead, DLP and ECDLP modules. However, certain things remain unattended with ECC.

- They claim that ECC is a viable solution; however it is expensive in terms of memory and power.
- Secondly, the public key of the initiator node is broadcasted in two 22 byte payloads. So, for this scheme to be efficient, it is imperative that both the packets of public key reach the other nodes in sequence without any packet losses.
- Further, network wide impact is not considered. e.g. Node 1 (Alice) broadcasts her public key, and is received by one or multiple nodes. But, how is the scenario

in reverse direction handled in case of network-wide key? There are two possibilities:

- If this is between two nodes only, then each node will need to do the computation for each of its neighbor, and what is the guarantee of shared secret being the same for all neighbors?
- If multiple nodes are targeted by Alice, then whose public key will be considered to compute the shared secret? Since k is selected randomly, so all neighbors will have different k values.

Admittedly sensor nodes of the future might successfully employ this key updating scheme, but where the thrust is miniaturization the scheme's reception might remain low, more so because of the transmission update overhead.

3.2 SPAWKU and SPAGKU Key Update Protocols

The author of [13] describes two different key update protocols for dense sensor networks having network-wide shared key and using link layer security like TinySec. The main assumption of this research is that each node is preloaded with two keys:

- Interchange key – used only for key update of session key
- Session key – used for all other communication purposes

Further it is assumed that due to dense distribution of network, there exist multiple paths between any pair of sensor nodes.

The two protocols are described as follows:

3.2.1 Sequenced pair-wise key update protocol (SPAWKU)

This protocol gives the mechanism to update key between any pair of nodes. For global key update, it is assumed that an algorithm on top decides key update pairs for

complete network and this algorithm is based on network topology. e.g. cluster heads, spanning tree. The scalability of the protocol depends on the underlying network topology on which it is based.

Protocol Description:

The handshaking for key update is done with 4 types of messages. Each message is 15 bytes long (15 x 4 = 60 bytes overhead). All messages have the same structure as shown in the figure below. Further, the key used for encryption is written with subscripts i.e. K_s is session key and K_i is interchange key.

Figure 3.1: SPAWKU and SPAGKU - Key Update Packet Format

The key update takes place as follows:

- First, the initiator node sends the Key Update Request (KUR) packet. Its format is (KUR, SN, RAND)K_s, MAC.
- The receiver node replies with Key Update Request Ack (KURA) packet. Its format is (KURA, SN, RAND)K_s, MAC.
- Then the initiator sends the new session key encrypted with the interchange key by using Key Update (KU) packet. Its format is (KU, SN,Key)K_i, MAC.
- And, finally the process is completed with the receiver sending the Key Update Ack (KUA) packet. Format is (KUA, SN,Key) K_i, MAC.

Analysis:

- The protocol does not guarantee key update in case of packet losses.

- The protocol does not promote load balancing; i.e. some nodes may be involved in more key update transactions than others.

Further more, the author does not provide any implementation or experimental results about this protocol.

3.2.2 Sequenced Partial Global key update protocol(SPAGKU)

This protocol takes the advantage of the redundant deployment in sensor networks. It treats the node without the most recent updated session key as temporarily disabled node. The protocol assures that a large fraction of network (and not the complete network) is updated with the new session key. The key update process takes place in following two steps:

- First, the initiator (external micro-server) broadcasts a request message to update the session key. All the nodes change to their interchange key.
- Then it sends the second message containing the new session key encrypted with the interchange key. All the nodes update their session key.

Packet format for the above process is same as that for the previous protocol, however only 2 packets (KUR and KU) are sent in this case.

3.2.2.1 Implementation and Results:

According to the author, both these protocols were tested using TOSSIM and Mica2 motes. In terms of measurements/results, the code size required for both these protocols is quoted. The code size is measured without considering storage for reused modules. Table 3.1 gives the memory requirements quoted.

Table 3.1: SPAWKU and SPAGKU - Memory requirements

	SPAWKU	SPAGKU
Code Size in Bytes	2062	1932
Memory Footprint in Bytes	81	80

The author did the general evaluation of the 2nd protocol by measuring the consistency ratio (Number of nodes with updated key/total number of nodes in the network) as a function of network size (9–900 nodes), network density (100 node network, spacing increased from 1-40 feet), and network traffic (packet load (20-200 byte packets) and packet frequency). His graphs based on Nido (TOSSIM) simulations show that the network maintains a consistency ratio of 1 in most of the cases.

However, he does not perform encryption and decryption along with this (revealed from the fact that they increase the packet load from 20 bytes to 200 bytes).

Analysis:

- Key-aging problem is solved, but the security is not enhanced. All nodes have the same interchange key, so compromise of a single node will reveal the key and further updates of session key will have no meaning. Anyone knowing the interchange key can decrypt the packet sent with new session key.
- The partial global key update protocol makes the assumption that the number of nodes updated is sufficient enough to maintain the network functionality. This assumption seems to unrealistic.

3.3 Localized Encryption and Authentication Protocol (LEAP)

The authors of LEAP, [14], propose a key management protocol in which each sensor node has four different keys for different security requirements. These keys are: an *Individual* key shared with the base station; a *pairwise* key shared with each neighboring sensor node; a *Cluster* key shared with multiple neighboring nodes; a *group* key shared by all the nodes in the network. LEAP provides schemes to establish and update all of these keys.

3.3.1 Establishing all keys

Individual keys: The base station generates it, and pre-loads it into the node. Master key, K_m is only known to the base station and all the individual keys are derived from it. Pairwise shared keys- All the nodes are also pre-loaded with initial key K_I which is used by the nodes to derive their respective master keys and also the master key for other nodes. Each node discovers its neighbors, generates the neighbor's master key, and then generates the pairwise key it shares with this neighbor. This is done in initial time T_{min} for each node. The key is actually not transmitted.

For sleeping neighboring nodes, it gives an alternative which is complex and computationally intensive. (to obtain the list of working nodes from the neighbors). Overhead – Derive neighbor's master key, verify MAC, compute pairwise key. Both nodes do this. The node deletes the key of neighbor when it detects that it is compromised. Cluster Key- One node generates key, encrypts it with pairwise keys, and sends to its neighbors. Group Key- It is also pre-loaded in each node.

3.3.2 Updating Keys

The focus is on updating the group and cluster keys.

Cluster key Update:

When one of the neighbors of a node is revoked, the node generates a new cluster key and transmits it to the remaining neighbors. It uses the pairwise key for encrypting the cluster key to be sent to each neighbor.

Group Key Update:

Group re-keying implies updating the key for the entire network and is considered one of the most difficult tasks by the authors. They claim to do this task in a unique and most secure way. The group key will be updated when any node in the network is compromised.

The scheme involves two stages:

Authenticated Node Revocation: Since the base station can never be compromised (one of the assumptions), it is the appropriate entity to announce the node revocation. Further, its announcement must be authenticated. The authors use μTESLA for this purpose. The first key (commitment/seed) of the key chain is pre-loaded in all the nodes prior to deployment. The revocation message consist of node name (id) for the node to be revoked, the seed/verification key for the new group key, the μTESLA disclosure key and the MAC. Each node stores the message for one μTESLA interval, receives the MAC key, and verifies the authenticity of base station. If the verification is successful, it stores verification key for the new group key. Further, if the revocation node indicated is one of its neighbors, it deletes the pairwise key for that node and updates the cluster key.

Secure Key Distribution: They assume the existence of a suitable routing protocol, like TinyOS beaconing protocol for key distribution sequence. The base station sends

the new group key to all its children in the spanning tree using the cluster key for encryption. This process continues until the key is distributed to all the legitimate nodes in the network.

3.3.3 Implementation and Results:

The authors implemented LEAP on TinyOS platform using RC5 block cipher for CBC-MAC and encryption. They just give the memory overhead. The ROM space required is 17.9KB for their code and the RAM space depends on the number of neighbors for an individual node as shown in the table below.

