

Improvements to Data Transportation Security in Wireless Sensor Networks

Elias EKONOMOU

School of Computing, Science and Engineering
University of Salford, Salford, UK

Submitted in Partial Fulfilment of the Requirements of the
Degree of Doctor of Philosophy, June 2010

Table of contents

Table of contents.....	ii
Lists of figures and tables.....	vi
1. List of figures.....	vi
2. List of tables.....	vii
Acknowledgements.....	viii
List of abbreviations.....	ix
Abstract.....	x
1. Introduction.....	2
1.1 Overview.....	3
1.2 Motivation.....	4
1.3 Contribution.....	6
1.4 Intended audience.....	7
1.5 Thesis structure.....	8
2. Literature review.....	10
2.1 Introduction to Wireless Sensor Networks.....	11
2.1.1 Overview.....	11
2.1.2 Operational scenarios.....	14
2.1.3 Hardware platform.....	19
2.1.4 Software platform.....	23
2.1.5 Networking.....	24
2.2. Security in Wireless Sensor Networks.....	28
2.2.1 Overview of security.....	28
2.2.2 Threat model.....	31
2.2.3 Application of security.....	35
2.3 Related work.....	37
2.3.1 Key management schemes.....	37

2.3.2	Data transportation layer security schemes	41
2.3.3	Routing security.....	45
2.3.4	Other work	47
2.4	Discussion	50
2.4.1	Baseline requirements	50
2.4.2	Aims.....	52
2.4.3	Research Challenges.....	53
2.4.4	Summary	54
3.	System requirements.....	56
3.1	Basic security requirements.....	57
3.1.1	Confidentiality	57
3.1.2	Authentication and integrity	57
3.1.3	Freshness.....	58
3.2	Additional security requirements.....	60
3.2.1	Regarding routing security	60
3.2.2	Availability	61
3.2.3	Hardware.....	62
3.3	Operation and efficiency requirements	63
3.3.1	Key management mechanism	63
3.3.2	Efficiency requirements	64
3.4	Other requirements.....	65
3.4.1	Essential deployment requirements	65
3.4.2	Other requirements and desirables	67
4.	SecRose Specification and Design.....	69
4.1	Algorithmic description	70
4.1.1	Specification of concepts	70
4.1.2	SecRose operation: outgoing packets.....	75
4.1.3	Operation: incoming packets	77
4.1.4	Operation: authenticated acknowledgement transmission and reception.....	78
4.1.5	Operation: intermediate nodes	80
4.1.6	Operation: diagrams.....	81
4.2	System design	83

4.2.1	System overview	83
4.2.2	Encryption component.....	85
4.2.3	Key management component.....	93
4.2.4	Authentication and integrity	102
4.2.5	Control component	105
4.2.6	Diagrams of system operation	108
4.3	Summary	111
4.3.1	Simple design, simple operation	111
4.3.2	Innovative features	112
4.3.3	Chapter conclusion.....	113
5.	Implementation	115
5.1	TinyOS and SecRose	116
5.1.1	Description of TinyOS.....	116
5.1.2	Operation of TinyOS.....	117
5.1.3	Communication model and SecRose's position	118
5.2	Algorithms, code and pseudocode	121
5.2.1	Encryption component.....	122
5.2.2	Authentication component.....	124
6.	Evaluation.....	139
6.1	Evaluation of security provision	140
6.1.1	The threat model again	141
6.1.2	Provision of confidentiality	145
6.1.3	Provision of authentication and integrity	149
6.1.4	Provision of freshness	155
6.1.5	Additional security provision.....	157
6.1.6	Evaluation against other solutions.....	160
6.2	Performance evaluation.....	172
6.2.1	Explanation of methodology	172
6.2.2	Energy requirements.....	173
6.2.3	Latency	178
6.2.4	Memory.....	179
6.2.5	Comparisons.....	180
6.3	Evaluation of non-functional requirements	188

6.3.1	Essential deployment requirements	188
6.3.2	Other requirements and desirables	189
6.4	Summary	191
6.4.1	Critical security evaluation	191
6.4.2	Energy efficiency and non-functional requirements.....	193
6.4.3	Final comparison	195
7.	Conclusion and Future Work	197
7.1	Conclusion.....	198
7.2	Future work.....	199
7.2.1	Provision against current vulnerabilities.....	199
7.1.2	Alternatives on authenticated acknowledgements	200
7.2.3	Improvements on key management	202
7.2.5	Flexibility and customisation features	203
	APPENDICES	205
	IMPLEMENTATION CODE	206
	File CC1000RadioIntM.ns	207
	File SecRoseM.nc.....	224
	REFERENCES.....	235

Lists of figures and tables

1. List of figures

Figure 1: an advanced sensor node.....	21
Figure 2: SecRose packets, their fields and their security features	71
Figure 3: mixing of the <i>initial key</i> with the <i>counter</i>	73
Figure 4: pair key advancement and state preservation sequence	82
Figure 5: position of SecRose in TinyOS	84
Figure 6: the subcomponents of the encryption component and their interfaces	88
Figure 7: the principle of ciphertext stealing	89
Figure 8: the stealing subcomponent of SecRose	91
Figure 9: example of stealing applied to a <i>long</i> packet with two bytes data payload	92
Figure 10: key management: interfaces, interactions and subcomponents	93
Figure 11: key request process	97
Figure 12: key management tasks after packet transmission	98
Figure 13: process after acknowledgement reception	99
Figure 14: process after packet reception	100
Figure 15: revert to <i>backup counter</i>	101
Figure 16: MAC generation process.....	104
Figure 17: MAC validation process	104
Figure 18: transmission of packets and reception of acknowledgements by the sender.	109
Figure 19: reception of packets, validation and transmission of acknowledgements.	110
Figure 20: sequence diagram of packet injection attacks.....	151
Figure 21: key changes and prevention of packet replays.....	155
Figure 22: CPU energy requirements for a SecRose transmitter	175
Figure 23: CPU energy requirements for a SecRose receiver.	176
Figure 24: radio energy requirements for a SecRose transmitter.	177
Figure 25: CPU energy requirements for both nodes.....	182
Figure 26: radio energy requirements for both nodes.....	183
Figure 27: energy requirements for both CPU and radio.	184
Figure 28: normalised energy requirements for both CPU and radio.....	185

Figure 29: accumulative radio energy consumption for both nodes	186
Figure 30: accumulative CPU energy consumption for both nodes	186
Figure 31: accumulative energy consumption for both nodes.....	186
Figure 32: normalised accumulative energy consumption for both nodes.....	186

2. List of tables

Table 1: Routing attacks, their requirements and how they can be prevented.	46
Table 2: packet field utilisation by the stealing subcomponent.....	92
Table 3: comparative summary of the provided security for each mechanism	168
Table 4: comparison of the security provision of SecRose and 802.15.14	170
Table 5: executable size of various proposals	179

Acknowledgements

I would like to thank my supervisor Dr. Kate Booth for providing inspiration, challenges, guidance and for showing unlimited patience and trust on myself. In addition, I would like to thank my family for their love and unconditional support at all levels. My work is devoted to them. Finally, I would like to thank my friends for the motivational input and the important moments of joy.

List of abbreviations

ACK	Acknowledgement
AES	Advanced Encryption Standard
AM	Active Messages
API	Advanced Programmer Interface
BP	British Petroleum
CBC	Cipher-Block Chaining
CMAC	Cipher-block chaining Message Authentication Code
CNSS	Committee on National Security Systems, The
CNT	counter
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
EEPROM	Electrically Erasable Programmable Read Only Memory
GNU	GNU is Not Unix (recursive)
GPL	General Public Licence
GPS	Global Positioning System
ID	Identification number
IEEE	Institute of Electrical and Electronic Engineers
IP	Internet Protocol
IV	Initialisation Vector
LIFO	Last In, First Out
MAC	Message Authentication Code (not Medium Access Control)
MSB	Most Significant Bit
NIST	National Institute of Standards and Technology
OCB	Offset Codebook Mode
OS	Operating System
OSI	Open System Interconnection
PC	Personal Computer
PCB	Printed Circuit Board
PKI	Public Key Infrastructure
PKT	packet
RX	receive
SSL	Secure Sockets Layer
TX	transmit
US	United States
USB	Universal Serial Bus
USD	United States Dollars
WSN	Wireless Sensor Network

Abstract

Wireless Sensor Networks are computer networks consisting of miniaturised electronic devices that aim to gather and report information about their environment. The devices are limited in computational ability, temporary and permanent memory and communication ability. Furthermore, the devices communicate via a wireless unregulated medium and usually operate on finite power sources such as batteries.

Security in Wireless Sensor Networks is the research area that seeks to provide adequate level of security for the limited sensor devices, aiming to increase the possible applications of Wireless Sensor Networks and allow them to be deployed for a wider variety of tasks, including monitoring of critical conditions or valuable infrastructure. The area has to solve the problems associated with the limited nature of the devices. Traditional security mechanisms are inappropriate for Wireless Sensor Networks, because they were not designed for resource-constrained environments.

This research attempts to solve the problems associated with secure message exchange via an open medium without introduction of significant resource overheads. The result of this research is SecRose, a security mechanism for the data-transportation layer of Wireless Sensor Networks.

SecRose provides a higher level of security than the existing proposals while it demonstrates better performance characteristics. In particular, the mechanism introduces authenticated acknowledgements and key management, improves the provided cryptographic strength and helps in securing the routing protocols. On the other hand, the mechanism operates without significant computational or communication overhead and is backward compatible with existing sensor network applications.

The thesis discusses the requirements, design and evaluation of the mechanism and demonstrates how its goals are achieved by following alternative approaches to provide the security properties.

Chapter 1

Introduction

1. Introduction

1.1 Overview

Wireless Sensor Networks constitute a promising technological advance that poses contradicting research challenges. The networks are limited, miniaturised computers that aim to sense and wirelessly report information about their environment. Typical networks are expected to be self-organising and able to operate with little or no human interaction.

Provision of security widens the possible application of sensor networks and is therefore a desirable marketing feature. On the other hand, security would require utilisation of energy resources, a precious commodity for the longevity of the battery powered wireless sensor networks.

This contradiction has posed a significant research challenge aiming to provide adequate security in a minimalistic resource-constrained environment. An area of research was created to explore the possibilities of providing Security in Wireless Sensor Networks [1]. A particular field of this research area is interested in the security of the exchanged messages. This field is the specific area of our research and is known as security for the data-transportation layer.

This document describes SecRose, a mechanism that provides security for the data transportation layer. Briefly, our thesis is that SecRose provides a level of security that is both better and consumes less energy than similar mechanisms.

1.2 Motivation

The security in wireless sensor networks paradox

There is a significant amount of research literature regarding data-transportation layer security for wireless sensor networks. The literature includes contributions that range from simple frequency hopping systems to complex public-key cryptosystem attempts. The literature covers scenarios that range from human heartbeat monitoring to military battlefields. Seemingly, many of these solutions were developed under unrealistic assumptions. Consequently, only a few of those solutions have made it to actual commercial products.

In our belief, three factors caused this paradox. Firstly, initial research set unrealistic targets. Secondly, improvements in microprocessors allowed conventional security mechanisms to be applied in many WSN applications. Finally, but most importantly, the assumptions of what is possible and profitable were narrowed down as sensor networks progressed.

As a consequence, there is only one proposal considered to be the de-facto standard in our area of research and that is the TinySec security architecture [2].

Inadequate security systems

TinySec is a relatively old mechanism that was publicly released six years ago and has not been updated since. TinySec is well designed, documented and most importantly fully implemented. On the other hand, it does not provide any cover for some of the available attacks or it provides limited level of security for others.

Other systems that were developed after TinySec attempt to improve it but they use untested or poorly evaluated innovations while there is no evidence on whether they were actually implemented on code or not.

Our thesis

SecRose is a data-transportation layer security mechanism based on an alternative than the existing direction. The resulting mechanism provides adequate security in conjunction with fewer resource requirements.

In brief, SecRose provides:

- Key management to facilitate frequent changing of used keys in a manner that is energy efficient and does not leak information to adversaries
- Authenticated acknowledgements which can secure third party routing protocols and validate communication integrity
- Up-to-date cryptographic strength using an established cryptographic cipher that natively provides appropriate key length
- Alternative solutions for semantic security and freshness that are subject to different limitations and conditions than other proposals
- Equal level of authentication as any other mechanism which provides acceptable level of confidence

In addition, SecRose manages all these features without requiring significant resources. Importantly, the mechanism is more efficient than TinySec, which acts as the primary basis of comparison.

SecRose is based on generic assumptions that are similar to other mechanisms. Most importantly, SecRose assumes that the node's hardware is secure and it cannot be tampered with.

Finally, SecRose includes useful non-functional requirements like backwards compatibility, ease of deployment, scalability and others. Adaptation of SecRose requires minimal effort and provides optimal results.

1.3 Contribution

This thesis contributes SecRose, a data-transportation layer security mechanism, which is capable of providing a higher than the currently available level of security without introducing significant resource overhead. SecRose achieves its aims by introducing innovative features or alternative design or by improving on existing functionality.

1.4 Intended audience

This document is intended to computer scientists. In particular, the document assumes that the reader is familiar with the following topics.

Principles of security

The document does not cover the generic principles of computer security, cryptography or the required mathematical background. Although complex, formal, explanations are present in very few sections, an understanding of network security and cryptographic principles is assumed.

Background reading includes references [3-6]

Computer and telecommunication networks

The mechanism operates on a computer network. Knowledge of computer communication techniques is assumed. In particular, knowledge of the OSI model and the protocols that operate at its lower layers is essential to follow the concepts described here. Some knowledge of telecommunications and the related electronic engineering is also assumed.

Background reading includes references [7-10]

Desirables

In addition to the above background knowledge, knowledge about programming and processor operation would be beneficial. If the reader wishes to understand the proof-of-concept and its optimisations code then extensive knowledge of event-driven C programming and processor operation is required.

1.5 Thesis structure

This document is organised in seven chapters each addressing a different problem in relation to our research.

Chapter 1 introduces the research area, provides background information and gives statements of our thesis and motivation.

Chapter 2 is a literature review of the area, which includes an overview of wireless sensor networks, the application security on them and description of similar work. The chapter concludes by describing our desired research direction and baseline requirements.

Chapter 3 provides the system requirements for our proposal and gives rationale for each requirement.

Chapter 4 gives a detailed and clean design and operation the SecRose mechanism, its components and interfaces.

Chapter 5 provides a description of the proof-of-concept implementation.

Chapter 6 evaluates SecRose against the requirements, states the level of security provision and critically compares its security and performance with other proposals.

Chapter 7 provides further work and conclusion.

Chapter 2

Literature review

2. Literature review

This chapter presents and evaluates the existing work in the area. A generic introduction to the role of Wireless Sensor Networks is given first, along with software and hardware. Then the chapter discusses the security of sensor networks and existing work that is similar with this thesis. The chapter concludes with a discussion of the baseline requirements and the challenges of this project.