Table 3.2: RAM requirements as a function of number of neighbors *d*

d	1	5	10	15	20	25	30
RAM (bytes)	600	736	906	1076	1246	1416	1586

In their performance evaluation section, the authors give the approximate mathematical formulas for computation, communication and memory needs (costs) for LEAP. They consider the cost only for updating the cluster keys and group key. According to them, generating the key using pseudo-random generator is a negligible overhead.

Computational cost: For updating Cluster keys, the number of encryptions required is $Se = \sum_{i=1}^{d_0} d_i$; where $\mathbf{d_0}$ is the number of neighbors revoked and $\mathbf{d_i}$ is the number of legitimate neighbors for each for these $\mathbf{d_0}$ nodes. Overall, for a network consisting of **N** nodes, the average number of operations performed by a node is **2Se/N**.

For updating group keys, the number of decryptions is equal to network size **N**. Since, each parent has to encrypt only once for all its children, maximum possible encryptions is also **N**. Thus, maximum operations required are **2N**.

Communication cost: For cluster key update, the communication cost is $(d-1)^2$ /(N-1) for a network of degree (maximum number of neighbors for each node) d and size N. For group key update, it is 2N.

Storage Cost: For each node, the memory (RAM) requirements will depend on the number of neighbors and it is equal to **3d + 2 + L**; where d is the number of neighbors a node has and L is the length of node's one way key chain.

3.3.4 Analysis:

LEAP is an extensive, robust and an excellent protocol for key establishment and key update. It considers security from different perspectives like routing, key management, node compromise, etc. It attempts to consider all possible situations when establishing or updating a key. However, there are some issues as described below.

- Their assumption that the nodes have memory to store hundreds of bytes of keying materials is kind of impractical. Sensor nodes are highly resource constrained, and will remain that way for the foreseeable future; this fact makes this assumption somewhat unrealistic.
- There is ambiguity about the node revocation. They neither make any assumption regarding how the base station will come to know of malicious/compromised node nor do they state explicitly something about it.
- Their overhead requirements ignore certain things, like the costs for μTESLA in group key update mechanism.

- In their implementation on TinyOS, they don't specify any details about the type of network, or the number of nodes in the network, network topology, etc. Further, no mention of whether they tested it for node revocation or not.
- There is inconsistency in the RAM values quoted.
- In pairwise key establishment, handling the sleeping nodes is complex and involves more computation which they never account for. Also, for nodes which are added later, the key establishment seems contradictory. As per their explanation, pairwise key is computed by both the nodes and not exchanged. For computing it, the node needs to derive the master key of neighbor from the initial key K_I. Once this is done, the nodes delete K_I and the master key for all the neighbors. Now, for lately added node, they say that it can establish pairwise key with the node which has erased K_I.

3.4 TinyPK

The main focus of this research was the design and implementation of public key based protocols for authentication and key agreement between the sensor network and a third party as well as between two sensor networks.

TinyPK design has a RSA based public-key infrastructure. There is a Certification Authority (CA), which is an entity with a private and public key pair that is trusted (or can establish an authenticated chain to a trusted entity), by all friendly units. Any third party that wishes to interact with the motes also requires its own public/private key pair and must have its public key signed (not on a hash of the data, but by transforming the data directly) by the CA's private key for establishing its identity, [15]. Also, as each mote is loaded with software before being deployed to

the field, it must have the CA's public key installed. TinyPK eliminates certificates due to lack of computational power for sensor networks.

TinyPK has a challenge-response protocol which performs two functions:

- authenticates the external party to the sensor network
- Securely transfers a session key from the sensor network to the third party.

3.4.1 Implementation and Results:

This whole design was done based on RSA cryptosystem using e=3 as public exponent. Their implementation involved performing private operations on PC and public operations on motes. Private operations take tens of minutes on motes, and thus their implementation of RSA scheme can be considered partial.

They also tried Diffie-Hellman (DH) key exchange on Mica2 motes to generate the key which would serve as an equivalent replacement for TinySec keys and can be used to create new TinySec keys. The basic scheme/mechanism of DH key exchange is same as explained in section 3.1.1. They could do it successfully, although the computational time and memory requirements were high.

The generator used by them is g = 2, and they showed the graphs for execution time versus the exponent size for three lengths of prime number p; 512 bits, 768 bits and 1024 bits. There are two graphs showing the results for both, the first and the second exponentiation.

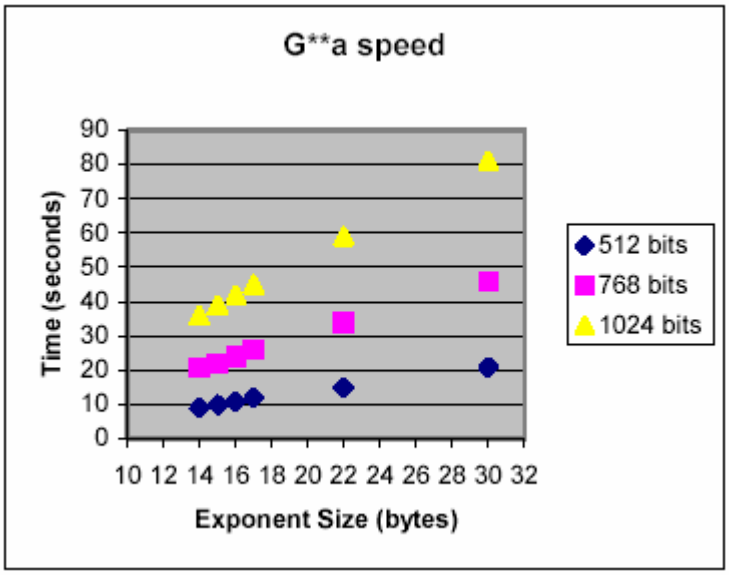


Figure 3.2: Execution time for first exponentiation

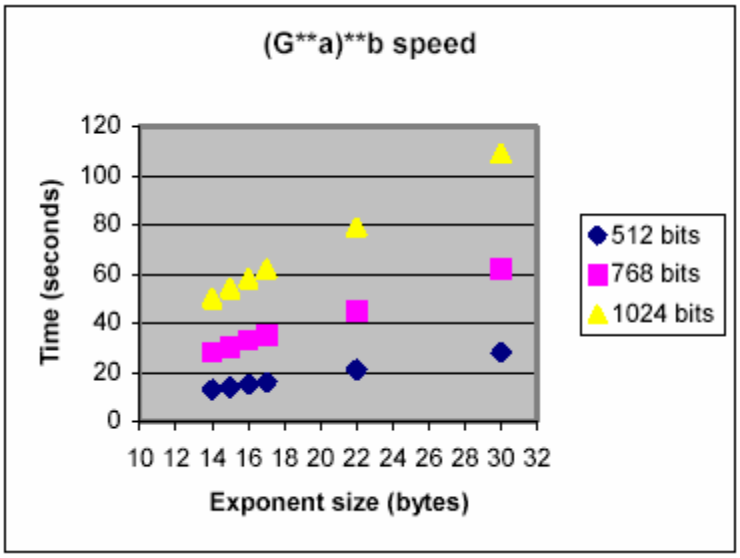


Figure 3.3: Execution time for second exponentiation

Further, the ROM bytes needed are around 12KB, and 1 KB of RAM bytes are needed.

Table 3.3: Memory requirements for Diffie-Hellman Key Exchange

	Modulus Size		
	512	768	1024
ROM (bytes)	12340	12376	12408
RAM (bytes)	847	1007	1167

This research is far from our concern, however we mention it here to emphasize that public key cryptography is very expensive for resource constrained sensor networks.

3.5 Conclusion

Thus, as we see, asymmetric cryptography is very expensive for resource constrained sensor nodes. LEAP is a good scheme but it is highly complex. Note that none of these schemes are implemented along with TinySec.

CHAPTER FOUR

PROPOSED SCHEME

As explained in chapter two, the encryption scheme of TinySec uses an initialization vector (IV) which is 8 bytes long. The structure of the IV is dst | AM | *l* | src | ctr where dst stands for destination address, AM represents the active message type, *l* is the length, src is the source address and ctr is the counter value. So, ideally speaking, for a particular node, the IV will be repeated after it has sent 2^{32} packets. However, we see that the source field remains constant always since it is the sending node's address. The fields like destination, AM type, and length may or may not change depending on the application and the node's position in the network with respect to other nodes, e.g. if the node is periodically sending a reading to the group head/data aggregation point or broadcasting it then the destination field will remain constant. Also, the AM type would be the same. The length could be constant if the sensor is designed to give fixed length reading. Conclusively, the field which is sure to be changed every time is the 2 byte counter field, irrespective of other fields. Thus, as we can see a node can send 2^{16} packets without reusing IV. Therefore, in order to achieve semantic security the key has to be changed. Hence, the need for key update.

ECC, LEAP and SPAKGU are good attempts to provide this functionality. As discussed in chapter three, each scheme has its own pros and cons.

In the next section, I propose a different key update scheme for updating the encryption key of TinySec. In chapter five, the results of implementation of this scheme are reported.

4.1 Proposed Algorithm

As we have seen in the first two chapters, the maximum data payload length for TinyOS packet is 29 bytes. This length can be encoded using 5 bits (2^5 = 32). TinySec uses the first two bits of length field to encode the two TinySec modes namely:

TinySec-Auth – 10 (Auth => Authentication)

TinySec -AE – 11 (AE => Authentication and encryption)

The third bit is still left unused. I will utilize this bit for my key update scheme.

Whenever the time for key update comes, the base station or the node wishing to do a key update first sets this third bit. It then rotates the existing encryption key. It encrypts the data with this new key and sends the packet. The receiver node(s), first decodes the length byte as it does for TinySec. It determines the TinySec mode from the first two bits and the status of the third bit indicates whether the key has to be modified or not. It will consider this status only if the mode is TinySec-AE. If the bit is set, the receiver will first rotate its encryption key, and then decrypt the data. In this way, we achieve two things at the same time; the synchronization and key update.

Steps for Key Rotation

- The 8 byte key is separated into two 4 byte parts (blocks)
- Each part i.e. 4 byte block is rotated to left by 1 bit
- The MSB of one block is inserted as LSB for another block

Since TinyOS follows little endian format for the memory structure, the resulting key is not just a shifted version of the original key, but its kind of random value.

4.2 Cost Analysis

The proposed algorithm is very efficient and simple. The key update does not involve generating a new key. Also, the sender need not send any key update message prior to key update. This purpose is served by setting the third bit at the time for key update. Further, no key is sent in the message since the key update is achieved by rotation of bits of the already existing key. In this way, the overhead of sending the key update message and the key are avoided. There is no bandwidth overhead at all. The latency overhead is also avoided since the packet size remains the same as that of TinySec and so no additional time is required to send the key update request or the key. In terms of computational time and energy, the operations required in the whole process are as follows:

- Setting the bit in the length field
- Performing circular rotation

These operations take very few instructions. Thus, the computational time and the power required are minimal.

Conclusively, bandwidth, latency and computational overheads per node are very minimal.

4.3 Security Analysis

Since we follow the TinySec model, our security provisions are at the same level as TinySec. We are enhancing the TinySec security level, in the sense that we are providing a better encryption security by means of key update method to prevent the IV reuse. Since TinySec is a link layer mechanism, it guarantees the authenticity, integrity and confidentiality of the messages between the neighboring nodes, while

permitting in-network processing, [5]. As in TinySec, we do not address resource consumption attacks, node capture attacks, and replay attacks. The keying mechanism applied by the application will decide whether these attacks are counteracted or not. e.g. Pair-wise key structure is robust against individual node capture, but network-wide is not. TinySec, and so does our scheme, are applicable with both the above keying mechanisms and so node capture attack is not handled by this scheme.

In TinySec, a 16 byte key is preloaded into the motes. The first 8 bytes are used for encryption key and the next 8 bytes are used for MAC key. These bytes are copied to their respective buffers. Since there are only 8 bytes, there is a limitation on the number of different/unique keys possible. The number of different keys possible is 64. We consider this number to be good enough for sensor networks. The best case would imply using the key update sparingly, for example, updating key once per day the scheme would enable for 64 days before the key repeats. However, if there is a need to overcome this limitation, then we can do that by swapping any two neighboring bits once the key update count has reached 64, and then again rotate the key 64 times.

4.3.1 Key Quality Analysis

For any security mechanism, the value of the key plays an important role in providing security strength. Key size and randomness of the key value are the major factors influencing the key quality. A key length of 80 bits is generally considered the minimum for strong security with symmetric encryption algorithms. TinySec takes the 64 bit value and expands it to 80 bits before performing encryption as required in SkipJack Block Cipher. Regarding the key value, according to the Report [16], there

exist no key value which can be considered as weak key for SkipJack cipher. Even the key with all ‘0’s or all ‘1’s is not weak because of the design of SkipJack algorithm. There is no pattern of symmetry in the SKIPJACK algorithm which could lead to weak keys.

In the proposed scheme the key is rotated circularly. Depending on the initial value of the preloaded key, the updated key will have various combinations of ‘1’s and ‘0’s. However, if we go by the analysis report of [16], any key value will not reduce the strength of encryption/decryption. Further, the number of ‘1’s and ‘0’s present in the original key and the updated key remains same due to rotation.

4.3.1.1 Rotation versus other simple operations

Besides rotation, if we consider other options like incrementing the key by 1 for key update, the resources consumed will be the same since the Atmega processor takes one cycle for any instruction. However, depending on the initial key value, after some updates the key value will become all ‘1’s. So, we need to store the initial value of the key somewhere to prevent the overflow. In spite of this, we can consider increment operation as an option since SkipJack cannot have weak keys. Similarly, we can consider any other simple arithmetic operation for updating the key as far as the key strength is concerned.

4.4 Applicability to WSN Models based on Topology

Section 1.1 describes two types of sensor networks based on their topology. The following subsections describe how the key update will be implemented network-wide.

4.4.1 Hierarchical/ Infrastructure based wireless sensor networks

In these networks, the time for key update will be decided by the base station. Since the base station is a trusted authority, it is an appropriate entity to take the decision regarding the timing for key update. It can update key periodically, once a day or decide the time based on some mathematical formula which is known only to it. Whenever, it wants to do the key update, it will send the packet with the third bit of length set and the data encrypted with new (shifted) key. The propagation of key update request within the network will be done based on the key distribution structure of the network.

4.4.1.1 Network-wide shared key structure

This is the simplest network key structure. In this case the base station will broadcast the key update request packet and since its transmission range covers all the nodes, it implies that all the nodes in the network update their key at the same time. So, the henceforth communication between all the nodes is done with the new encryption key.

This key update scheme can provide a partial protection against the node capture attack in the situation that a compromised node removed from the network by the adversary for some time and placed again in the network. If this time is larger than the key update interval determined by the base station, then the adversary will not know the key. The base station is the only one who knows the key update interval and so even after knowing the key update mechanism there are chances that the adversary will not be able to detect the current key.

4.4.1.2 Group-wise shared key structure

In this type of network, we can do the key update with the help of routing protocol or by following the hierarchy. Again, the base station will determine the time for key update. It will update the keys with its neighboring group heads. The group heads in turn will send the key update request to their group nodes and their neighboring group heads. This process continues till all the nodes update their respective keys. Note that first the group head will modify its key based on the message received from the base station. Then, it will send the key update request to all the group nodes belonging to his group with the third bit set, but no shifting operation since it has already done it when it received the key update request. We assume that a group head shares a pair-wise key with all its neighbors for inter-group communication. So, it updates this key and then sends the key update request to that respective neighbor. The neighbor in turn will update its pair-wise key shared with the sender, update its group key and then send a request to its group nodes for the key update.