2.1 Introduction to Wireless Sensor Networks

2.1.1 Overview

Terminology definition

The definition of *Wireless Sensor Networks* can be traced in research work appearing on or soon after 2000. The term is abbreviated as WSNs while they are also referred to as *sensor networks*. The terms are used to describe a network of sensor nodes connected together wirelessly. The definition of a sensor node was also provided in the same period but the exact first appearance of both terms is difficult to trace.

The most popular definition of a *sensor node* came by a survey work from Akyildiz *et al.* [1] when they said “*recent advances in wireless communications and electronics have enabled the development of low-cost, low-power, multifunctional sensor nodes that are small in size and communicate untethered in short distances.*” They have also defined a sensor node to consist of “*sensing, data processing, and communicating components.*” The work of Akyildiz *et al.*, cited by a considerable number of authors, is probably the most reputable survey paper of the area. Additionally, it was published after the field had matured and so their definitions are considered accurate.

Purpose

Sensor networks are a technologically improved version of the simple electronic sensor devices used in automation applications. Their main purpose is to monitor one or more environmental parameters and then report to one or more central locations, called base station(s) [1].

Their computational and communication capabilities widely broaden the application range of the simple sensors that they replace. They were initially thought to be useful in military, health and commercial applications [1]. Since then, they have been commercially advertised as capable of working in industrial applications, building automation and asset management [11]. An important application field that has recently become very popular is in assisting scientific work that requires monitoring the natural environment; various scenarios include

bio system monitoring [12], wildfire protection [13], global warming [14] and agriculture [15].

Characteristics

There is great debate on the detailed abilities and characteristics of sensor networks. Although most research recognises a set of properties what sensor networks should possess, they disagree on what the reality allows them to accomplish. The difference of theoretical and practical limitations fuels the academic debate and recent research tends to deny previous statements. For example Akyildiz *et al.* state that “*a sensor network is composed of a large number of sensor nodes that are densely deployed*” [1] while Gamage *et al.* claim that a such network is “*mythical*” [16].

Although authors fail to agree on defining limits, there is consensus on a core set of generic characteristics. The characteristics listed by [1] are in agreement with both early publications, as in [17-19] and with later work, for example [20-22]. All the above authors characterize sensor networks as self-organising, able for in-network processing, fault tolerant and scalable. However, despite the agreement on the concept of scalability, the actual size of a typical network is a highly disputed property. Gamage *et al.* in [16] claim that a practical network size might be one hundred nodes while [23] use examples with network sizes of 10,000 nodes.

Limitations

Every published work agrees that the sensor devices are limited in computational, storing and communication capabilities when compared with traditional computer networks. However, the literature does not express a homogenous definition on the exact limitations of sensor devices.

The ultimate limitation of sensor networks is the fact that they operate on a finite power source. This argument is extensively supported by the literature as there are publications attempting to address the problem, for example [19, 24-28] and others that consider it a limitation, as in [1-2, 16, 21, 29-30]. Since different power sources provide different energy capacity [11], there is variation on offered abilities.

The typical sensor device that dominates the literature is the MICA2 node [11]. It offers an 8MHz CPU and communication capabilities at the rate of 38.4KBps. This device can run TinyOS [31] and benefits from the TinySec [2] security mechanism. Much more capable [11] and less capable devices [32] also exist. Thus, differences in capabilities are usually associated with differences in the devices physical size as well.

Security considerations

The modern world is full of security risks and every networked computer system is a potential target [6]. Wireless Sensor Networks are no different; security must be provided by in order to make them suitable for a wider range of applications [33].

Many of the potential application scenarios for wireless sensor networks are either too critical [34] or too valuable [35] to be run without an acceptable level of security. Therefore, security can help Wireless Sensor Networks to reach reaching their full potential by making them more attractive, a prospect that would result in fewer costs and easier use of WSNs everywhere.

However, the limited nature of the devices is an important obstacle in providing adequate security [28], sufficient to enable application in most environments [2]. Traditional security mechanisms cannot operate in sensor networks as they are too demanding in resources [29]. A new security solution has to be developed to provide an acceptable level of security with as low as possible impact to the longevity of the sensor network.

2.1.2 Operational scenarios

Wireless sensor networks have been proposed for a variety of different applications [1]. This subsection presents popular application fields and gives a brief description of representing operational scenarios. The application fields that demonstrate similar characteristics can be grouped into four categories: military applications, automation systems, health monitoring and environmental monitoring. Finally, the importance of security in each of the different application scenarios is discussed.

Military applications

Wireless sensor networks are considered suitable for military applications, since their abilities make them attractive for C4ISR¹ systems [1]. Their ability to be deployed rapidly, self-organise and then operate while tolerating possible faults was valued as a cheap alternative to previously used military sensors.

Typical military applications assume that sensor networks are very large and aim to equip them with methods to resist intelligent attacks by the enemy. Their ultimate target is to allow the network to remain functional even when it is attacked. Military interests helped promote research in WSNs and in the area of security in particular. Notably, it has been suggested that the assumptions and requirements of military applications are unrealistic and that the limited nature of WSNs cannot support them [16].

The most difficult requirement is to maintain the security of the network using a system that does not trust any component. This idea posed an intriguing research question that produced important contributions to knowledge, primarily the probabilistic key management schemes for wireless sensor networks. Examples include the work presented in [23, 27, 36-39].

A prime example is the work of Eschenauer and Gligor [37] where the authors specifically designate a solution suitable for military sensor networks. They assume a wireless sensor network which may be deployed in hostile environments, consists of tens of thousands of nodes, its size is dynamically scalable, might be subject to eavesdropping attacks and most importantly; its limited nodes are susceptible to capture and intelligent manipulation by

¹ C4ISR refers to the concept of Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance.

adversaries. The authors provide a solution that limits the impact of attacks to a small subset of the network.

The security requirements of such applications are rather obvious; with lives and national interests at stake, the military cannot rely on a system that the enemy can disable. Current surveillance systems like satellites and drones are expensive and the WSNs promised a radical decrease in costs. If WSNs were proved equally secure and reliable, they could replace or complement existing surveillance methods.

Automation systems

This category includes applications that utilise a wireless sensor network to monitor the condition of a particular structure or machinery. Possible application fields include industrial applications and building automation. There are two examples to illustrate their characteristics.

The first example is [40], where BP uses a sensor network to monitor the vibrations of rotating machinery onboard a tanker ship. Vibrations are known to be representative of the condition of any rotating machinery. Therefore, BP can remotely monitor their equipment and plan servicing only when required. Consequently, both servicing costs and valuable port time is saved. The wireless nature of WSNs allows BP to retrofit this system on any of its tankers without special planning.

The second example, described by Huang in [41], is a sensor network that monitors temperature, humidity and airflow in the rooms of a building. Because of the accurate monitoring, the system is able to fine-tune the air conditioning parameters to save energy and produce a more pleasant environment.

These applications share a common set of characteristics. These sensor networks are:

- operated within a controlled environment, in a man-made structure
- composed of a relatively small number of nodes

These similarities are usually present in all automation applications. WSNs provide a modern solution to the problem of automation, which was tackled in the past by utilising wired sensors, e.g. thermostats, which reported raw data to a central system. The central location then analysed the data, made decisions and possibly engaged into corrective action. In the

sensor network paradigm, the nodes analyse the gathered data and the central location now focused in utilising the information rather than obtaining it. The wireless nature of WSNs is cheaper and allows for much faster and easier deployment, especially in retrospective installations.

BP expects sensor network applications to make a high impact in the Industrial sector in the coming years [35] while Rabaey *et al.* claim \$55 billion USD annual savings should air conditioning monitoring be improved by the use of wireless sensor networks in the US [42]. Those statements, accompanied with the low cost and ease of deployment, suggest that the generic market of commercial applications for wireless sensor networks will thrive in the future.

There are pragmatic reasons why security must be provided to automation applications, regardless of their importance. First and most importantly, any system might be attacked by anyone for no obvious reason [6]. Secondly, there are increasing concerns that an attack aiming to facilitate espionage or sabotage might happen in industrial automation systems and if successful it might incur costs and disruption[43]. Moreover, sensor networks are often parts of bigger systems. If they are not secured, they are a weak link risk to other systems.

Finally, security-aware organisations are unlikely to invest into systems that are not properly secured, regardless of the system's role. Security might act as catalytic influence when deciding to deploy a WSN. Consequently, security makes WSNs more marketable and gives them a higher chance of adaptation by companies.

Healthcare

There is a range of sensor network applications directly applicable to the health and fitness sector. Usually such applications operate in the Personal Area Network (PAN) domain, as they are wearable devices utilised to monitor the biological condition of a patient or an athlete. Although there are many genuine health applications, most of them are commercial products and are not associated with published academic work. These products are outlined in [34].

References [34, 44] describe healthcare WSNs with the following characteristics:

- they operate in a relatively safe indoors environment
- the network consists of a small number of nodes, usually less than five
- they have to be small enough to be comfortable to wear
- they are not limited by power as their user is able to replace the batteries

An example health application is the one described by Milenkovic *et al.* [44]. Their system is a wearable sensor network that has to be secure, reliable and interoperable. Its objective is to allow patients to live in their home while allowing the hospital to monitor them constantly. The system consists of few very small sensor nodes, a personal server and a base station. The patient wears the small sensors and the personal server while the base station can be located anywhere in the house as long as it is connected to the hospital via the internet. The small sensors report on the higher-capability personal server, which in turn transmits the information to the base station. From that point, information can be transmitted to the hospital. This solution allows the doctors to monitor the patient while they enjoy the comfort of their home.

Any health record is considered private in many jurisdictions and it is protected by appropriate legislation. For example, health condition is considered as “sensitive personal data” by the U.K. Data Protection Act 1998 [45]. Healthcare sensor networks might generate information classified as health records and thus they must conform to these legal requirements. On the other hand, fitness systems might not require security but high-end fitness systems might have security enabled as they can be used for health monitoring as well.

Environmental monitoring

This category includes various applications that aim to monitor some aspect of the natural world. Examples are applications intended to monitor bio systems [12], wildfire [13], the effects of global warming [14] and agriculture [15].

The first example, known as the Great Duck Island project, is intended for natural habitat monitoring in general and more specifically to monitor “seabird nesting environment and behaviour” [12]. The nodes on this system are deployed in seabird nests to monitor incubation and other breeding patterns. They also aim to correlate the bird’s behaviour with other environmental parameters, such as temperature.

In the second example, the authors describe a wireless sensor network deployed in a forest [13]. The network monitors temperature and humidity and reports to other components of the system. This information can then be used to determine the risk of fire breaking out while temperature itself is a clear indicator of an actual fire ignited in the forest. The authors describe a system capable of accurately reporting the location of a fire.

The third example is part of a scientific project that “focuses on sub-glacial bed deformation” [14]. This system uses sensors deployed under the moving glacier at a depth of 50 to 80 meters. Once the custom sensors were deployed, they were non-recoverable and their longevity would rely on their battery. The sensors monitored temperature, pressure and tilt information.

The final scenario is a representative of systems that might be used in farms to assist in regular activities like farming and cattle grazing. The systems might consist of either static sensors that monitor ground moisture or mobile sensors attached to livestock. The authors describe systems for both objectives and there are companies that selling such systems commercially [11].

These applications are quite different and each has its own problems and requirements. However, they share a set of common characteristics:

- the network covers a large geographical area, nodes might be separated by significant distances
- the nodes are deployed in the physical environment, exposed to harsh conditions
- the network is expected to function for a large period of time
- the individual nodes might remain in network isolation for months
- the nodes might employ data aggregation and batch reporting techniques
- the network needs to be reliable and robust
- the network might sleep for long periods of time before it activates and reacts to events
- the network needs to be remotely managed
- the nodes might be fully mobile

These common features are usually dictated by the nature of the environmental monitoring activity. In the glacial monitoring application for example, the difficulty of deployment and

the non-recoverability of the sensor nodes required them to remain in operation for as long as possible.

It is evident that wireless sensor networks might be used in a range of environmental monitoring applications for scientific, welfare or Health & Safety reasons. Since the current environmental concerns require even more remote monitoring capability, sensor networks will probably become a tool to scientists of other disciplines.

Many of these applications require security for reasons similar to the automation systems discussed before. For example, security is a perfectly justifiable requirement for wildfire monitoring WSNs. However, others applications might not seem to require security, for example seabird behaviour monitoring. Security is a desirable requirement even in these cases since any insecure system might be attacked even if there is no apparent reason or gain [6]. Provision of security in all applications would increase the confidence in the reliability of WSN-generated information.

2.1.3 Hardware platform

This subsection discusses the hardware in use for wireless sensor networks. The computational and communication abilities of hardware, the physical structure of sensor node hardware and the way that they are organised are discussed. This subsection concludes by analysing the implication of the hardware's abilities and limitations to the security of sensor networks.

Capabilities

One key advantage of wireless sensor networks against conventional sensors is their ability to process data and wirelessly communicate information only when that is required. In order to facilitate their advantage, sensor devices are equipped with a modern but lightweight central processing unit (CPU), and a radio communication component [1].

The CPU of a sensor node acts a central decision maker, controller and coordinator for all components of the device [1]. As discussed in 2.1.4, the CPU might be limited in computational power but it is rather complete in features and capable of running miniature

operating system. A variety of microcontrollers has been used in sensor devices. Their computational capabilities range from 16-bit 8MHz[46] to 32-bit 419MHz processors [47].

In addition, the hardware provides temporary and permanent storage. Typical nodes like the MICA2 node offer 4KB configuration EEPROM, 128KB program flash memory, 512KB serial flash memory [48].

Essentially, the sensor device is a very small computer with processor, memory and radio acting as input/output interface. The devices have to be minimalistic because they depend on a finite power source but current research is implementing features found on larger devices. Examples of such features are; security [2], fault-tolerance [49], scalability [50], multi-agent systems [51] and others, as described in [1]. In addition, the CPU allows for energy savings as demonstrated in [31, 52]. The radio component on the other hand acts as the input/output interface and enables devices to form a network.

Components and organisation

Sensor hardware follows a modular design where individual components form tailored sensor nodes suitable for all possible usages. There are four types of modules, as defined by [1];

- main boards; provide computation, communication and storage
- sensing boards; host analogue or digital electronic sensors
- connectivity boards; provide other than radio connectivity like USB
- power sources; provide energy

The typical sensor node consists of a sensor board, a sensing board and a power source [53]. The exact components are determined by the role of the node in the network. Some nodes might lack a sensing board or benefit from connectivity boards or auxiliary power sources. The components are connected in the form of a stack and enclosed in a case occupying about 100cm³ of volume. The sensor devices might utilise any existing sensing technology; from the simplest microphone, to complex cameras or anything else technology can sense [1, 11]. For an example sensor, see Figure 1.

Each main board consists of four basic components that are connected together using one or more PCBs [1]. The microprocessor and storage components are usually integrated in one chip while the radio component might be separate. Figure 2 illustrates how components on different modules interface with each other. The sensing unit marks components hosted in the

sensing board, the processing unit and the transceiver are hosted in the main board. Primary and auxiliary power sources are also illustrated. Future sensor nodes might integrate all the components in a single chip, as shown by Khan *et al.* in [32].



Figure 1: an advanced sensor node.

The picture illustrates the sensing board, which includes a camera, the sensor board in the middle and a connectivity board on the bottom. The scale of the picture is close to real life size.

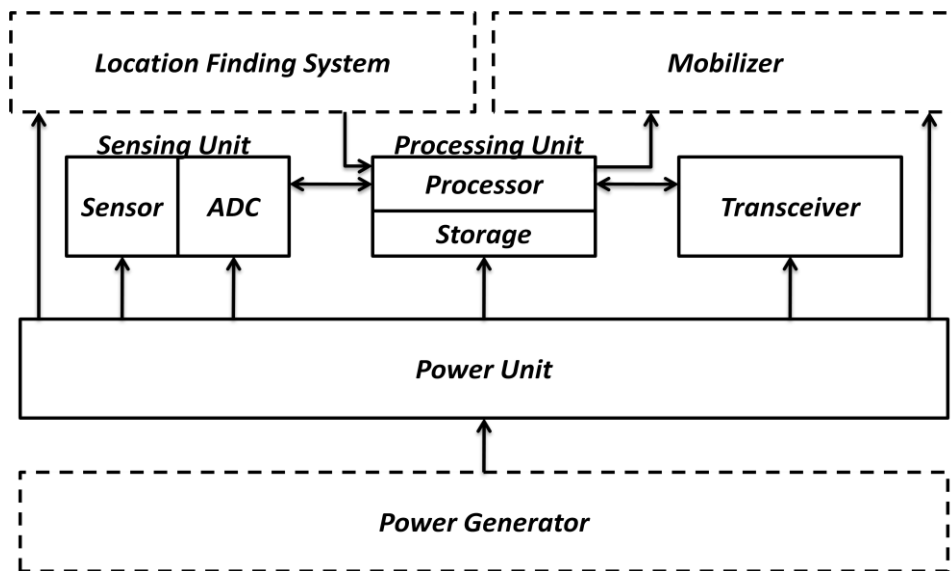


Figure 2: component organisation. Figure is based on similar figure from [1]

Security implications

As shown in the hardware explanation and examples, the devices are limited in their abilities but they are complete computers but they cannot handle large amounts of data, they are very slow in their processing and extensive utilisation of resources reduces their lifetime.

Therefore, any design of a security protocol must consider three important points in their proposal; the amount of permanent or temporary memory required to operate, the time it takes for the security computations to conclude and the energy overhead introduced.

Traditional security mechanisms often require ability to apply complex processing on large amounts of data. As explained in [2, 28-29], sensor devices lack storage capacity. In fact, they cannot even hold variables of sufficient size to accommodate a traditional security mechanism[29] like the Public Key Cryptography described by [54]. The security components that can be utilised for WSN use have to be selected while taking memory requirements as a criterion.

Another problem with traditional security is the processing overhead that it introduces. Although a sensor network's processor is capable of executing any calculation, it cannot do so fast enough. Many sensor network applications would require rapid response [13] from the sensor network and more significantly, some network communication methods are based on timeouts, they cannot operate if normal latency exceeds a certain threshold.

A sensor node might eventually execute a computation and might manage to send the information over its network but continuous execution of difficult computational tasks would deplete its resources and result to a disabled network. A MICA2 node running on full power would only last for 2 weeks before its batteries dry out [2]. Therefore, effort must be made so that security designs do not deplete the battery source as similar efforts were done for other systems of a WSN, for example [1, 19, 24, 31, 55], accounts for energy consumption in one way or another.

2.1.4 Software platform

Sensor networks are miniature computers that can run software. There is usually an operating system and an application which runs on it [1]. The software is often developed using PC simulators. This subsection describes the operating systems, the simulators and the security challenges associated with operating systems.

Operating systems

The Operating System (OS) provides the underlying mechanisms that allow the applications to interact with hardware. The purpose of the OS as defined by [31] is;

- manage the limited resources of the device
- perform the requirements of the sensor network in real-time
- balance modularity, flexibility and optimisation
- provide a low power platform to run applications

In order to achieve its purpose, the OS provides reusable APIs to the applications. The tasks are executed via the APIs as desired by the application, allowing it to become the main component that controls the device's behaviour. This is a typical design found in operating systems suitable for low end devices, like the TinyOS [31]. High end devices might run different operating systems that allow for multitasking and are therefore based on different designs [56].

Although a number of operating system designs have been proposed, for example [56-58], TinyOS enjoys the highest popularity by researchers, organisations and commercial distributors, references [1, 12-13, 15, 28, 31, 40, 44, 59] directly or indirectly endorse TinyOS. It is a flexible, application-specific operating system for sensor networks. It is a component-based, event-driven operating system written in nesC, a dialect of C. TinyOS provides a lightweight networking architecture featuring Integrated-Layer Processing, power management, hardware/software transparency, precise time synchronisation and routing.

Simulators

Due to the nature of wireless sensor networks, development of complex applications might be a difficult process. Testing small changes in the applications might require many hours of preparation and expensive hardware [60-61]. WSN simulators have been developed to tackle these problems. A simulator attempts to represent the behaviour of the sensor hardware on a

personal computer in a manner that would assist research on sensor networks and development of applications.

As discussed in our survey paper on WSN simulators [61], there are many simulators available to potential developers. The present research has concluded that the TOSSIM [62] simulator is suitable for development stage and the Avrora [63] simulator should be used for evaluation purposes.

Security implications

The existence of a miniature OS is advantageous and does not pose any problems far as security is concern. In fact, the existence of the OS helps in providing efficient security as much as it helps with other WSN applications.

2.1.5 Networking

This subsection discusses the networking capabilities of hardware and how they are utilised by the TinyOS operating system to achieve wireless communication. The topology of sensor networks is also described. Finally, the security implications of the networking model of WSNs are analysed.

Network capabilities

As explained before, the sensor devices are miniature computers and the radio component acts as an input/output interface. The combination of radio with CPU and storage creates a network. Sensor networks benefit from features such as medium sharing [2, 64], self-organised routing [65] and packet division [64]. These features are similar to what is found in all computer networks, only greatly optimised and miniaturised for energy efficiency and better suitability on a constrained environment.

A variety of radio capabilities is observed. The exact details of radio operation and medium utilisation are defined by the operating system² but the OS may or may not follow an international communication standard.

² Discussed in 2.1.3.

Low-end devices use customised communication methods designed for low power consumption. For example, the popular MICA2 node [48] provides a radio component able to communicate at 38.4KBps but relies on the operating system to define the method of medium access.

High-end devices with increased abilities have adapted existing communication standards to improve compatibility, such as Bluetooth [66] and IEEE 802.15.4 [67]. For example, the Imote2 [47] node is capable of 250KBps data rate and compatible with IEEE 802.15.4.

Typical network topology

Wireless sensor networks follow loosely defined rules on network topology and deployment. As explained by [1] and illustrated in Figure 3, the networks are “*usually scattered*” in a “*sensor field*.” Alternative topologies have also been identified, for example [19, 44, 68]. Nevertheless, each node performs a predefined role; it can be a sensor node, a cluster head or a base station.

The network design and the application’s specifics dictate the exact role. Sensor nodes are devices that act as data generators and they need to communicate this information with the base station. Since the base station is often out of range, the sensor nodes can utilise neighbouring sensor nodes or cluster heads [19] to propagate their information [1].

An important role in a wireless sensor network is the base station, or sink. They act as a gateway between the wireless sensor network and other devices or computers. In order to facilitate this functionality, they are equipped with appropriate hardware and software.

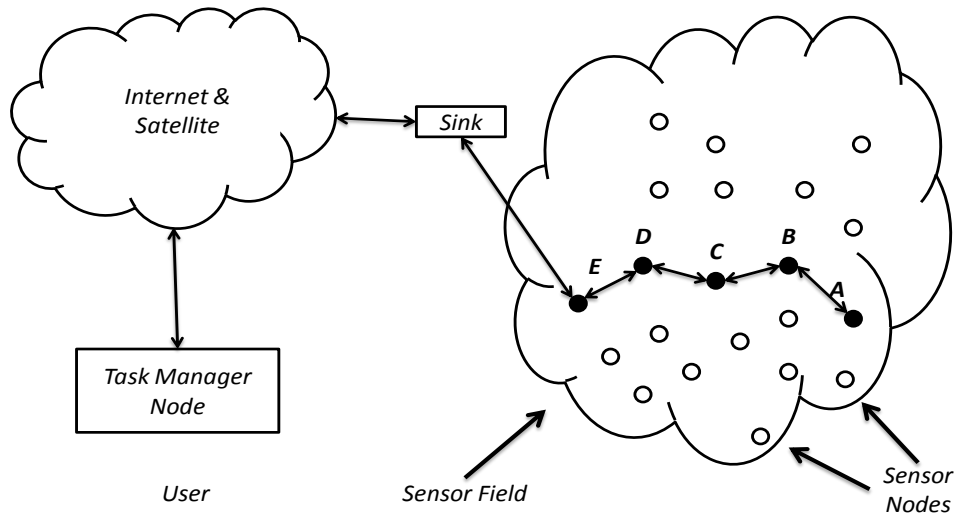


Figure 3: a typical sensor network. Based on similar figure from [1].

Security implications

The devices are capable of communicating information in relatively small packets [29]. They cannot transfer data fast enough [28] and the act consumes a disproportionate amount of resources [69]. In addition, the network is prone to various DoS-type attacks [33], including routing attacks, and it may be unreliable[28]. Therefore, every security protocol must be designed to work with small packets and be resilient. Ultimately, they have to be energy efficient.

Security mechanisms require inclusion of additional information to function. For example, TinySec[2] introduces a 5-byte overhead compared with TinyOS. Assuming maximum payload of 29 bytes, the security overhead is about 12%. This affects resiliency, energy efficiency and latency. In contrast, if the packet could carry 200 bytes payload then the overhead would be 2.5% and its impact would be equally smaller. Clearly, the small packet size of WSNs affects the operational margins of security protocols.

Resiliency will help the network to cope in cases where reliability issues cause the network to malfunction and when the network is under DoS-type attacks. However, resiliency is usually achieved by sending additional information, like parity-based error-correction[67] and replay of badly transported information [64]. Resiliency is therefore a trade-off with energy

efficiency and security mechanisms that operate at the transportation level have to account for this matter.

Security mechanisms have to select an appropriate trade-off between resiliency, overhead and level of security provision. Success in this problem achieves energy efficiency and ultimately increases the longevity of the network and widens the application possibilities. For these reasons, security mechanism to opt to use computational resources instead of radio because every bit transmitted is equivalent in energy costs to 800-900 instructions [69].

2.2. Security in Wireless Sensor Networks

This section explains what security is, its requirements and how they are applied in WSNs. A brief overview of security, and how it is applied in Wireless Sensor Networks, is given. Then the threat model is presented and the section concludes by explaining how different research groups apply security on different levels.

2.2.1 Overview of security

Definition

References to security in this work are shorthand for *Information Security*. The U.S. Code Collection [70] defines the term *Information Security* to mean “protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction.” The same document defines that *Information Security* should provide;

- Integrity; “guarding against improper information modification or destruction, and includes ensuring information non-repudiation and authenticity”
- Confidentiality; “preserving authorized restrictions on access and disclosure, including means for protecting personal privacy and proprietary information”
- Availability; “which means ensuring timely and reliable access to and use of information”

The above is an idealistic description of what a secure system should provide. Practice has shown that computer systems that are operated by humans cannot fully meet the requirements to be *unconditionally secure*. For this reason, a system’s security is considered satisfactory when it makes the system *computationally secure*. Various degrees of security like *unconditional security*, *computational security*, *ad-hoc security* and others are described by Menezes in [3].

Despite the limitations on reaching unconditional security, an acceptable *degree of security can be provided*, but there will always be trade-offs between the desired security provision and the practical limitations. Despite the fact that it is more correct to refer to security with relativistic expressions like *degree of security* or *level of security provision*, when a system provides an acceptable *degree of security*, it is said that this system is *secure*. The degree of

security a system must provide in order for it to be considered secure varies greatly and depends primarily on the aim of the system and its limitations.

Importance

Security is rapidly becoming an important part of the modern world. Be it protection against national threats or safe monetary transactions, security is important and here to stay [5].

Wireless Sensor Networks are no different from other computer networks in the level of security that they must provide. The literature that highlights the importance of security of all computer systems;

- every networked computer system is a potential target [6]
- security will make WSNs suitable to a wider range of applications [2, 5, 33]
- security and privacy might be legal requirements [45]
- applications may be critical [34] or valuable [35] enough to justify security

Each of the above references makes clear that the type of system is irrelevant to the security requirement.

Therefore, the security of WSNs has to be regarded as important as any other security system. The associated problems must be treated equally and the final solution must provide a security level that is comparable with other systems. After all, sensor networks are often parts of larger systems. If not equally secured they would become weak links on these systems.

Basic requirements

The aim of security in wireless sensor networks is no different that the aims for any computer network. As defined by [29], the basic requirements of a secure sensor network would be;

- Data confidentiality; information stored or in transfer on a sensor network should not be leaked to adversaries
- Data authentication; messages must be authenticated so that trust on them can be earned. Upon reception of a message, the receiver must be able to determine to trust or reject it
- Data integrity; a trust requirement is that a receiver must be able to determine if a message has been altered by an adversary while it was in transit
- Data freshness; messages must demonstrate to the receivers that they are not replays of older recorded messages

These requirements comply with the generic information security definition and they represent what a basic security system should provide. Provision of these requirements in a sensor network would protect it from the attacks described in the threat model³.

Desired level of security provision

Different applications have different security requirements; there is no level of security provision uniform for all cases. However, analysis of the literature points to the minimum level of acceptable protection against security threats.

The level of confidentiality is directly related to encryption strength [3] which is determined by the effective key length [4]. Acceptable key lengths are provided by various organisation and researchers. Since they are related to technological advances, they differ for each calendar year. In reference [71], NIST proposes that that 80-bit keys should not be used after the year 2010, a fact supported by [72]. In addition, this argument is indirectly supported by [73] in which CNSS stated in 2003 that 128-bits encryption is “*sufficient to protect classified information up to the SECRET level.*”

Authentication and integrity provides an assurance, or confidence, that the messages have not been forged or altered while in transit. It is suggested that the level of this assurance must be high enough to match the provisions of confidentiality [3].

As explained by [29], there are three levels of freshness; no freshness, weak freshness and strong freshness. References [29, 31] agree that most WSN applications require weak freshness and that it is up to the individual application to provide strong freshness when needed.

Since WSNs are limited, they should only be matching the minimum level of security but one might argue that even this is too high and unnecessary. Such argument does not properly consider the importance of security as discussed previously. In specific, it treats WSNs as less important parts of a system or as compelled to be unacceptably insecure. Nevertheless, attempts to provide WSNs with that level of security by a number of researchers are seen in the literature[30, 74].

³ Will be discussed next, in 2.2.2.