4.4.1.3 Pair-wise shared key structure

This structure is the most robust one and does not need key update on regular basis since every node has to decrypt the message received and re-encrypt it with the key for next node. Nevertheless, the scheme could be applied if needed with just two neighbors involved in the key update.

4.4.2 Distributed Wireless sensor networks

In these networks, there is no base station but there is one or more node(s) which act as the gateway or the access point for human interface to send the readings/data/information collected from the entire network. We assume that this

node will be given the instruction by the application controller for the key update. This node will, then, broadcast/multicast the key update request after updating its own key to all nodes within its range. They would in turn, update their key and propagate the request further. The issue here could be the time required to update the key for the entire network, especially in the case of network-wide shared key. This time will depend on the size of the network.

4.5 Conclusion

We proposed an efficient and resource aware key update scheme which consumes very minimal computational resources. This scheme is developed with TinySec as the reference security protocol. It enhances the confidentiality provided by TinySec. This scheme is more apt for hierarchical networks as compared to pure distributed ones.

CHAPTER FIVE

IMPLEMENTATION

To test the algorithm presented in chapter four, we implemented it with TOSSIM - the simulator for TinyOS, [8]. TOSSIM is a simulator specifically designed for sensor network running TinyOS; an operating system specifically designed for resource constrained sensor nodes. TOSSIM is a de-facto for testing, debugging, and analyzing TinyOS applications.

5.1 TOSSIM

TOSSIM, [17] provides a scalable simulation environment for sensor networks based on TinyOS. Unlike machine-level simulators, TOSSIM compiles a TinyOS application into a native executable that runs on the simulation host. This design allows TOSSIM to be extremely scalable, supporting thousands of simulated nodes. Deriving the simulation from the same code that runs on real hardware greatly simplifies the development process. TOSSIM supports several realistic radio-propagation models and has been validated against real deployments for several applications, [18].

In TOSSIM, the TinyOS application is compiled directly into an event-driven simulator that runs on the simulation host. This design exploits the component-oriented nature of TinyOS by effectively providing drop-in replacements for the TinyOS components that access hardware; TOSSIM provides simulated hardware components such as a simple radio stack, sensors, and other peripherals. This design allows the same code that is run on real hardware to be tested in simulation at scale.

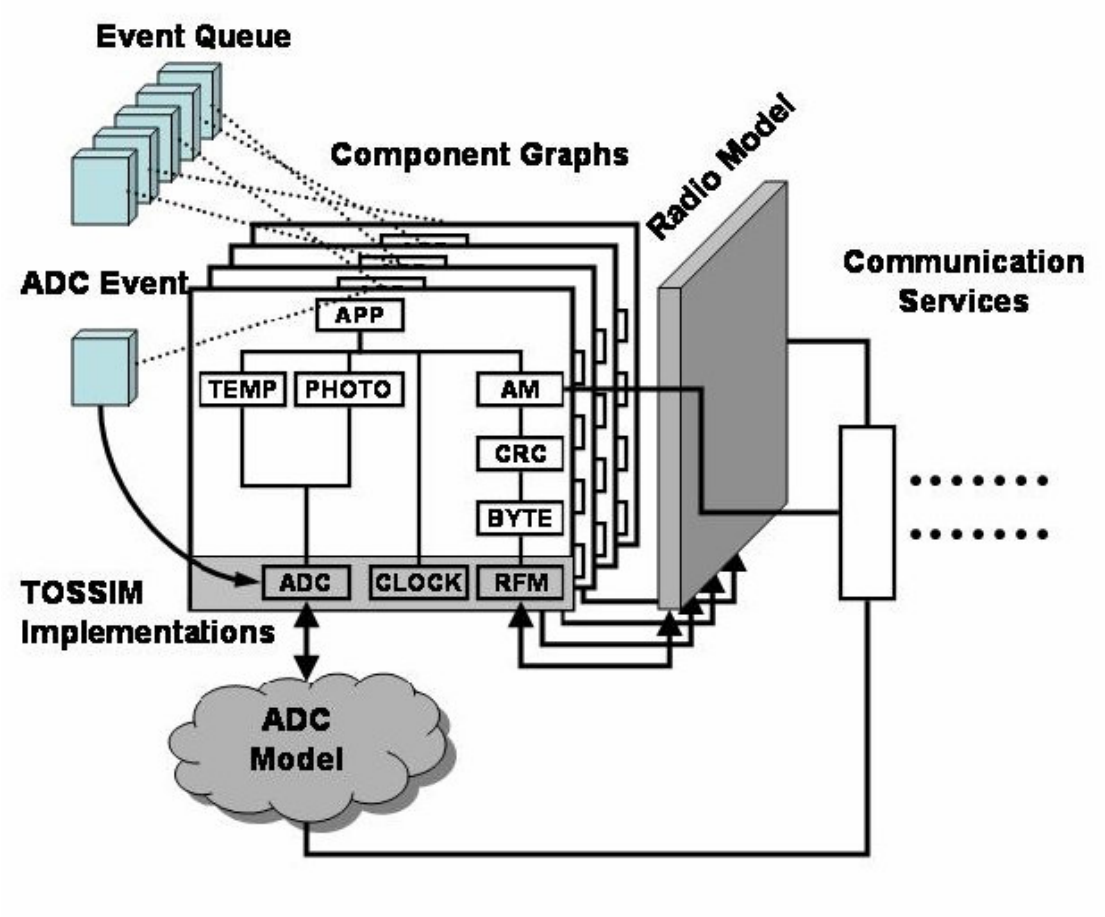


Figure 5.1: TOSSIM Architecture, [17]

TOSSIM captures the behavior and interactions of networks of thousands of TinyOS motes at network bit granularity. Figure 5.1 shows a graphical overview of TOSSIM. The TOSSIM architecture is composed of five parts:

- Support for compiling TinyOS component graphs into the simulation infrastructure
- A discrete event queue
- A small number of re-implemented TinyOS hardware abstraction components
- Mechanisms for extensible radio and ADC models
- Communication services for external programs to interact with a simulation

TOSSIM takes advantage of TinyOS structure and whole system compilation to generate discrete-event simulations directly from TinyOS component graphs. By replacing a few low-level components (e.g., those shaded in Figure 5.1), TOSSIM translates hardware interrupts into discrete simulator events; the simulator event queue delivers the interrupts that drive the execution of TinyOS application. The remainder of the code runs unchanged, [17].

TinyOS abstracts each hardware resource as a component. By replacing a small number of these components, TOSSIM emulates the behavior of the underlying raw hardware. These include the Analog-to-Digital Converter (ADC), the Clock, the transmit strength variable potentiometer, the EEPROM, the boot sequence component, and several of the components in the radio stack. The low level components that abstract sensors or actuators also provide the connection point for the simulated environment.

The nesC compiler (ncc) is modified to support compilation from TinyOS component graphs into the simulator framework. With the change of a compiler option, an application can be compiled for simulation instead of mote hardware, and vice versa.

TOSSIM provides run-time configurable debugging output, allowing a user to examine the execution of an application from different perspectives without needing to recompile. TOSSIM also incorporates TinyViz, a Java-based GUI that allows for visualization and control of the simulation as it runs, inspecting debug messages, radio and UART packets, and so forth. It has a set of plugins like debug messages, radio model, ADC readings, etc which provide the desired functionality. A TinyViz

plugin is a software module that watches for events coming from the simulation and reacts by drawing information on the display, setting simulation parameters or actuating the simulation itself, [19]. Plugins can be selectively enabled or disabled depending on what information is required during simulation. e.g. we can set the ADC readings by activating the ADC plugin.