2.2.2 Threat model

In addition to the basic requirements, the individual threats need to be accounted in order to provide the desired level of security. This subsection describes what these threats are, how they are carried out by attackers and what the possible evasive solutions are. The threat model is discussed and proposed countermeasures are explained. Some of these solutions are more closely connected to this research and thus discussed in detail in the next section.

Attacks on confidentiality

The wireless communication medium that sensor networks use to communicate implies that any receiver in range is capable of reading all radio traffic [33]. Therefore, data must be sealed from eavesdroppers and this constitutes the requirement of confidentiality, which is met via encryption. Sufficiently complex encryption schemes are implemented in WSNs with care, as strong encryption methods are associated with high computational overheads [29]. Nevertheless, most systems that attempt to achieve confidentiality introduce encryption schemes with varying computational difficulty. Attacks on confidentiality are known as cryptanalytic attacks, discussed in [3], and are characterised by a level of required computational complexity before they conclude by revealing the encryption key.

The most known cryptanalytic attack is the known-plaintext attack, popularly known as brute force attack. It involves obtaining a plaintext and the associated output ciphertext. By knowing two of the three inputs of the encryption function, the attacker can try every possible key value until the correct key is revealed. Brute force attacks have a complexity equal to the effective key length of the encryption function. Longer keys provide greater confidentiality.

Other confidentiality attacks aim to reduce the level of computational complexity required by discovering a fundamental flaw in the mathematics underlying the encryption function. There are many types of cryptanalytic attacks in this category [3]. Cryptanalytic attacks usually require a number of conditions or pieces of the cryptographic input or output data to be acquired before they can be executed [3].

Cryptanalysis of encryption functions is the ongoing process conducted by the cryptographic community that aims to discover cryptanalytic attacks in existing algorithms. Using an

encryption function that is well analysed by the cryptographic community is a way to further protect the confidentiality of a secure cryptosystem.

Attacks on authentication and integrity

Communications via an open wireless medium allows attackers to alter or inject messages into the medium [33]. These attacks involve utilising a carefully crafted and powerful transceiver to overwrite the radio signals emitted by the network devices. Although it requires a skilful attacker, it is easier to conduct than a brute force attack on confidentiality, as it does not require great computational ability.

The efficient solution to this problem is to employ a cryptographic hash function to protect messages [3]. The function accepts the sender's identification information and the data as input and outputs a cryptographic signature. This is then appended to the data in the form of a Message Authentication Code (MAC). Each signature is associated with a probability of failure, known as confidence. Depending on the kind of hash function, the length of the input data and the amount of unique possible output, the exact probability that the protection might fail can be determined.

The hash function might be derived by the encryption function itself, as it was for example proposed in TinySec[2]. Under this solution, a possible attempt to inject authenticated messages or to alter existing messages would require knowledge of the authenticity mechanism's key, which in turn requires a cryptanalytic attack. Such authentication mechanism is believed to be as secure as the encryption function upon which is based [75].

Attacks on routing

Routing security is not a direct threat since most routing protocols can be protected if the security requirements for message exchange are met. However, many attacks become available if the security mechanism fails to protect the routing messages. These vulnerabilities are clearly demonstrated by Karlof *et al.* [76]. The majority of routing attacks is briefly described here. Many of the routing attacks described here might be classified as Denial of Service (DoS) attacks in the literature. Other DoS attacks are further discussed later.

If allowed to spoof, alter or replay routing information, an attacker can manipulate the routing table with bogus routing information. Successful manipulation allows the attackers to create a

number of problems including routing loops, attract or repel network traffic, extend or shorten routes, generate false error messages, partition the network, increase end-to-end latency etc.

Black hole and *Sibyl* [77] attacks aim to place a malicious node inside the network. This node will be able to choose whether to forward or drop a packet; it will therefore conduct a *selective forwarding* attack. The aim of the attacker is to provide a location on the network, which would look extremely attractive to the routing protocol and thus trick it to route traffic via the area under attack. Usually the method involves compromising a node or injecting nodes in the network. It therefore requires either insecure hardware or weak authentication.

A *wormhole attack* [78] creates and provides an actual high quality route between two points on the network using a communication medium normally unavailable to a sensor network. This route would quickly attract traffic and the attacker can then manipulate traffic in the way they choose. This attack requires the attacker to be able to inject packets or nodes in the network.

The HELLO flood attack utilises a powerful transceiver to broadcast a HELLO message that will trick distant nodes to believe that the attacker's node is neighbouring them. A laptop-class attacker can convince the whole network that they are a preferred route via a HELLO flood attack and thus it will make other attacks possible. This is another attack that requiring ability to inject nodes.

The final potential attack documented by [76] is *acknowledgement spoofing*. The attack involves recording and then replaying a legitimate acknowledgement message at the attackers will in order to pretend that a destroyed link is in good working order. That implies that the attacker can destroy nodes and hide that fact from the network. In addition, the attacker can trick the routing protocol and make it think that low quality routes are actually healthy routes. That would lead to higher energy consumption and increased latency. This attack requires the ability to inject acknowledgement messages.

Denial of Service attacks

The term Denial of Service (DoS) attack is loosely used to describe attacks that cause complete or partial system failure. Usually, the attacks involve sending a powerful

transmission in order to jam the radio channel or to confuse the medium access control protocol.

Brutal DoS attacks involve completely blocking the communication medium by causing interference. The low power radio capabilities of sensor networks mean that it is relatively easy to create a portable transmitter with enough power to jam the network [76]. There is no easy defence against such attacks [28], although some groups are working on solutions given particular assumptions [79].

A survey of intelligent attacks and countermeasures is provided by Wood and Stankovic [80] who describe DoS attacks that do not rely on raw power but flaws in the design instead. It might be possible for an attacker to inject a small, maliciously structured message that would exploit a design vulnerability of the system. The target of the attacker is to trigger asymmetrical resource usage, in the sense that the attacker can cause a great problem to the system with little effort on their part.

Physical attacks

Secure systems are vulnerable if an attacker obtains physical access in the hardware that hosts the system. This is as true for sensor networks as for every other system. Initial research in sensor networks suggested that they would be vulnerable to physical attacks including node tampering and reading of memory contents.

Such potential attack would be catastrophic for a secure cryptosystem that relies on keys as those have to be stored on memory and thus revealed to the attacker in case of node tampering. It is therefore essential to design systems that would sustain physical attacks as well. Research in this area has taken many directions depending on different assumptions⁴.

Attacks on single points of failure

By definition, every system will have a component, which would be weaker than the other components. Examples of such components in wireless sensor networks might be cluster heads and base stations. A carefully designed secure wireless sensor network should disclose as little information as possible on the location of such single points of failure. Additional, non-physical, points of failure might involve poorly designed components of the system.

⁴ This topic is further discussed in subsection 2.3.1.

The work of Deng *et al.* [81] analyses the security of the base station viewing it as a single point of failure. There are other examples in the literature that refer to the importance of single points of failure but very little published research considers just that.

2.2.3 Application of security

Security is a large area of research with lots of different branches. In sensor network security, there are at least three major branches: key management, data transportation security and routing security. There is also work that cannot be categorised, like hardware security or DoS attack prevention.

Each of these research areas attempt to solve security problems found at different levels. A uniform level of security must be provided to achieve effectiveness in the final solution. Any finalised security system will have to either be a combination of solutions from different areas or at least be compatible with other work. There may be no academic work assessing how various security systems could be combined to form a uniformly secure sensor network.

Key management

Key management is an important part of security in all computer networks [3]. In sensor networks, the term refers to the process of key agreement in a protected from adversaries method. In sensor networks, these keys are intended for use by the security mechanism of the data transportation layer.

Most key management schemes assume that that no component of the system can be trusted under almost any circumstance. These solutions expect that at least some component will be compromised and then used to launch an attack on the system. Sensor node hardware is believed to be susceptible to compromise, a problem known as *node compromise*.

Solutions found in the literature aim to limit the impact of a node's hardware compromise. There are relatively old [23, 82-83] and relatively recent [20, 84-85] proposals based on this assumption. The solutions might be categorised by type in probabilistic solutions and location-aware solutions. Other types of solutions and hybrids also exist.

Data-transportation layer security schemes

This research area attempts to provide security for the process of transporting messages between the sensor nodes. The solutions apply their security features on the sensor network equivalent of the OSI data transportation layer [66]. The systems are responsible for providing security properties like encryption and authentication for the data exchanged information.

These schemes are focused into providing energy efficient security and thus they rely heavily on the security of other components. Importantly, they ignore hardware and routing security. These problems are not considered relevant to this research area.

This research area is relatively more recent than other areas and there are not many well-cited solutions. There are no distinct categories in this research area.

Routing security

Routing refers to how a sensor network is organised and how available message paths are defined. This process involves exchange of information which if not secured might enable DoS attacks.

There are a number of proposals tackling the generic routing problem [17, 24, 65, 86-89] but most research is focused on providing energy-efficient routing. Surprisingly and despite the fact that most routing protocols seem vulnerable [76], very few articles consider both security and efficiency of routing.

Other areas

There are also open security questions in hardware security, prevention of denial of service attacks, management of points of failure and most importantly: cryptanalysis and benchmarking of encryption algorithms. All these potential problems are related to the security of wireless sensor networks but not directly related to this research.

2.3 Related work

This section describes and critically analyses important work seen in the literature that relates to the proposed system. The proposal is primarily a security mechanism acting on the data transportation layer but it includes a key management system as an important part of the solution.

This section starts with an overview of the research area, explaining how security is applied in wireless sensor networks. Then it presents work that is both important and directly related to this proposal; key management and data transportation layer security schemes.

This section concludes by discussing routing security and other work, which is related to security in sensor networks but not directly related to this proposal, like routing security and other work.

2.3.1 Key management schemes

This subsection presents established key management schemes found in the literature.

Probabilistic key management

Probabilistic key management schemes work by clustering the network into virtual groups of nodes. Nodes in a group share the same security state but groups benefit from different states. This design limits the impact of node compromise to affect only its group instead of the entire network. The way distinctive groups are created and managed is what differentiates the solutions. Most of these proposals assume that the sensor network is large and dense.

The work of Du *et al.* [23] is one of the oldest and probably the most cited. They have described a system in which nodes are pre-loaded with keys randomly selected from a large pool of keys. The pool of keys has to be larger than the size of the network but the number of keys loaded is much smaller than that. The density of the network, i.e. how many neighbours a node will have on average, defines how many keys should be loaded in each node. For a given the pool size, network size, density and number of keys loaded on the nodes, the

probability of two neighbouring nodes sharing the same key can be calculated. A share probability greater than one guarantees that the vast majority of node pairs will be able to discover a common key loaded on both ends of the pair and secure communication can commence. On the other hand, attackers who compromise one node can only obtain a limited number of keys, which does not allow them to intercept the communications from the whole network.

A similar but more efficient approach is proposed by Pietro *et al.* [84]. The main difference is that nodes are loaded with keys from the pool while obeying a mathematical relation, which allows all nodes to know in advance which keys are loaded on every other node. This is done with the help of a pseudo-random number generator, which accepts the node ID as seed, effectively selecting keys using the node's ID as a selection criterion. Therefore, anyone who knows the node ID also knows the keys that are loaded on the node and thus nodes know in advance which key to use for communicating with another node, without the need of a key agreement process, saving key-agreement radio communication overheads.

Location-aware key management

Discovering the physical location of a node is rather straightforward for advanced nodes that incorporate GPS receivers in their sensing board. This information is a valuable asset, which can help improve not only efficiency and information gathering but security as well. The following are a set of different proposals that enhance the security assuming that the network's physical topology is known.

The authors of [82] claim that by using location awareness they can achieve graceful degradation of network performance while nodes are gradually compromised. The method works by dividing the deployment area of the network in a virtual grid. Each cell of the grid may contain some nodes, which have a certain relative position on the network and use keys, which are shared only by their neighbours to communicate. The grid is determined by using the range of the nodes to identify which nodes are nearby and then the cell keys are derived from a master key, which is in turn deleted. If a potential attacker compromises one node in a cell of the grid, they only take control over that cell, instead of the whole network. More compromised nodes mean more compromised grids as well but the process is gradual and not destructive after a certain threshold of compromised nodes is reached.

The work of [20, 85] relies on robots or robot-assisted manual determination of node location and of the keys that will be loaded in the nodes. After the nodes are randomly scattered in the environment the area is traversed by robots who note the location of the nodes and load keys on them using rules defined by the method. The system can provide highly enhanced security, even compared to public key cryptography but unfortunately, it involves the difficult requirement of utilising robots.

Other proposals

The literature includes proposals that are simpler than those discussed previously but still expect network hardware to be insecure. An example is provided here for completeness.

The proposal of Zhu *et al.* [83] works by categorising types of messages, and security requirements of these messages. The authors distinguish four types of messages and they establish four types of keys:

- an individual key shared between each node and the base station
- a pair-wise key shared between other nodes
- a cluster-key shared with multiple neighbouring nodes
- a global key shared by the whole network

They also provide a protocol for weak local broadcast authentication, based on the use of one-way key chains. The authors claim that they can use a master key for few seconds before an adversary can capture a node and obtain it. They use this key immediately after deployment to involve a simple handshake and then they derive a unique pair key using the information exchanged in the handshake and the master key. Finally, they erase the master key. After initial negotiation and agreement on the pair keys, the authors build upon this secure platform to share all the other keys they define in their mechanism.

Discussion

The hardware's insecurity assumption that leads to the development of complex key management schemes has been disputed in other work in the literature. There is strong evidence that the assumption is both unrealistic and too extreme for pragmatic wireless sensor networks. Notably, Gamage *et al.* [16] have strongly expressed their objections to both the realism of the assumption and the feasibility of systems that attempt to accommodate it. In

addition, the work on references [90-92]⁵ indicates that hardware can be secured, effectively solving the problem of complex key management.

In addition, even if complex key management is adapted, the sensor network will ultimately fail. Complex key management schemes can only delay the ultimate failure of the system and maybe allow for more time for corrective action to take place but they cannot infinitely do so. Therefore, complex key management will eventually fail as well.

In support to the above arguments, there is a clear tendency to ignore this problem by many other authors [2, 21, 30, 93-94]. This research sides with these authors and chooses not to actively seek a solution to the supposed hardware security problems.

Implications

Developers of wireless sensor networks are likely to agree with Gamage *et al.*[16], that complex key management is not necessary. However, there are few alternative, more simple, options, for example [83] which is probably too simple.

There is no key management mechanism in between the two options. A mechanism that would;

- not use the same key for large groups of nodes
- automatically change keys when they are likely to become a security liability
- keep its state secret and operate without leaking information over the radio
- not require storing large numbers of keys on the sensor nodes

Such hypothetical simple key management system would be less complex than other key management systems while it would provide better security than lack of key management. It would also require fewer resources than complex key management systems.

One might argue that data-transportation layer security mechanisms do not need to provide key management as this can be done at application level. Therefore, provision of a key management is unnecessary. However, a different key management system means that the cryptosystem is not designed comprehensively; it relies and trusts the security of other subsystems and it may therefore be inconsistent.