5.1.1 PowerTOSSIM

Although TOSSIM captures TinyOS behavior at very low level, it does not model power consumption for motes. This is because it does not model CPU execution time, and thus, cannot provide accurate information for calculating CPU energy consumption, [18]. The authors of [19] designed a tool called *PowerTOSSIM* to measure the CPU cycles and power consumption for a particular node running a specific application. PowerTOSSIM generates an event-driven simulator directly from TinyOS code and emits power state transitions for multiple hardware peripherals (radio, sensors, LEDs, etc.). In addition, PowerTOSSIM obtains an accurate estimate of CPU cycle counts for each mote by measuring basic block execution counts and mapping each basic block to microcontroller instructions. PowerTOSSIM obtains very accurate power consumption results for a wide range of TinyOS applications and exhibits very little overhead above that of the TOSSIM environment upon which it is based. PowerTOSSIM's accuracy for power measurement is 0.45-13% of true power consumed by nodes running identical application program, [19]. The power is modeled with respect to Mica2 Energy model. This tool is integrated in all the version of TinyOS after version 1.1.9.

5.2 Compiling and executing an application with TOSSIM

For compiling an application for TOSSIM, we have to use 'make pc' after entering into an application directory. The TOSSIM executable is called main.exe and it resides in build/pc directory within the application directory. This main.exe file is run with various usage options for the required network parameters. The compulsory parameter for TOSSIM to run is the number of nodes to be simulated. All the other parameters are optional. We can specify the simulation time, enable power measurement, select ADC model, select the radio model, etc.

TinyOS source code contains a lot of debugging statements which are displayed in the command window during the execution/simulation run. By default, TOSSIM prints out all the debugging information which is huge. So, we can configure the TOSSIM output instead by setting the DBG environment variable in the shell. e.g. export DBG=leds will just display the debugging statements for LEDS. DBG option crypto displays the TinySec specific messages and the option power is used for displaying power related information. We can add our own debugging statements in the TinyOS code using options like usr1,usr2, usr3 and temp. For measuring power, there are a set of special powerTOSSIM instructions which gives the total energy consumed by each simulated mote, and the CPU cycles required for total runtime.

5.3 Algorithm Implementation and Results

I used TinyOS 1.1.14 which is the latest version of TinyOS for implementation.

The component TinySecM.nc, present in the library, is the main component for TinySec implementation. It contains the code for cryptographic operations. I

manipulated this file to include the key update code. TinySecMode interface contains commands to specify the TinySec transmission and reception modes. I added new command to set the key update mode.

I used three different applications to get the results for memory overhead, CPU cycles, power consumption and network behavior for my scheme. The main thrust of my results is to show the efficiency of my algorithm in terms of resource consumption on a single node.

5.3.1 Memory overhead measurement

I used the application called TestTinySec as reference for memory overhead evaluation. This application is pre-built in TinyOS and comes with its download package. The simulation results with debugging option crypto shows the encryption and MAC details.

To get the memory size for actual motes, I compiled the application with mica2 option. The compilation output shows the ROM and RAM sizes for Mica2 executable. I compiled the original TestTinySec application before and after adding my code. The percentage increase in memory is as shown in table 5.1 below.

Table 5.1: Memory Overhead – Proposed Algorithm

Memory	% Increase
ROM	1.66%
RAM	0.347%

Thus, memory overhead for my scheme is minimal.

5.3.2 Power and CPU cycle measurement

To measure these two parameters, I used powerTOSSIM. PowerTOSSIM rely on some other intermediate tools like CIL (C Intermediate Language) and OCaml

(Objective Caml) for computing CPU cycles. CIL library code and OCaml source code are needed to get the CPU cycle results.

5.3.2.1 Power Measurement

To measure the amount of power consumed by key update process, I simulated the TestTinySec application without invoking key update and then by invoking key update periodically. However, the resulting values were same. I could not get any difference in the power values. One reason for this could be that my algorithm takes very little power and powerTOSSIM cannot capture it. Another possibility is that powerTOSSIM ignores the dynamic runtime behavior. To find the exact cause, I measured power for two modes (AE and Auth_only) of TinySec itself without involving key update. Encryption operation is expected to consume a significant amount of power and so, the power difference is expected definitely. However, I got almost the same values for both the modes of TinySec. There was a little difference, but it was due to radio power.

Table 5.2: Power measurement results

AE mode – 180 seconds, 2 motes	**Auth_only – 180 seconds, 2 motes**
Mote 0, cpu total: 2215.837732	**Mote 0, cpu total: 2215.807996**
Mote 0, radio total: 3821.707598	Mote 0, radio total: 3814.067606
Mote 0, adc total: 0.000000	Mote 0, adc total: 0.000000
Mote 0, leds total: 1177.586503	Mote 0, leds total: 1177.586503
Mote 0, sensor total: 370.200493	Mote 0, sensor total: 370.195525
Mote 0, eeprom total: 0.000000	Mote 0, eeprom total: 0.000000
Mote 0, cpu_cycle total: 0.000000	Mote 0, cpu_cycle total: 0.000000
Mote 0, Total energy: 7585.332327	*Mote 0, Total energy: 7577.657631*
Mote 1, cpu total: 2208.342711	**Mote 1, cpu total: 2208.312975**
Mote 1, radio total: 3808.821858	Mote 1, radio total: 3801.202602
Mote 1, adc total: 0.000000	Mote 1, adc total: 0.000000
Mote 1, leds total: 1175.987241	Mote 1, leds total: 1175.987241
Mote 1, sensor total: 368.948298	Mote 1, sensor total: 368.943330

Mote 1, eeprom total: 0.000000 Mote 1, cpu_cycle total: 0.000000 *Mote 1, Total energy: 7562.100108*	Mote 1, eeprom total: 0.000000 Mote 1, cpu_cycle total: 0.000000 *Mote 1, Total energy: 7554.446148*

As revealed from table 5.2, PowerTOSSIM will not be helpful for measuring power consumption for my scheme.

5.3.2.1 CPU cycle Measurement

For this also, I followed the same procedure as done for power measurement. Here also, the results were on similar lines. There was no difference between the CPU cycle counts for the two modes of TinySec.

Table 5.3: CPU cycle measurement results

AE – 180 seconds, 2 motes	**Auth_only – 180 seconds, 2 motes**
Mote 0 CPU_CYCLES 126229.5 at 720000029 Mote 1 CPU_CYCLES 125966.5 at 720000029	Mote 0 CPU_CYCLES 126229.5 at 7200000291 Mote 1 CPU_CYCLES 125966.5 at 720000029

Since the results were same, I analyzed the way powerTOSSIM computes CPU cycle. As a result of my analysis and trials, I concluded that powerTOSSIM measures the CPU cycles based on the compile time information it gathers. For applications which uses TinySec, looks like it is unable to capture correct power. To further verify, I looked into PowerTOSSIM paper, [18] where they have tabulated the power measurements for many build-in applications. Unfortunately, TestTinySec is not listed there. This might be because they are unable to capture the power calculation of applications which use TinySec.

To measure the CPU cycle count for my algorithm, I made a new application with just the code involved in key update. This code is put into a task. The task is called every second when the timer fires. To get the difference, I compiled and ran the application for two cases:

1) Measuring CPU cycle count without calling the task
2) Measuring CPU cycle count by calling the task at each timer firing event.

Table 5.4: CPU cycle measurement for key update task

CPU cycle count with KeyUpdate Task	**CPU cycle count without KeyUpdate Task**
CPU_CYCLES **10310.0** at 239401694	CPU_CYCLES **991.5** at 239401694

Since the simulation is run for 60 seconds, the task is executed 60 times.

CPU cycles = (10310 – 991.5) / 60 **= 155.308 cycles.**

The number of CPU cycles required for this operation is comparatively less than other cryptographic operations, for example, the encryption which takes 103756 cycles. This shows the efficiency of this scheme.