⁵ Discussed in 2.3.4.

Consequently, the data transportation layer security schemes in references [2, 21, 30, 93-94], which are further discussed in the next subsection, might not perform securely if they are used in conjunction with the wrong key management scheme.

2.3.2 Data transportation layer security schemes

This subsection presents the three established data transportation security schemes found in the literature. The solutions are presented in chronological order; SPINS, TinySec, MiniSec and SenSec. These mechanisms form the basis of comparison for this proposal, which is also a security mechanism for the data transportation layer.

The SPINS proposal

SPINS [29] by Perrig *et al.* published in 2002 was the earliest attempt to evaluate and implement secure wireless sensor networks. The work provided two protocols, a network encryption protocol named SNEP and μ TESLA, an authentication protocol. At the time of development, sensor networks were expected to suffer from unrealistically low resources and thus the authors discuss if a secure sensor network is even a realistic target.

To facilitate the realism argument, the authors list a set of requirements that a system must fulfil in order to be regarded as secure⁶. Briefly, they are confidentiality, authentication, integrity and freshness. They define that nodes should never be trusted, they rule out public-key cryptography as an option and then they proceed into describing their protocols.

From an operational point of view, the authors identify base station to node (broadcast) communication as the fundamental communication primitive but they also distinguish node to base station and base station to node communication as possible patterns.

Their security protocols are regarded as security primitives, building blocks. They are provided in the basis that future work will build upon them to achieve the security requirements. SNEP provides for all requirements except authenticated broadcast. It operates

⁶ The full requirements are discussed in 2.2.1.

by encrypting messages using a private key shared by the communication pair. In addition, it uses a secret counter, which is incremented by 1 after each communication but not transmitted with the message. μ TESLA uses a symmetric encryption mechanism to authenticate messages from a small number of authenticated broadcasters. Both protocols seemed promising but they were never implemented[2].

The TinySec mechanism

TinySec [2] was the first protocol to ignore hardware security, ignore key management and simply provide security at the data transportation layer. It is currently considered the standard security mechanism for sensor networks by many literature articles [95-97]. TinySec introduces about 10% increase of energy consumption compared with unsecured TinyOS operation.

The protocol is built as an extension to the TinyOS platform and it alters the original packet format to facilitate the security features. TinySec meets each of the security requirements set by SPINS [29]. It achieves confidentiality by encrypting message contents using SkipJack [98]. Authenticity and integrity is facilitated with a 4-Byte message authentication code (MAC) which is also generated by the SkipJack cipher under the CMAC standard [99]. Finally, freshness and semantic security are achieved via an initialisation vector.

TinySec provides two packet types in an attempt to increase energy efficiency: the *TinySec-A* packet, which only provides message authentication and the *TinySec-AE* packet, which complies with all the basic security requirements.

The MiniSec mechanism

MiniSec improves the performance of TinySec, is more resilient to network error and uses a different approach to generate and transmit the initialisation vector. MiniSec uses SkipJack in OCB [100] mode of operation as encryption function. This mode of operation provides encryption and authentication. MiniSec seems to be an equally secure optimisation of TinySec while it otherwise offers very few innovative security features.

MiniSec greatly optimises the way the initialisation vector (IV) is implemented by using a number of solutions including overloading, Bloom Filters [101], epochs and time synchronisation protocols [102-103]. In further optimises the efficiency of the network by

introducing packets that are suitable for different communication types. Figure 4 illustrates the security-related radio overhead of various solutions.

The SenSec mechanism

The SenSec[30] framework is promoted as a TinySec alternative as well. It is solely based on TinySec and aims to improve its security provision and some of the performance. SenSec employs a packet format which is very similar to TinySec [2] but claims that it is slightly better than it. The major difference in packet format from the TinySec scheme is the way the IV (Initialisation Vector) is constructed.

A custom and improved variant of SkipJack [98] is used in SenSec. The customisation claims to provide 144-bit security and is based on the DES-X [104] method. They call this variant SkipJack-X. They also aim to reduce the computation cost of MAC processing by using a one-pass MAC computation mechanism. They claim that their MAC mechanism is secure as long as “the total amount of packets being encrypted and authenticated with the same key is much less than 2^{32} .”

SenSec defines a hierarchical access control scheme divided in three levels and thus they use three levels of keys; global key, cluster key and sensor key. By using these three keys, they can produce three packet types for use in the appropriate context. They claim that this method is resilient to node capturing attack as well since revealing the keys in one node will not compromise the whole network but only one group of nodes.

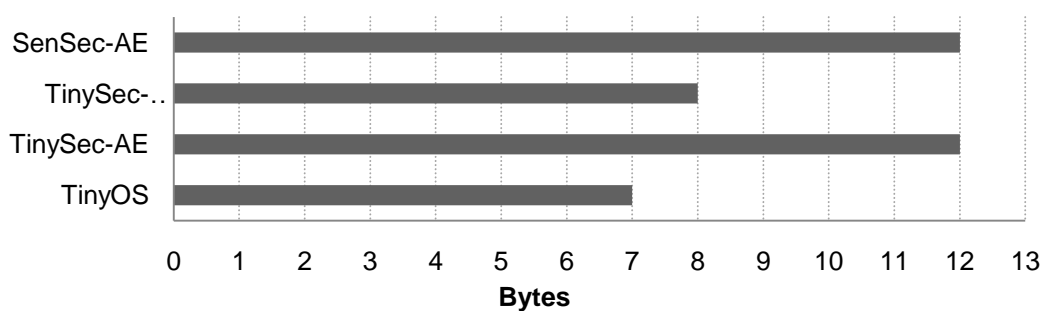


Figure 4: Packet overhead of TinyOS, TinySec-AE, MiniSec-U and SenSec. Numbers represent the size of packet’s header and MAC/CRC in bytes.

Other mechanisms

As this research was being conducted, more security schemes have appeared in the literature. Recent mechanisms like, ContikiSec [93] and FlexiSec [94] will be discussed in Chapter 6.

Finally, an alternative solution to data transportation security is ZigBee [105]. It is the lightest, fully developed security standard for small devices and it might be suitable for high-end sensor boards but it is still extremely costly for the low-end devices that this project targets.

Discussion

TinySec and MiniSec offer 80-bit complexity which is deemed unacceptable by 2010 [71-73] but both research groups claim that their mechanism can be used with any encryption algorithm and thus the possibility of higher complexity is not ruled out. The encryption function is indeed a black-box type of component, which can be upgraded without redesign of the system. However, the related implications in energy efficiency have to be taken into consideration. Should such an update be ever implemented, the mechanism would have to be re-evaluated and comparisons with other mechanisms would need to be discussed again.

On the other hand, the authors of MiniSec apply the key whitening technique to SkipJack and they call it SkipJack-X. Key whitening is an anecdotal term coined by J. Rivest and it is briefly explained by Schneier [4]. The scheme is better than plain SkipJack but not as good as a cipher that would natively offer greater cryptographic strength. This solution poses two problems, which are discussed next.

The first problem regards the effective key length. According to Schneier's explanation, SkipJack-X should increase the complexity to $2^{n + (m/p)}$ where n is the key length, m is the block size and p is the number of known plaintexts. Therefore, SkipJack-X does offer greater complexity but the true effective key length is reduced if the attacker is able to obtain a number of key lengths. Sensor networks whose purpose is to report a single value to the base station allow the attacker to guess possible plaintexts and reduce the complexity to unacceptable standards.⁷

⁷ Suppose that a network measures ambient temperature. The attacker can use a thermometer to find the temperature that nodes should be reporting and monitor the network for 24 hours. Using the method, the attacker might be able to obtain 10 plaintext-ciphertext pairs. That would reduce the complexity to $\sim 2^{87}$ bits, way below acceptable standards of 2^{128} bits.

Secondly, the approach might be problematic because the specific application of key whitening for SkipJack has not been crypt-analysed. There is no information on the security of SkipJack-X on the published literature. It is unknown if other limitations of the SkipJack cipher might cause further reductions to complexity and it is unknown if whitening can protect the particular cipher from future cryptanalytic attacks. If that is true then MiniSec might not provide the required 128-bits of complexity.

It seems that no existing data-transportation security mechanism is up to date with modern security requirements and that none provides a comprehensive solution, as they lack key management.

Implications

It is rather clear that a system that would provide an acceptable level of confidentiality, authentication and integrity is missing from the current literature. Even more modern mechanisms than the established TinySec, MiniSec, SenSec triplet, do not offer 128-bits encryption while simultaneously being capable of operating in low-end devices like the MICA2 node.

This research aims to provide a mechanism that would provide acceptable level of security and meet modern requirements for encryption complexity.

2.3.3 Routing security

Secure routing protocols

There are many routing protocols proposed for use in wireless sensor network but most of them are proved insecure by Karlof *et al.* [76] in their 2003 survey paper. Since then a number of new proposals attempt to tackle the problems revealed. A few recent and more secure protocols are presented for completeness in this subsection.

Wood *et. al* describe a protocol family called SIGF in [106]. It consists of two configurable and secure routing protocols. Along with the traditional routing duties, the authors claim that their protocols are secure against some of the problems identified by [76]. The protocol family does not store routing information on a table and therefore attacks that manipulate the routing

table with bogus entries are not applicable. In addition, wormholes and HELLO floods are not possible due to the dynamic route selection of SIGF. However, the authors admit that protection against other attacks is impossible without a reliable authentication mechanism.

Finally, despite being primarily a key management scheme, the LEAP+ mechanism [83], supports the use of authenticated acknowledgement packets.

Discussion

The work of Karlof *et al.* [76] does not take secured networks into account. They made this an important assumption when they said, “*because sensor networks use wireless communications, we must assume that radio links are insecure.*” They also state a number of other assumptions that equally disregard the possibility that a sensor network can be secured. The reasons why this assumption was made in 2003 are understandable, but ironically, it was Karlof *et al.* who published TinySec [2] a year later, effectively proving their own assumption wrong.

If a security mechanism existed that would provide a combination of the security features provided by TinySec, and SIGF and if it was operating on secure hardware, then each routing attack described by Karlof *et al.* is impossible. Table 1 provides a summary of each known attack, the vulnerability it targets, how it can be prevented and which security mechanisms are not vulnerable to this attack.

Attack	Requires ability to	Prevented by	Offered by
spoofed, altered, or replayed routing information	inject messages alter messages replay messages	authentication integrity freshness	TinySec MiniSec SenSec
selective forwarding	node compromise or node injection	hardware security or authentication	
sinkhole / black hole			
sibyl attack		authentication	
wormholes	unauthenticated broadcast	broadcast authentication	
HELLO floods	unauthenticated ACKs	ACK authentication	SIGF
acknowledgement spoofing			

Table 1: Routing attacks, their requirements and how they can be prevented.

An interesting observation can be made; all that is required to protect a wireless sensor network from all routing attacks is to secure all communication, including acknowledgement packets, with a security system that would provide authentication, integrity and freshness. As long as that stands true, and as long as hardware remains secure, routing cannot be compromised.

Implications

None of the current data-transportation layer security system provides authenticated acknowledgements. Potential developers who want resilient routing are able to select from three data transportation security mechanisms but they would have to use SIGF in order to protect acknowledgements, there is no alternative.

This research aims to provide a mechanism that would offer total protection of all messages including acknowledgements, broadcast messages and normal messages, regardless of the layer they originate. That solution would completely protect the network from all known routing attacks.

2.3.4 Other work

This subsection lists some indirectly related work for completeness and as a basis of argument for subsequent sections. This work is important but it is not directly comparable to this research. Therefore, no critical evaluation or any further analysis is made here.

Hardware security

The area of hardware security has little connection with computer science but it interests this research since the proposed solution does not tackle hardware security problems.

There is great debate on whether hardware is secure or not and great antithesis in the claims of research papers in the literature. Some articles describe a number of microprocessor attacks while also advising countermeasures [90, 107]. There are also software-based solutions to the problem [108].

On the other hand, the microprocessor manufacturer Atmel [70] claims that the ATmega128L microprocessor is secure [46]. This microprocessor is used in many sensor nodes, including the MICA2 node.

Nevertheless, a significant research interest towards solving this problem is observed. In addition to sensor networks, other microprocessors are rapidly becoming secure and are used in everyday life, like for example in Credit Cards. Security of such applications is very important.

Denial of Service

As explained in the threat model, it is difficult to defend against brutal DoS attacks and thus current research is focused on reducing the implications of such attacks. Most approaches consider DoS attacks that affect some parts of the network and propose detecting and procedures to minimize attack impact. All these solutions work under a number of assumptions that might not be applicable to all sensor networks.

Gu *et al.* in [109] propose a system to detect a search-and-destroy DoS attack if nodes are equipped with a means to detect that they are about to be destroyed, e.g. accelerometers. When such an attack is detected, an alarm can be sent to signal to neighbouring nodes that it is under attack. Other nodes can take evasive action.

Another DoS detection system uses game theory to detect when nodes behave maliciously [110]. The methodology uses a set of parameters and a method to assign reputation to nodes. The network isolates malicious nodes if their reputation declines. The authors model and evaluate the methodology to determine a node's reputation.

Points of failure

Wireless sensor networks are systems subject to the usual weakest-link and single point of failure problems. Some of the physical or virtual components of the network might fail more easily or might cause disproportionate damage. There is research work that attempts to evaluate such threats at specific locations on the network.

Examples include the work of Deng *et al.* [81] which proposes and evaluates options available to protect the security of the base station which is presented as a single point of failure. The

authors propose and evaluate three mechanisms that introduce a degree of difficulty in identifying the location of the base station: multipath routing to multiple base stations, confusion and encryption of the identification fields and relocation of the base station.

2.4 Discussion

This section acts as a summary of what was presented in the literature review. The section starts by defining a set of baseline requirements, a presentation of what is expected from this research. This section also explains the aims of the basic requirements and the expected benefits of meeting them. This section concludes by presenting the associated challenges.

2.4.1 Baseline requirements

All existing approaches suffer from problems that range from unrealistic assumptions to insufficient security provision. A more modern solution that will benefit from the latest research and technological advance is required.

Please note, not all the requirements and aims presented in this subsection were met in the final solution. Some of them were met only under certain conditions. This subsection is only a guideline of this project's intentions. Please read Chapter 6 for evaluation of the final design.

Key management requirement

A key management mechanism that would be designed under the assumption that hardware cannot be tampered needs to be provided. The mechanism should:

- provide different keys for small groups of nodes
- store at most as many keys as distinct communication endpoints
- frequently change the keys preferably after each time a key is used
- keep its state secret and not leak information that would assist in cryptanalysis

A key management mechanism as such would provide strong semantic security that would not rely on initialisation vectors which are potentially unsafe [111].

In addition to being a security risk, current systems transmit IVs over the radio. TinySec transmits 2 bytes, and SenSec transmits three bytes. The described key management would not need to transmit IVs then the energy required to transmit the IVs would be saved.

Routing security requirement

A security mechanism that would primarily protect data transportation but will also prevent most routing attacks needs to be provided. This requirement can be achieved by authenticating every communication without any exceptions, including acknowledgement packets.

Protecting the acknowledgement messages is a complex task and it is expected to require additional radio transmission. On the other hand, it would enable the developers of each network to choose a routing protocol that fits best for their network characteristics. Currently, their only option is to use SIGF but it might not be the most energy efficient routing protocol for all types of networks. Potentially, the energy gains from selecting an appropriate routing protocol will be much greater than the small energy losses that will be required for authenticated acknowledgements.

Cryptographic strength requirement

In order to achieve the acceptable [71-73] level of security provision, 128-bits encryption strength is required. The requirement must be fulfilled using an encryption function that was natively designed to provide this level of complexity. The solution should also provide a similar level of authentication and integrity.

Additional requirements

The mechanism should be:

- similarly or less energy demanding than existing proposals
- easy to deploy, not require changes in existing application code
- demonstrated on code partially or as a complete mechanism
- based on open and well known standards and provided free of charge

2.4.2 Aims

Security aims

The baseline requirements aim to elevate the security provision of the proposed mechanism by providing:

- means for authentication and trust establishment
- an evolving security state
- semantic security and weak freshness without security risks
- resilience against the effectiveness of some cryptanalytic attacks
- protection against known routing attacks
- acceptable level of cryptographic strength for all related components

Energy performance aims

The longevity of the network will be improved by supplying more efficient mechanisms for:

- key management
- provision of weak freshness
- provision of semantic security
- an overall more efficient design

Other aims

The following should be achieved:

1. The integrated key management will allow evaluation the security mechanism as an comprehensive system
2. Authenticated acknowledgements will enable the base station to obtain health status reports of multiple nodes by simply sending one packet
3. The ease of use requirements will make the mechanism attractive to existing installations, increasing the potential for adaptation
4. A direction to open protocols allows comparisons with other mechanisms , promotes the advancement of knowledge and achieves low cost of ownership

2.4.3 Research Challenges

Comprehensive authentication

Provision of authenticated acknowledgements is rarely seen in WSNs but it is common in other types of networks. It is easy to authenticate forward going messages but it is energy consuming to assure the sender that the receiver has received its message.

Significant radio overhead will probably be introduced by this requirement and the necessary energy savings must be found on other components in order to satisfy the requirements.

Strong encryption

Another challenge is the provision of 128-bit strong encryption under the limited nature of sensor devices. Selecting such an encryption function is not a straightforward process.

An adequately performing cipher has to be selected after considering a number of criteria. Important criteria include the computational overhead, the block size, memory requirements and the security of the cipher. Additional criteria will be set in the relevant chapter⁸.

Satisfaction of conflicting requirements

The defined solid requirement on better security with similar energy efficiency describes two conflicting requirements. Provision of an appropriate solution is an important research challenge.

This research attempts to solve this problem by using carefully selected components and a better design. If successful, the solution will form the primary contribution to knowledge.

⁸ See Chapter 3.

Innovative key management

The key management proposal promises many benefits but it has to remain relatively simple and very energy efficient. It has to be simultaneously energy efficient, rich in features and operate without radio overhead.

Other systems satisfy some of the SecRose requirements but there is no evidence of anything similar to this proposal in the literature. The closest match is SPINS [29], which does not introduce radio overhead but satisfies only a fraction of SecRose's aims.

Key management design and implementation will be a unique research and engineering challenge and that it will complement existing knowledge.

2.4.4 Summary

Main requirements

The baseline requirements are provided:

- provide an acceptable level of overall security
- protect against currently available attacks
- operate consuming equal or less energy than other solutions
- allow ease of adaption and deployment with minimal costs

General aim

We aim to develop, document and implement a replacement for current data-transportation layer security mechanisms. The proposed system will have to perform equally or better than existing solutions, regardless of the basis of comparison.

Main challenges

- design the unique key management system in accordance to conflicting requirements
- provide complete protection of every communication, including acknowledgements
- select a suitable encryption function that offers the required level of encryption strength
- provide a secure, efficient and easy to deploy overall design

Chapter 3

System requirements

3. System requirements

This chapter defines security, energy efficiency and non-functional requirements for the proposed mechanism. The rationale underlining the selection of each requirement and the consequent benefits or drawbacks are discussed.

3.1 Basic security requirements

This section determines basic security characteristics of SecRose. As discussed in the literature review, there are a number of security requirements [29, 112] and possible threats [76-78] to wireless sensor networks. Each requirement and its protection it offers against threats is discussed.

3.1.1 Confidentiality

Requirement

SecRose must provide the accepted level of confidentiality, specified to 128 bits key length, for all information payload data when a packet is transmitted to the radio medium.

No limitations are set as to what additional packet information might be protected. It is desirable for SecRose to obscure by encryption or other means as much packet information as possible, as long as the act does not pose additional energy overhead.

Rationale

As discussed before, sensor networks need to be protected by a security level that is equal to other computer systems. The current acceptable confidentiality complexity level requires 128-bit long encryption keys.

3.1.2 Authentication and integrity

Requirement

SecRose must guarantee that all information present to any received packet, including acknowledgement packets, is legitimate information that originated from the claimed sender, and was delivered intact to the receiver.

Acceptable level of confidence for this guarantee is defined as that which exceeds the energy or lifetime capabilities of high-end sensor networks. A potential authentication or integrity breach attack must cause the network to deplete its energy resources or take longer than the lifetime of the network to succeed.

This guarantee may be provided in any way and is not limited to a Message Authentication Code (MAC). Packet fields may not be included in the MAC if the design guarantees that MAC validation will fail when authentication or integrity of these fields is breached.

Rationale

The authentication and integrity requirement aims to protect the network from a range of attacks, including; injection of bogus or altered packets, injection of broadcast and ACK packets, injection of attacker-controlled nodes and malicious packet replays. These attacks can introduce serious implications in the network's routing and the validity of reported measurements, allowing the attacker to completely destroy, alter or cause confusion to the network.

The level of confidence provided by authentication and integrity is a trade off with efficiency. Therefore, the confidence of the guarantee is defined pragmatically; the attack would be meaningless if it depletes the network's resources or if it cannot succeed in time. This is adequate provision that would guarantee authentication and integrity while consuming the least possible resources.

3.1.3 Freshness

Requirement

SecRose must provide weak freshness; a means to determine whether a packet is definitely not a replay of previously broadcast packets. Nodes should be able to designate packets as fresh or as undetermined. For packets that are determined as fresh, the level of confidence must be equal to the authentication and integrity guarantee.

SecRose is not required to provide information on the exact time elapsed between transmission and reception of a packet.

Rationale

Provision of a weak freshness guarantee is required by most applications as it prevents some routing attacks and reporting of false information. Strong freshness, which is not required by all applications, involves inclusion of a timestamp on packets thus allowing the application to know exactly when a packet was initially transmitted. Strong freshness is not required by all applications or for every packet. Provision of either type of freshness is an energy-demanding requirement.

SecRose is required to provide weak freshness for all accepted packets and indirectly allow the provision of strong freshness to applications that demand it.

In conjunction with the key management mechanism⁹, the weak freshness provision will be implemented without any additional energy overhead and will be secure for attackers with realistic computational abilities.

Applications that require strong freshness may include timing information on the packets and then rely on the authentication and integrity requirements for secure delivery of this information.

⁹ Discussed in 3.3.1.

3.2 Additional security requirements

This section clarifies some security issues and describes what is required to solve them. These requirements are distinct from basic requirements as they are influenced or derived by a number of factors.

3.2.1 Regarding routing security

Requirements

SecRose must meet its basic security requirements for all communication between sensor nodes. This requirement is not limited to application packets but it also includes routing packets and transportation control packets, like acknowledgement packets.

SecRose is not a routing protocol and routing functionality is not required, but its compliance with existing routing protocols must be evaluated.

Rationale

By meeting this requirement, SecRose will effectively seal the system from known routing attacks, as presented in 2.2.2 and in references [76-78].

Compliance with existing routing protocols is essential to enable the developers of WSNs freedom of choice on selecting between various routing protocols. SecRose is expected to enrich the list of usable routing protocols.

3.2.2 Availability

Requirement

The design of SecRose must contribute to the overall availability of a sensor network by not introducing weaknesses that would enable asymmetric DoS attacks and by not introducing single points of failure. With the exception of routing, SecRose is not required to protect, detect or prevent threats to availability.

Rationale

Sensor networks are prone to a number of availability threats. Most of these attacks are tackled by their own research fields and are therefore outside the scope of SecRose. The essence of the availability requirement for SecRose is not to introduce additional weaknesses that would make availability attacks easier.

3.2.3 Regarding security by design

Requirement

SecRose is required follow the principles of security by design. SecRose will have to be an open mechanism and the design will be available for everyone to examine and improve.

Rationale

Security protocols should be based on a secure design. SecRose should follow this principle. Obviously, this does not refer to the run-time cryptographic state, e.g. the secret key, but on the design of the mechanism itself.

3.2.3 Hardware

Requirement

SecRose is required to ignore potential hardware security problems. SecRose will assume that its hardware is trusted and secure.

Rationale

SecRose aims to be based on pragmatic assumptions. The assumption that hardware is not secure has been disputed and attempts to overcome this assumed problem introduce significant resource utilisation overheads. SecRose will belong to the group of solutions that do not attempt to solve this problem and act as a lighter alternative in networks where hardware security is assumed.

3.3 Operation and efficiency requirements

3.3.1 Key management mechanism

Requirement

SecRose must include a key management mechanism. The ideal target of the mechanism would be to alter every cryptographic key after it has been used once to encrypt a message. Deviations due to operational conditions are allowed. Key management will enable a number of other described requirements to be achieved.

Key management must synchronise without transmitting information over the radio. When communication of a packet occurs, key management will change keys using a secret value, derived by the last packet content. Knowledge of the used key will be required in order to derive a new key. Over time, each communication pair will evolve its own state, derived from the contents of communicated packets.

Fail-over mechanisms must be utilised to ensure successful synchronisation and continuous communication in case of accidental or deliberate radio interference that would cause packets not to be communicated properly. This is the case where deviations from the ideal target are allowed.

Rationale

The key management system will serve as a core component with efficiency and security objectives. It will provide an efficient platform to achieve a number of requirements that would normally require radio overhead but it would only introduce computational and memory overhead. Key changes will achieve semantic security, by changing at least one part of the cryptographic input and if the changes are frequent then weak freshness can be provided as well.

3.3.2 Efficiency requirements

Requirement

SecRose is required to perform equally or better than TinySec[2] when average values of the following metrics are compared;

- energy consumption overhead
- introduced latency
- memory requirements

In addition, SecRose must indicate, via a theoretical evaluation, that it would also match or outperform the efficiency of other mechanisms found in the literature, such as [21, 30, 93-94].

Rationale

Security in WSNs is constrained by the limitations of the devices. SecRose plans to impose no additional overhead than other solutions while it will improve the security provision and efficiency. This requirement must be met in order for SecRose to achieve its objectives.

3.4 Other requirements

This section describes requirements related to assist with the successful selection, installation, maintenance and future updates of SecRose. The requirement for a proof-of-concept is also discussed.

3.4.1 Essential deployment requirements

This subsection defines a set of requirements that aim to increase the marketability of SecRose to future implementers.

Rationale

Changes to existing systems require planning, time, cost and effort. The allocation of human resources and the associated costs of such process would deter sensor network owners that want to deploy a more secure mechanism.

The requirements in this subsection complement each other in order to facilitate rapid and easy migration from to a secured network. They also give an advantage to SecRose since alternatives require changes in the applications¹⁰.

Finally, the customisation requirement is given for those sensor networks that require higher security and have to commit to the associated problems of adapting a security mechanism.

Please note that SecRose is not required to be backwards compatible with other security mechanisms. The requirement assumes that the networks under update are operating with TinyOS in its original form.

¹⁰ For example, TinyOS does not tackle freshness at all, it leaves the problem to the application

Backwards compatibility

SecRose must replace the existing transportation mechanism of a WSN in a manner that is completely transparent to higher layers. SecRose must accept exactly the same input and provide exactly the same output as the TinyOS operating system. By no means should an application have to be re-programmed in order to accommodate the introduction of SecRose.

Preconfigured distribution

SecRose's distribution package must assist sensor network operators that have minimal experience with security systems to secure their network easily and rapidly.

SecRose should readily provide a level of security to every sensor network within which is deployed, without requiring complicated configuration. Thus, it must ship preconfigured with sensible parameters that would allow secure operation for most applications.

Please note that the pre-configuration options described here will be used for all direct comparisons in Chapter 6.

Customisation

Some sensor network applications demand higher security. Should these networks decide to adapt SecRose, they would have to plan before they deploy it on their network. SecRose must be fine-tuneable to allow their requirements to be accommodated.

3.4.2 Other requirements and desirables

This subsection covers the important requirement of providing a proof of concept demonstration for SecRose. It also discusses low importance, low priority and out of scope requirements that would be nice to achieve in SecRose but cannot be provided by this research project. They are documented in order to direct a design with future targets in mind.

Proof of concept

SecRose must come with a proof of concept implementation that would cover communication between two nodes. This demonstrative implementation will be used for evaluation of SecRose. It must be complete enough to allow for a fair comparison but it may not be developed beyond this point.

Scalability

SecRose's design should not pose fundamental limitations that will prevent future developments from scaling SecRose up to use it in larger networks. This will allow SecRose to scale up and function on larger or different networks.

Chapter 4

SecRose Specification and Design

4. SecRose Specification and Design

This chapter describes the SecRose data-transportation layer security mechanism. Algorithmic and systematic descriptions are provided. Section 4.1 defines the protocol and describes how it operates to achieve secure communication. Section 4.2 gives a systematic explanation of system, its components and their interactions in order to facilitate the protocol. Rationale of the various decisions is given in section 4.2.

4.1 Algorithmic description

This section gives a plain specification of the SecRose protocol and its operation without discussing rationale or other matters extensively. The section begins by defining the assumptions and some concepts and then specifies the protocol's operation step-by-step.

4.1.1 Specification of concepts

Assumptions

SecRose assumes a network consisting of a few hundred nodes. The nodes run on hardware similar to the MICA2[48] sensor node; 8MHz processor, 4KB application memory, 128KB runtime memory, 512KB permanent storage memory and 38.4Kbps radio communication capability. The nodes are running the TinyOS operating system [31].

Nodes use tamper-resistant microprocessors such as the ATMega128 microprocessor found in MICA2 nodes. Thus, in case a node is captured, the attacker is unable to extract data from the sensor, as specified in the microprocessors technical manual [46].

All nodes are loaded with a 96-bit long random string before their deployment. This will be used by SecRose's key management as the *initial key*. In addition, they are loaded with a 32-bit long random string, which will be the *initial value* of the counter.

Packet categorisation

The communication patterns within a sensor network will be categorised for better allocation of energy resources. Applications may use any communication direction or pattern but SecRose designates three distinct communication types, which are based on the flow of data. SecRose distinguishes and categorises the following communication types: (a) node to node, (b) node to base station and (c) base station to everyone. SecRose introduces one packet type for each communication pattern.