5.3.3 Network Behavior for this algorithm

To simulate the network behavior, I designed a new application having multiple nodes and a base station. TOSSIM programs all the simulated nodes with the same code (both for base station and normal node) but they could be distinguished at runtime by their node number or node ID. Here **node 0** is the base station and all the other nodes sense the environment and send their readings to the base station. Each node has two sensors attached to it; temperature and light (photo). They sample the readings of these sensors alternately and send them to the base station at each timer firing event. The data is sent after encryption. The base station sends a key update

request periodically (after every 20 timer firing events) and all the nodes are expected to update their keys and send the future data encrypted with the new key.

I varied the number of nodes in steps of 5 and ran the simulations for 3 minutes to check their performance. The results for 10 mote simulations are shown below. Also, some screen shots of TinyViz GUI for these simulations are shown.

Table 5.5: Power results

With base station sending key update messages	Base station not sending any messages
maxseen 9	maxseen 9
Mote 0, cpu total: 2215.966439	Mote 0, cpu total: 2215.966975
Mote 0, radio total: 3786.756697	Mote 0, radio total: 3784.647721
Mote 0, adc total: 0.000000	Mote 0, adc total: 0.000000
Mote 0, leds total: 1076.597958	Mote 0, leds total: 1079.766559
Mote 0, sensor total: 370.221996	Mote 0, sensor total: 370.222085
Mote 0, eeprom total: 0.000000	Mote 0, eeprom total: 0.000000
Mote 0, cpu_cycle total: 0.000000	Mote 0, cpu_cycle total: 0.000000
Mote 0, Total energy: 7449.543091	Mote 0, Total energy: 7450.603341
Mote 1, cpu total: 2212.550457	Mote 1, cpu total: 2212.550993
Mote 1, radio total: 3792.002861	Mote 1, radio total: 3791.861213
Mote 1, adc total: 0.000000	Mote 1, adc total: 0.000000
Mote 1, leds total: 1073.616636	Mote 1, leds total: 538.374193
Mote 1, sensor total: 369.651287	Mote 1, sensor total: 369.651377
Mote 1, eeprom total: 0.000000	Mote 1, eeprom total: 0.000000
Mote 1, cpu_cycle total: 0.000000	Mote 1, cpu_cycle total: 0.000000
Mote 1, Total energy: 7447.821241	Mote 1, Total energy: 6912.437777
Mote 2, cpu total: 2212.550457	Mote 2, cpu total: 2212.550993
Mote 2, radio total: 3641.495513	Mote 2, radio total: 3641.353865
Mote 2, adc total: 0.000000	Mote 2, adc total: 0.000000
Mote 2, leds total: 1050.834728	Mote 2, leds total: 515.592000
Mote 2, sensor total: 369.651287	Mote 2, sensor total: 369.651377
Mote 2, eeprom total: 0.000000	Mote 2, eeprom total: 0.000000
Mote 2, cpu_cycle total: 0.000000	Mote 2, cpu_cycle total: 0.000000
Mote 2, Total energy: 7274.531985	Mote 2, Total energy: 6739.148234
Mote 3, cpu total: 2212.550457	Mote 3, cpu total: 2212.550993
Mote 3, radio total: 3684.812769	Mote 3, radio total: 3684.671121
Mote 3, adc total: 0.000000	Mote 3, adc total: 0.000000
Mote 3, leds total: 1057.279628	Mote 3, leds total: 522.036900
Mote 3, sensor total: 369.651287	Mote 3, sensor total: 369.651377
Mote 3, eeprom total: 0.000000	Mote 3, eeprom total: 0.000000
Mote 3, cpu_cycle total: 0.000000	Mote 3, cpu_cycle total: 0.000000
Mote 3, Total energy: 7324.294141	Mote 3, Total energy: 6788.910391
Mote 4, cpu total: 2212.550457	Mote 4, cpu total: 2212.550993
Mote 4, radio total: 3778.133508	Mote 4, radio total: 3777.991860
Mote 4, adc total: 0.000000	Mote 4, adc total: 0.000000

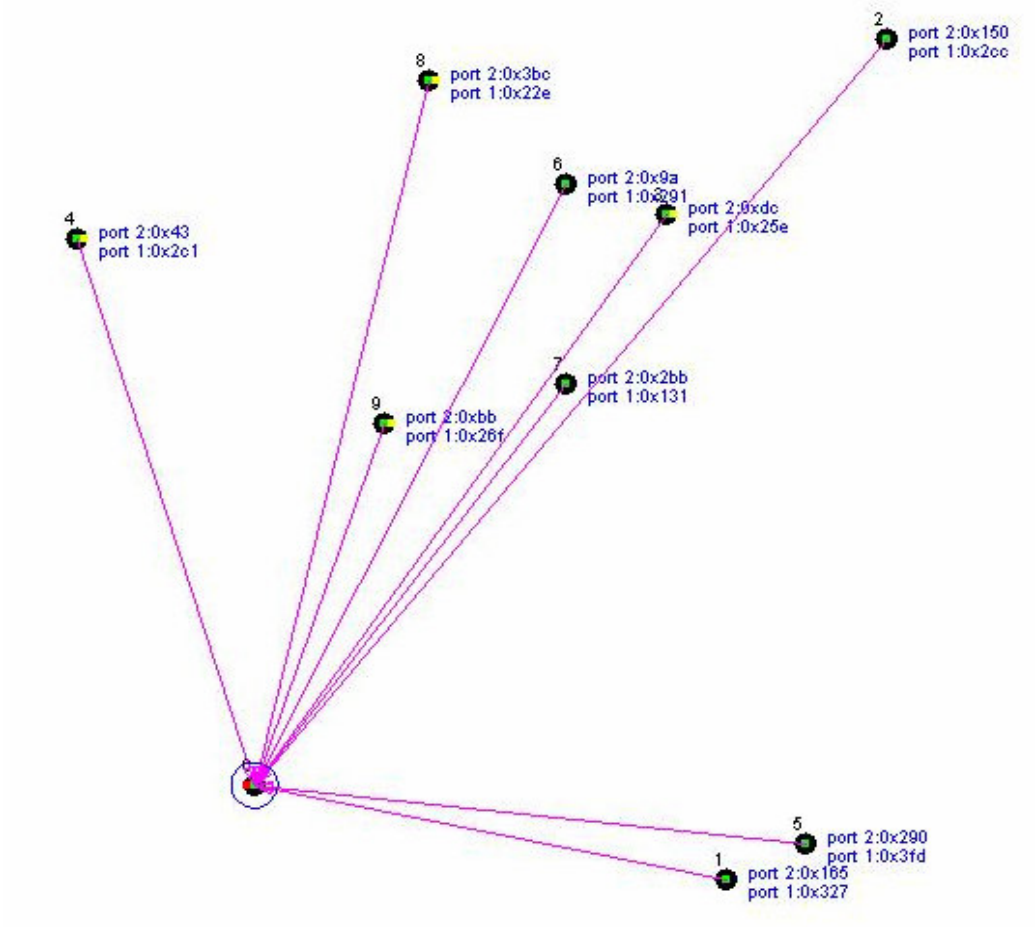
Mote 4, leds total: 1072.536061	Mote 4, leds total: 537.293618
Mote 4, sensor total: 369.651287	Mote 4, sensor total: 369.651377
Mote 4, eeprom total: 0.000000	Mote 4, eeprom total: 0.000000
Mote 4, cpu_cycle total: 0.000000	Mote 4, cpu_cycle total: 0.000000
Mote 4, Total energy: 7432.871313	Mote 4, Total energy: 6897.487848
Mote 5, cpu total: 2212.550457	Mote 5, cpu total: 2212.550993
Mote 5, radio total: 3640.965758	Mote 5, radio total: 3640.824110
Mote 5, adc total: 0.000000	Mote 5, adc total: 0.000000
Mote 5, leds total: 1050.834728	Mote 5, leds total: 515.592000
Mote 5, sensor total: 369.651287	Mote 5, sensor total: 369.651377
Mote 5, eeprom total: 0.000000	Mote 5, eeprom total: 0.000000
Mote 5, cpu_cycle total: 0.000000	Mote 5, cpu_cycle total: 0.000000
Mote 5, Total energy: 7274.002230	Mote 5, Total energy: 6738.618480
Mote 6, cpu total: 2212.550457	Mote 6, cpu total: 2212.550993
Mote 6, radio total: 3627.806326	Mote 6, radio total: 3627.664678
Mote 6, adc total: 0.000000	Mote 6, adc total: 0.000000
Mote 6, leds total: 1048.310333	Mote 6, leds total: 513.067891
Mote 6, sensor total: 369.651287	Mote 6, sensor total: 369.651377
Mote 6, eeprom total: 0.000000	Mote 6, eeprom total: 0.000000
Mote 6, cpu_cycle total: 0.000000	Mote 6, cpu_cycle total: 0.000000
Mote 6, Total energy: 7258.318404	Mote 6, Total energy: 6722.934939
Mote 7, cpu total: 2212.550457	Mote 7, cpu total: 2212.550993
Mote 7, radio total: 3634.504447	Mote 7, radio total: 3634.362799
Mote 7, adc total: 0.000000	Mote 7, adc total: 0.000000
Mote 7, leds total: 1050.406474	Mote 7, leds total: 515.164031
Mote 7, sensor total: 369.651287	Mote 7, sensor total: 369.651377
Mote 7, eeprom total: 0.000000	Mote 7, eeprom total: 0.000000
Mote 7, cpu_cycle total: 0.000000	Mote 7, cpu_cycle total: 0.000000
Mote 7, Total energy: 7267.112665	Mote 7, Total energy: 6731.729200
Mote 8, cpu total: 2212.550457	Mote 8, cpu total: 2212.550993
Mote 8, radio total: 3798.946116	Mote 8, radio total: 3798.804468
Mote 8, adc total: 0.000000	Mote 8, adc total: 0.000000
Mote 8, leds total: 1075.789489	Mote 8, leds total: 540.547047
Mote 8, sensor total: 369.651287	Mote 8, sensor total: 369.651377
Mote 8, eeprom total: 0.000000	Mote 8, eeprom total: 0.000000
Mote 8, cpu_cycle total: 0.000000	Mote 8, cpu_cycle total: 0.000000
Mote 8, Total energy: 7456.937350	Mote 8, Total energy: 6921.553885
Mote 9, cpu total: 2179.798490	Mote 9, cpu total: 2179.799026
Mote 9, radio total: 3742.898319	Mote 9, radio total: 3742.756671
Mote 9, adc total: 0.000000	Mote 9, adc total: 0.000000
Mote 9, leds total: 1066.946978	Mote 9, leds total: 531.704250
Mote 9, sensor total: 364.179409	Mote 9, sensor total: 364.179498
Mote 9, eeprom total: 0.000000	Mote 9, eeprom total: 0.000000
Mote 9, cpu_cycle total: 0.000000	Mote 9, cpu_cycle total: 0.000000
Mote 9, Total energy: 7353.823195	Mote 9, Total energy: 6818.439445