All packets are illustrated in Figure 2. SecRose retains and communicates the *type*, *group* and *data* fields as specified by TinyOS [31] in order to achieve full compatibility with the OS. The packet management mechanism of SecRose will transparently alter the length, source,

destination and MAC fields. Utilisation of the appropriate packet type, depending on the destination address, will be forced by SecRose for all communications. Higher layers of the communication stack need not be aware of SecRose's operation. All changes will be applied to outgoing packets at SecRose's level while changed incoming packets will be reverted before those are passed to higher layers.

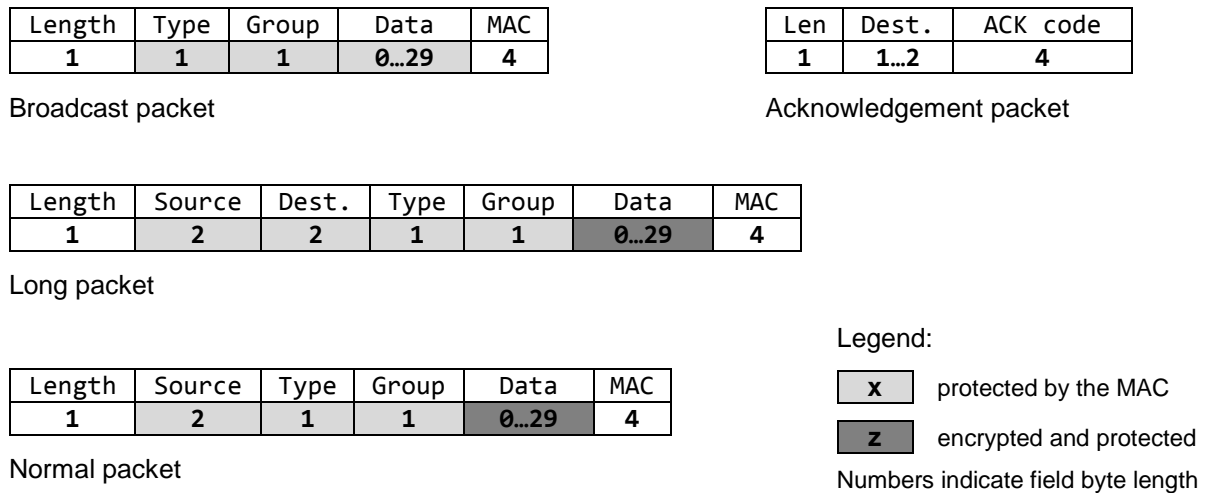


Figure 2: SecRose packets, their fields and their security features

Numbers express the field's size in bytes

SecRose defines a *flag*-based packet designation method, which allows for identification of the packet type. The 2-bit *flag* will be overloaded in the most significant bits of the packet's length, making it the first piece of information on a packet. This will allow nodes to make swift decisions while packets are being received. Nodes may quickly decide to accept or reject a packet, saving the radio reception energy. For accepted packets, the flag defines the expected packet fields and their length, allowing for proper reception regardless of packet type.

SecRose will use a packet of type *normal* for communication from any node to the base station. *Normal* packets have no destination address in the packet's fields. The address of the base station will be regarded as the default destination of all *normal* packets. This optimisation is effectively replacing the 16-byte *destination address* with the *flag*, which introduces zero radio overhead. SecRose chose to utilise this optimisation on *normal* packets on the assumption that the most frequent communication will originate from the nodes and be

destined to the base station. This assumption is based on the ultimate purpose of sensor networks; to deliver information to the base station [1].

For point-to-point communication between any pair of nodes in the network, SecRose will use a *long* packet. This type of packet retains all the original packet fields of TinyOS and does not provide any efficiency savings. *Long* packets can also be used by the base station to communicate with a specific node.

The third packet is the *broadcast* packet, which can only be used by the base station. It allows the base station to reach the whole network in an efficient and secure way. Nodes that receive *broadcast* packets are required to forward them once, maximising the potential reach of the packet. The *broadcast* packet does not include a *source* or a *destination* address; they can only originate from the base station and are destined to everyone.

Finally, SecRose defines a fourth packet, the *control* packet, which is a general-purpose packet for controlling SecRose's operation. Currently control packets are only needed to facilitate acknowledgements of reception and thus they will only be mentioned as *acknowledgement* packets in this document. However, the format of the packet is flexible. Therefore, this packet might facilitate other control commands in future versions of SecRose.

Acknowledgements will be sent by the receivers to the senders in cases where *normal* and *control* packets are received without any errors. *Broadcast* packets will not be associated with an acknowledgement.

Key management

SecRose defines a key management system aiming to minimise reuse of the same cryptographic key. The system represents the network as a set of communication pairs. Each communication pair has its own pair key, which is derived from a preloaded *initial key*.

The *initial key* is stored in the pairs during deployment and can be common for all the pairs. Even a broadcast communication is treated as a pair, with the base station on one side and the whole network on the other side of the pair.

Pair keys are derived using the *initial key* and a *counter value*. The *initial key* is 96 bits and the counter is 32 bits. SecRose advances the counter value after each communication, achieving creation of new pair keys. Counters will not be monotonically advanced. Instead, the exact update value will be determined by unused *meta-bytes* of the authentication component. The actual value that will be used to increase the counter is called the *counter update value*.

Key management will mix the *initial key* with the counter as shown in Figure 3. The process involves slicing the counter update value and then adding the bits to the blocks of the key. This functionality will slightly increase the diffusion of the counter's value in the key. The mixing process is only a complement to the cipher's diffusion properties.

It is not absolutely required to change keys after each communication. SecRose defines a failover mechanism, which allows for continuation of communication should packets or their acknowledgements fail to be delivered. SecRose will store two counters to facilitate failover: the *backup* counter and the *active* counter. It will also store the *counter update value*.

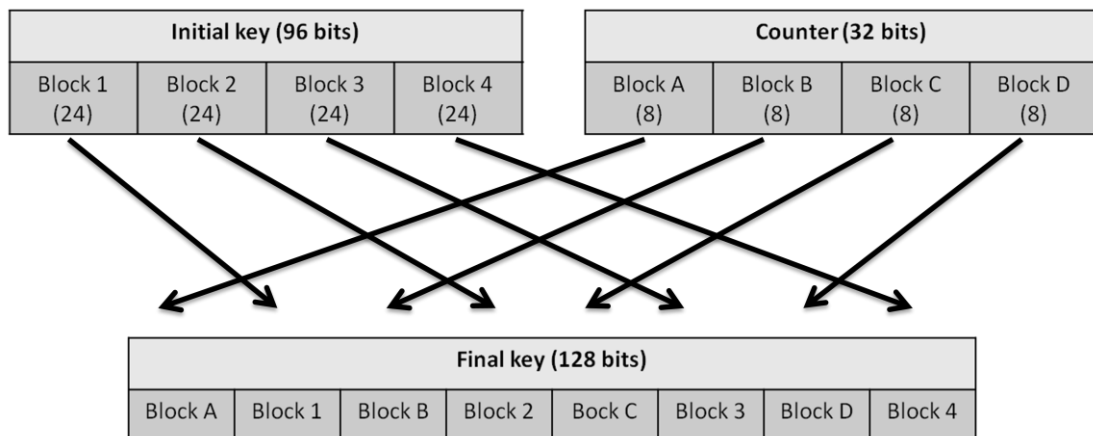


Figure 3: mixing of the *initial key* with the *counter*

Key management exception for *broadcast* packets

Broadcast packets are a special case that poses conflicting requirements. It is infeasible to facilitate the full key management features for *broadcast* packets, without increasing the

hardware capabilities of the base station. These packets will not trigger acknowledgements and will therefore not benefit from the failover mechanism.

The base station, which is the only node allowed to send *broadcast* packets, will faithfully advance its broadcast key after each broadcast communication to preserve freshness and the other properties of SecRose.

The lack of acknowledgement elevates the provided freshness and semantic security but introduces a potential attack on availability. This topic will be discussed in Chapter 6.

Authentication and integrity

SecRose defines the use of a *Message Authentication Code (MAC)* to provide authentication and integrity. The code will be transmitted with every packet.

The MAC will be calculated using the encryption function as specified by the CMAC [99] recommendation. The MAC will be calculated by utilising all packet fields and it will be appended to the end of the packet. According to the CMAC specification, the output value will depend on the key and the input data. As long as they remain constant, the CMAC method will always produce the same output.

The CMAC method outputs an 8-byte string, equal to the block size of the encryption function. The first four output bytes will become the MAC code and they will be transmitted with the packet. The rest will be utilised as *meta-bytes*, which is reproducible at both communication ends but is not transmitted with the packet.

The receiver will calculate the MAC code using the same input as the sender used. If the codes match then the packet is validated, the encryption key can be changed and the acknowledgement packet can be sent.

The fifth output byte, which is the first *meta-byte*, will be utilised by key management and will become the *counter update value*. The sixth and seventh byte will be sent back to the receiver, acting as an authenticated acknowledgement value.

The authenticated acknowledgement will guarantee to the sender that the receiver has updated their pair key. As a side effect, acknowledgements also guarantee that the packet has been received but the lack of an acknowledgement does not indicate the opposite.

Encryption

SecRose will use the XTEA [113] in block mode with 8 cipher cycles. Efficiency modifications in XTEA's code are allowed as long as the mathematical function remains the same. The number of cipher cycles might be increased to accommodate increased security needs. SecRose will use MAC stealing and selective encryption of the type and group packet fields to achieve efficiency gains. These techniques are similar to the ciphertext stealing technique proposed by [114].

4.1.2 SecRose operation: outgoing packets

Packets are passed to SecRose from the higher layers of the TinyOS. When this event happens, SecRose will execute the following steps. This subsection does not apply to acknowledgement packets. These will be discussed later in their own subsection.

Preparation

Each potential destination is associated with an appropriate packet type and consequently related to a flag. Therefore, both the packet type and the flag can be determined by the destination, as shown in the following resolution table:

If the destination is ...	The flag will be ...
The base station (usually 0)	0
The broadcast address (usually 65535)	1
Any other address	2

After determination, the flag's value will be overloaded to the two most significant bits of the packet's length.

Then the MAC code will be calculated; the packet will be passed to the authentication function and the 8-byte MAC code will be added to the packet's data.

Then the packet will be passed to the encryption function to encrypt it. Encryption might include any of the following efficiency features: padding, MAC stealing or inclusion of the group and type. The actual case depends on the packet's data size.

Packet transmission

After preparation, the packet is ready to be transmitted. SecRose transmits the packet fields described in the packet categorisation definition and illustrated in Figure 2. For reasons that will be explained later, only 4 of the 8 MAC bytes are transmitted.

Post tasks

During transmission, a copy of the packet is transferred to the receiver but the original packet's data remain in the sender's memory as well. These data will be passed to key management, which will undertake the following steps.

For *normal* and *long* packets, the value of byte 5 will become the *counter update value* but it will not be utilised yet, it will only be saved to a temporary memory location. Bytes number 6 and 7 will also be saved for future use. These form the authenticated acknowledgement value. No keys will be changed unless a valid acknowledgement is received.

All values will be stored in a memory location identifiable by the receiver's node ID. Should the receiver be the broadcast address then a special ID will be used.

Broadcast packets will not utilise acknowledgements. Key management will immediately increase the active counter by adding the 5th MAC byte to it.

At this stage, the sender can continue with other tasks while it waits for the acknowledgement packet to be received.

4.1.3 Operation: incoming packets

All data transmitted over the radio are passed, byte-by-byte, to SecRose from the lower layers of the TinyOS. SecRose will detect when a packet preamble is received, and will begin the packet reception sequence as specified here.

Type determination and appropriate reception

The first byte is the length of the packet and it includes the overloaded flag. The first two bits of the length are the flag. SecRose will examine these bits to determine the packet's type. It will also remove these bits from the length, allowing it to take its normal value, representing the actual data length.

Packet rejection or forwarding might happen at this stage. Assuming that neither of these happens, the receiver will use the flag, which characterises the packet type, to determine which packet fields it should expect. For *normal* packets, the destination will be set to the base station's ID. For *broadcast* packets, the destination will be set to the broadcast address and the source will be set to the base station ID.

Then the receiver will continue to receive the bytes of the packet in accordance to what should be expected by this packet type. Upon reception of the MAC, packet reception has concluded.

Decryption and validation

In the next stage, SecRose will pass the packet to the encryption function in order to decrypt it. The process will take any efficiency features that were selected during encryption into account.

When decryption concludes, an attempt to validate the MAC commences. The packet will be passed to the authentication component and the actual MAC of the received data will be calculated and checked against the received MAC.

If MAC validation succeeds;

1. the packet is accepted and passed to the higher layers of the operating system
2. key management will be informed to update the counter value
3. an acknowledgement will be sent, unless the packet was a *broadcast* packet

If validation fails, the failover mechanism is invoked. SecRose will ask key management to revert to the *backup key*. Then the decryption and MAC validation will be repeated. If the secondary validation succeeds then the packet will be accepted but an acknowledgement packet will not be sent. Should secondary validation fail as well, then SecRose cannot process this packet and it has to be rejected.

4.1.4 Operation: authenticated acknowledgement transmission and reception

SecRose defines a way to exchange authenticated acknowledgements. The method is computationally efficient but consumes significant radio energy. The additional energy cost is introduced by the transmission of the preamble of the acknowledgement packet.

Acknowledgements are sent when the transmission and reception process is carried out smoothly and without any problems. Acknowledgements are not sent if MAC validation fails.

Concept

During packet transmission, a copy of the 6th and 7th MAC bytes was kept by the node in its memory. In the receiver, the same bytes are also calculated during the packet validation process when the MAC of the received packet was calculated as well. Therefore, both nodes have access to these bytes.

These bytes are the acknowledgement value. The receiver transmits them back to the sender inside an acknowledgement packet. The sender receives them and matches them against its previously stored data.

When the received acknowledgement value matches the expected acknowledgement value, the sender is confident that the receiver has successfully received its message and has changed its key accordingly.

Exchange of acknowledgements

Acknowledgement packets will be exchanged using the normal packet transmission process of any packet. Acknowledgements are initiated within SecRose and its flag is pre-set to value '3'.

This flag will be detected by packet management and by transmission/reception control. At the preparation stage, the normal flag and type identification will be bypassed. In addition, instead of a MAC calculation, the packet will be populated with the authentication data. The transmission/reception stage will also be adapted to exchange them. The real differences are the validation method and the post-tasks.

Post tasks

Acknowledgement validation will happen as described in the concept. The process will be handled by the key management mechanism, which has stored the acknowledgement authentication data. The authentication component will not be invoked.

Upon detection of a valid acknowledgement, the key management will add the previously stored counter update value to the active counter. Before doing so, a backup copy of the *active* counter must be kept, in case the failover mechanism needs to be invoked.

At this stage, the pair has exchanged a packet and has successfully advanced the cryptographic pair key that it uses.

Acknowledgement table size and security limits

The number of values that the acknowledgement table can hold is configurable by the developer. The actual value should be chosen after accounting for the expected packet transmission rates. SecRose recommends a default of 10, which should be suitable for all applications. The value expresses how many packets will be in the waiting stage for their acknowledgement to be received.