The results of table 5.5 show that there is some difference in the powers of normal motes, however that is due to LED power.

Table 5.6: CPU cycle results

With base station sending key update messages	Base station not sending any messages
Mote 0 CPU_CYCLES 147764.0 at 720000150 Mote 1 CPU_CYCLES 71912.5 at 720000150 Mote 2 CPU_CYCLES 68733.0 at 720000150 Mote 3 CPU_CYCLES 69583.0 at 720000150 Mote 4 CPU_CYCLES 71487.5 at 720000150 Mote 5 CPU_CYCLES 68733.0 at 720000150 Mote 6 CPU_CYCLES 68512.5 at 720000150 Mote 7 CPU_CYCLES 68512.5 at 720000150 Mote 8 CPU_CYCLES 71912.5 at 720000150 Mote 9 CPU_CYCLES 70858.0 at 720000150	Mote 0 CPU_CYCLES 145718.0 at 720000123 Mote 1 CPU_CYCLES 71764.0 at 720000123 Mote 2 CPU_CYCLES 68584.5 at 720000123 Mote 3 CPU_CYCLES 69434.5 at 720000123 Mote 4 CPU_CYCLES 71339.0 at 720000123 Mote 5 CPU_CYCLES 68584.5 at 720000123 Mote 6 CPU_CYCLES 68364.0 at 720000123 Mote 7 CPU_CYCLES 68364.0 at 720000123 Mote 8 CPU_CYCLES 71764.0 at 720000123 Mote 9 CPU_CYCLES 70709.5 at 720000123

There is a difference in the CPU cycle count for the base station. This is due to the inclusion of the code for sending the Key update messages in the application component.




```
[2] MOTE 2: Send Done PacketCount: 14, at Time: 0:0:35.57919525
[0] ****************** TIME for KeyUpdate 20 at 0:0:35.61007100 *******
[0] ----------------------------------------------------------
[0] The encryption key (Tx side) after modification is : 28 3c 39 82 94 71 5e cd
[1] The encryption key (Rx side) after modification is : 28 3c 39 82 94 71 5e cd
[4] The encryption key (Rx side) after modification is : 28 3c 39 82 94 71 5e cd
[6] The encryption key (Rx side) after modification is : 28 3c 39 82 94 71 5e cd
[8] The encryption key (Rx side) after modification is : 28 3c 39 82 94 71 5e cd
[3] The encryption key (Rx side) after modification is : 28 3c 39 82 94 71 5e cd
[7] The encryption key (Rx side) after modification is : 28 3c 39 82 94 71 5e cd
[9] The encryption key (Rx side) after modification is : 28 3c 39 82 94 71 5e cd
[2] The encryption key (Rx side) after modification is : 28 3c 39 82 94 71 5e cd
[5] The encryption key (Rx side) after modification is : 28 3c 39 82 94 71 5e cd
[0] Sent Message <BaseTOSMsg>   [addr=0xffff]  [type=0x4]  [group=0x7d]  [length=0xe8]  [data=0x0 0x0 0x0 0x0 0x0
[0] MOTE 0: Send Done PacketCount: 1, at Time: 0:0:35.63250000
[5] MOTE 5: sending Temp data - PacketCount: 8, Data: 65533 at Time: 0:0:35.65383050
[0] BASE: Received Reading Count: 113 , Data = 65533 at 0:0:35.67644700
[5] Sent Message <BaseTOSMsg>   [addr=0x0]  [type=0x4]  [group=0x7d]  [length=0xc8]  [data=0xfd 0x0 0x0 0x0 0x0 0x
[5] MOTE 5: Send Done PacketCount: 8, at Time: 0:0:35.67645475
[9] MOTE 9: sending Photo data - PacketCount: 23, Data: 65467 at Time: 0:0:35.73497700
[0] BASE: Received Reading Count: 114 , Data = 65467 at 0:0:35.75685175
[9] Sent Message <BaseTOSMsg>   [addr=0x0]  [type=0x4]  [group=0x7d]  [length=0xc8]  [data=0xbb 0x0 0x0 0x0 0x0 0x
[9] MOTE 9: Send Done PacketCount: 23, at Time: 0:0:35.75685950
```

Figure 5.2: Screenshot of TinyViz for 10 node simulation

5.3.4 Multi-Hop Network Simulation

For implementation in multi-hop environment, I used a simple network consisting of 3 nodes, in which node 0 acts as base station and broadcast the key update request. However, here node 1 is in the range of node 0 but node 2 is not. Due to this, only node 1 can hear the key update request. It first updates it key and then propagates the request to node 2. Node 2 then updates its key. The screenshot in figure 5.3 shows this simulation.


```
[0] ****************** TIME for KeyUpdate 10 at 0:0:20.588211175 *******
[0] ----------------------------------------------------------
[0] USR1: The length value in send.send is : 232
[0] The encryption key (Tx side) after modification is : e7 ee 1b 4d e6 d7 78 a0
[1] The encryption key (Rx side) after modification is : e7 ee 1b 4d e6 d7 78 a0
[1] Sending key update message from node 1
[1] USR1: The length value in send.send is : 232
[0] Sent Message <BaseTOSMsg>   [addr=0xffff]  [type=0x4]  [group=0x7d]  [length=0xe8]  [data=0x18 0x18 0x18 0x18
[2] The encryption key (Rx side) after modification is : e7 ee 1b 4d e6 d7 78 a0
[1] Sent Message <BaseTOSMsg>   [addr=0x2]  [type=0x4]  [group=0x7d]  [length=0xe8]  [data=0x18 0x18 0x18 0x18 0x1
[0] ****************** TIME for KeyUpdate 20 at 0:0:25.4707l175 *******
[0] ----------------------------------------------------------
[0] USR1: The length value in send.send is : 232
[0] The encryption key (Tx side) after modification is : cf dd 37 9a cc af f1 40
[1] The encryption key (Rx side) after modification is : cf dd 37 9a cc af f1 40
[1] Sending key update message from node 1
[1] USR1: The length value in send.send is : 232
[0] Sent Message <BaseTOSMsg>   [addr=0xffff]  [type=0x4]  [group=0x7d]  [length=0xe8]  [data=0x18 0x18 0x18 0x18
[2] The encryption key (Rx side) after modification is : cf dd 37 9a cc af f1 40
[1] Sent Message <BaseTOSMsg>   [addr=0x2]  [type=0x4]  [group=0x7d]  [length=0xe8]  [data=0x18 0x18 0x18 0x18 0x1
```

Figure 5.3: Multi-Hop Network Simulation Results

5.4 Comparisons with other key update schemes

The table 5.7 gives the comparison of my scheme with the schemes described in chapter 3. Since each scheme has different platforms/parameters for measurements, this is a broad comparison.

Table 5.7: Key Update Schemes - Comparison Summary

	ECC	**LEAP**	**SPAGKU**	**My Scheme**
RAM requirements	1 KB	3d + 2 + L	80 bytes	3 bytes (0.347%)
ROM requirements	34.1KB (26.5% of 128KB)	17.9KB	1.9KB	402 bytes (1.66%)
Time Required	1 minute 8 secs	-	-	38.75µsec
Complexity of key generation	High, ECC algorithm	Random Number generation	Random generation	Key rotation
Complexity of key distribution	N/A	Very high	Flooding the network with message	1 message with the third bit set
Computational costs	0.9 joules of energy, 1 minute 8 secs	2N (2 per node) encryption /decryption operations	One decryption operation per node	155 CPU cycles

5.5 Conclusion

It is evident from the results that the proposed scheme is extremely efficient in terms of resource consumption. Due to the limitations of TOSSIM, we could not get the exact computational overhead for this algorithm.

CHAPTER SIX

CONCLUSION AND FUTURE WORK

We proposed an extremely efficient algorithm for updating the encryption key in TinySec. The cost analysis and the results show that it is highly resource aware. There is no bandwidth or latency overhead since we are modifying the already existing key and not generating or sending the new key. However, there are some limitations to it e.g. there could be a maximum of 64 values for the key. So, there is a tradeoff between resource consumption and complexity which is the case for all the other schemes described in chapter three. The proposed algorithm is equally secure when compared to these schemes. Further, we have implemented it along with TinySec and the simulations were successful.

6.1 Implementation on Motes

Since our scheme is not that computationally intensive, TOSSIM and PowerTOSSIM were unable to model the results accurately therefore good future work would involve implementing the code on actual motes and verifying the results. Since the researchers of [12], [13] and [14] had access to the Berkeley mote environment, they implemented their code on Mica2 motes and presented their results.

6.2 Extension of the Scheme

As described in section 4.3, the extension to this scheme for making the key values less predictable would be to swap some intermediate neighboring bits after a certain number of key updates.

BIBLIOGRAPHY

[1] Ian f. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam and Erdal Cayirci, "A Survey on Sensor Networks", *IEEE Communications Magazine*, Aug. 2002

[2] Chris Karlof and David Wagner, "Secure Routing in Wireless Sensor Networks: Attacks and Countermeasures", *IEEE International Workshop on Sensor Network Protocols and Applications,* 2003

[3] Zhihua Hu and Baochun Li, "Fundamental Performance Limits of Wireless Sensor Networks", May 2005

[4] Seyit A Camtepe, Bulent Yener, "Key Distribution Mechanisms for Wireless Sensor Networks", *Technical Report TR-05-07, March 23, 2005*, Department of Computer Science, Rensselaer Polytechnic Institute.

[5] Chris Karlof, Naveen Sastry and David Wagner, "TinySec: A Link Layer Security Architecture for Wireless Sensor Networks**",** *SenSys'04,* November 3–5, 2004, Baltimore, Maryland, USA.

[6] Alessio Falchi, "The Berkeley Motes Environment", Chapter 3.
http://etd.adm.unipi.it/theses/available/etd-05252004-154652/unrestricted/Chap3.pdf

[7] http://www.tinyos.net/tinyos-1.x/doc/tutorial/lesson1.html

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K.Pister, "System architecture directions for networked sensors", *In Proc. of ASPLOS IX*, 2000.

[9] http://www.di.unipi.it/~scordino/sisop/tinyos_intro.pdf

[10] Philip Buonadonna, Jason Hill and David Culler, "Active Message Communication for Tiny Networked Sensors",

[11] Complete SkipJack and KEA Specifications,

http://csrc.nist.gov/CryptoToolkit/skipjack/skipjack.pdf

[12] David J. Malan, Matt Welsh, Michael D. Smith, "A Public-Key Infrastructure for Key Distribution in TinyOS Based on Elliptic Curve Cryptography", *IEEE International Conference on Sensor and Ad Hoc Communications and Networks (SECON04)*, 2004

[13] Moshe Golan, "Key Update in Sensor Networks", project report, http://lecs.cs.ucla.edu/~mosheg/Projects/KUReport.htm

[14] Sencun Zhu, Sanjeev Setia, Sushil Jajodia, "LEAP: Efficient Security Mechanism for Large-Scale Distributed Sensor Networks", *Tech-Report (ACM), Aug. 2004.*

[15] Ronald Watro, Derrick Kong, Sue-fen Cuti, Charles Gardiner, Charles Lynn1 and Peter Kruus, "TinyPK: Securing Sensor Networks with Public Key Technology**",** *SASN'04 (ACM)*, October 25, 2004, Washington, DC, USA.

[16] Ernest F. Brickell, Dorothy E. Denning,Stephen T. Kent, David P. Maher and Walter Tuchman, "SKIPJACK Review - Interim Report",July 28, 1993

[17] Philip Levis, Nelson Lee, Matt Welsh, and David Culler, "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications**",** *In Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys) 2003*, Nov. 2003.

[18] Philip Levis and Nelson Lee, "TOSSIM: A Simulator for TinyOS Networks", http://www.tinyos.net/tinyos-1.x/doc/nido.pdf

[19] Victor Shnayder, Mark Hempstead, Borrong Chen, Geoff Werner Allen, and Matt Welsh, "Simulating the Power Consumption of LargeScale Sensor Network Applications**",** *SenSys'04,* November 3–5, 2004, Baltimore, Maryland, USA**.**

[20] http://www.tinyos.net/tinyos-1.x/doc/tutorial/lesson5.html

[21] http://www.eecs.harvard.edu/~shnayder/ptossim/install.html