SecRose will keep track of how many packets are awaiting acknowledgement and how many invalid acknowledgements have been received. If a valid acknowledgement is received then the invalid counter is reset. If the number of invalid acknowledgements equals the number of

awaiting acknowledgements then the whole acknowledgement table is wiped and counting resets.

4.1.5 Operation: intermediate nodes

Packets are often exchanged over a number of intermediate nodes (hops). SecRose provides facilities for these nodes to determine their role as receivers, forwarders or terminators of a packet. This functionality needs collaboration with an appropriate routing protocol.

Path position discovery

The destination of a SecRose packet is always discoverable. *Long* packets have it written on the appropriate packet field, which is never encrypted. *Normal* and *broadcast* packets contain an appropriate flag, which can be analysed to determine the packet's destination.

After determining the destination, nodes can consult their routing table to determine their position within the route of a packet.

Forwarding

Nodes that form parts of a route should forward every packet without attempting to decrypt or validate it. Since sensor networks are half-duplex, the nodes have to receive the whole packet and then send it again.

Under normal circumstances, all nodes must forward all *broadcast* packets. Some exceptions might stand for specific routing protocols or applications. The exact behaviour is neither defined nor affected by SecRose and it is therefore out of the scope of the SecRose proposal.

Rejection

Nodes that are not in the route of a packet might reject it. Packets can be rejected as early as after reception of the flag. The flag allows destination determination for *normal* packets. For *long* packets, the nodes have to wait until reception of the destination field has concluded.

4.1.6 Operation: diagrams

Pair key advancement and state preservation sequence

Figure 4 illustrates how key management maintains the state of the pair key in accordance with events that might happen during communication.

The first packet is transmitted and received without any errors. Both ends advance their keys to K_2 .

The next packet is secured using K_2 but the packet arrives altered to the receiver. Therefore, MAC validation fails and the receiver does not pass the packet to higher layers. K_2 continues to be the active key.

In the third packet, the sender uses K_2 again to send SP_3 and this time the packet arrives correctly. The receiver successfully changes the key to K_3 but the acknowledgement delivery is erroneous, preventing partial recovery. Consequently, the sender does not change to K_3 .

In the fourth packet shows partial recovery: SP_4 arrives correctly and the receiver reverts from K_3 back to K_2 . However, this is a case where acknowledgement should not be transmitted. Therefore, there is no change in the sender.

The final packet demonstrates full recovery. The sender is obliged to reuse K_2 for a fourth time to send SP_5 , which arrives without problems. The acknowledgement is also successful and thus both nodes advance to K_3 .

Despite two consequent communication errors, SecRose manages to maintain pair key synchronisation by reusing K_2 four times.

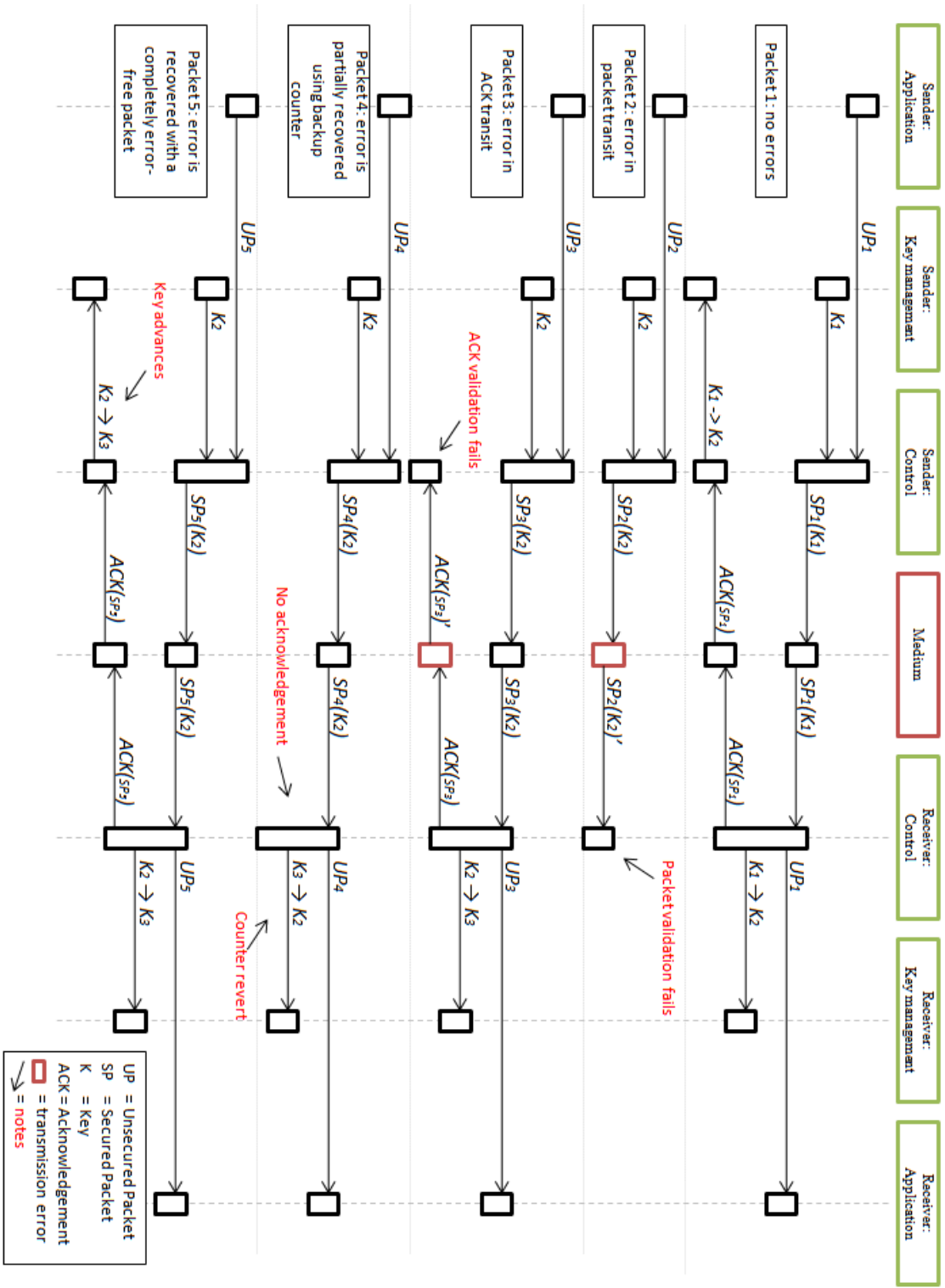


Figure 4: pair key advancement and state preservation sequence

4.2 System design

This section provides the systematic description of the SecRose mechanism. The components and subcomponents of the system are described. Initially, an overall system description is given and then detailed information for each component is discussed. Some of the components are designed by a method selected among various options. For these components, the evaluation and selection process is discussed.

Notes

This is the design of the system, not the design of the proof-of-concept implementation, which will be discussed in Chapter 5. Components are shown clearly here but, due to the structure of TinyOS, it is impossible or inefficient to retain the design clarity in actual code.

This section makes extensive use of coloured diagrams. The colours in the diagrams provide important information. All the diagrams include legends in which the meaning of colours is explained.

4.2.1 System overview

SecRose consists of four fundamental components: encryption, key management, authentication and the control component. All components interface with each other at various operation points and the control component interfaces with the rest of the TinyOS system. This subsection gives a systematic description of the components, the interactions between them and with TinyOS. A high-level illustration of the system is provided in Figure 5.

Control component

The control component is responsible for (a) interfacing of the SecRose mechanism with other layers, (b) coordinating the other SecRose components to achieve secure packet communication. This is the key component of the system. The control component hosts a number of subcomponents, including the packet management subcomponent, which is responsible for packet categorisation, reception and transmission.

Encryption component

The encryption component utilises a cryptographic algorithm to provide encryption services to any component that needs it. The component accepts data bytes and a key as input. After execution, the function returns the encrypted data. The component also facilitates efficiency subcomponents.

Key management component

The key management component, or mechanism, is primarily responsible for maintaining, altering and providing cryptographic keys. The goal of the component is to alter the key used to send a packet after the event has completed. The mechanism keeps a state of the key for each communication pair and provides the control component with functionality to manage this state.

Authentication component

The authentication component is responsible for providing and validating a Message Authentication Code (MAC). The code can be used to both authenticate data and validate their integrity.

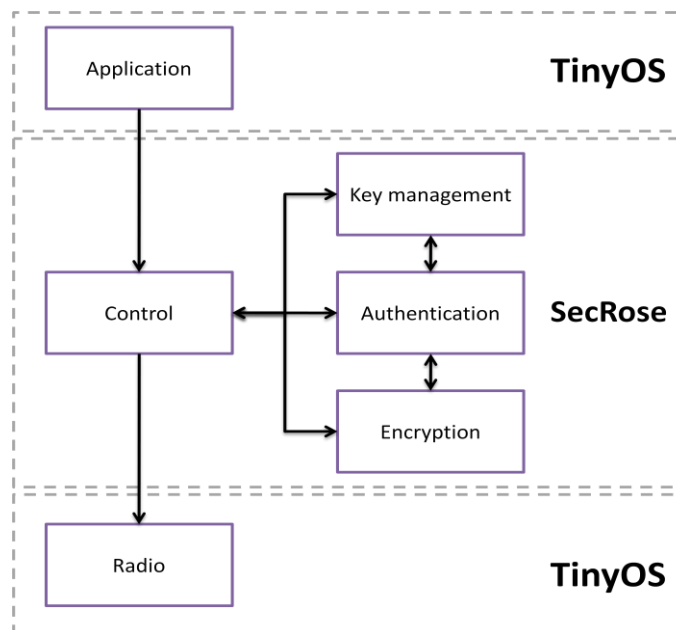


Figure 5: position of SecRose in TinyOS and the possible internal interactions of fundamental SecRose components.

4.2.2 Encryption component

Interfaces and their operation

The encryption component accepts four requests:

1. Encrypt packet with appropriate stealing
2. Decrypt packet while accounting for stealing
3. Encrypt stream of bytes
4. Decrypt stream of bytes

The control component uses the first two requests while the authentication component uses the raw encryption of data streams. In the cases where whole packets are handled, the component requests appropriate keys from the key management component. The component provides a MAC and ciphertext stealing method to reduce radio overhead for small payloads.

The primary subcomponent is the encryption function itself, which for reasons discussed below, is the XXTEA[113] encryption algorithm.

Cipher selection criteria

A number of selection criteria have been examined in order to select an appropriate cipher for use with SecRose. Three basic criteria were defined; resource utilisation, cryptanalytic reputation, non-proprietary and compatible with the GNU licence. In addition, the cipher had to support the basic requirement of at least 128-bit key lengths and had to be a stream cipher. The rationale behind these criteria is discussed below.

Resource utilisation was the most important criterion. The final requirement is determined as a composite of the computational difficulty of cryptographic operations, the memory needed to function and the minimum block size. The block size is important for the small sensor network packets as it saves radio overhead. The advantages are greater if ciphertext stealing is in place.

Secondly, the selected encryption function had to be reputable in the cryptographic community as resilient to cryptanalysis and as a cipher with strong cryptographic properties. This criterion automatically disqualifies new ciphers, as they are considered immature.

Finally, the cipher has to be free of any patents or other restrictions that would either introduce additional costs to the deployment of SecRose or be incompatible with GNU General Public Licence (GPL) [115]. This requirement is forced by legal restrictions surrounding the licence. The GPL requires software updates of GPL products to be released under the GPL licence as well. Since SecRose can legally be regarded as an update for TinyOS, which is GNU-licensed, it has to be GPL as well. Any cipher with licence restrictions that disallow that is automatically disqualified.

The additional criteria are set by the requirements and the technical limitations of WSNs. The cipher had to support at least 128-bit keys, a basic requirement, but longer keys are not prohibited.

Stream ciphers were not considered due to a number of reasons. Firstly, the packet-based information is essentially information in blocks and thus block ciphers are more suitable for sensor networks. Secondly, block ciphers are a better fit for SecRose itself as they can be used for both encryption and authentication. Thirdly, stream ciphers require use of an initialisation vector to operate securely, which would ultimately affect their radio energy demands. Finally, block ciphers tend to be more mature and utilised than other types of ciphers.

Candidates and initial evaluation

The initial criterion for considering a cipher was to support 128-bits key length. There are a number of ciphers that satisfy this; RC4 [116], RC5 [117], RC6 [118], Twofish [119], Blowfish [120], Triple-DES [121], Rijndael (AES) [122], TEA [123], XTEA and XXTEA [113]. However, most of are either worse than AES or suffer by some other problem.

The Blowfish cipher was replaced by Twofish, RC4 and RC5 were succeeded by RC6, and TEA was corrected by XTEA/XXTEA. All these replacements are attempts to address either suggested or proven problems with the ciphers. Unfortunately, there is no replacement for the problematic Triple-DES which is considered to have an effective key length of 80-bits [71].

At this stage, the remaining ciphers are Twofish, RC6, Rijndael and XTEA/XXTEA. The first three were contenders in the AES competition, which concluded that the best cipher is Rijndael. Therefore, Rijndael, which then is known as AES, is the most reputable and a better cipher than the other AES competitors are, and there is no reason to dispute that.

After the preliminary evaluation and given the problems described, the remaining algorithms are XTEA/XXTEA and AES. These were examined for their performance and resource utilisation in the final evaluation.

Final evaluation and selection

There is no published direct comparison of the computational requirements of the final candidates. However, a relative but inconclusive comparison between TEA variants and AES can be derived from references [96, 124-128]. With the exception of AES, all TEA variants outperform other ciphers. On the other hand, AES requires less computational resources while XXTEA has a smaller memory footprint [96, 126].

However, reference [126] does not account for ciphertext stealing or the increased cycles that XXTEA uses when data input is small or research showing that 8 cycles are enough [129]. Eight XXTEA cycles will definitely outperform AES by a factor of at least two, without considering ciphertext stealing and block-size radio overhead.

For these reasons, the XXTEA encryption algorithm will be selected for SecRose and AES should be considered the next viable alternative should XXTEA be proven insecure.

SecRose will use XXTEA without the gradual reduction of cycles. The cycles of the cipher will be permanently set to eight, which is proposed by the current maintainer of TEA and is claimed to be secure for most applications [129]. Additional cycles might be added by sensor network developers who demand higher security margins.

Alternative ciphers

Although XXTEA was selected and highly recommended for SecRose, the secure design of the system is not limited to this particular encryption function. Use of any block cipher would not affect the security of the SecRose protocol but some changes to the system will be required.

The modular design of the system components allows for relatively easy transition to other ciphers. Updates on the code will be required on the stealing subcomponent, since it is optimised for the characteristics of XXTEA.

However, any changes will require appropriate efficiency evaluation, since the various ciphers utilise resources differently.

Component interactions

The encryption component provides four interfaces, which might be utilised by other components. Possible interactions are illustrated in Figure 6.

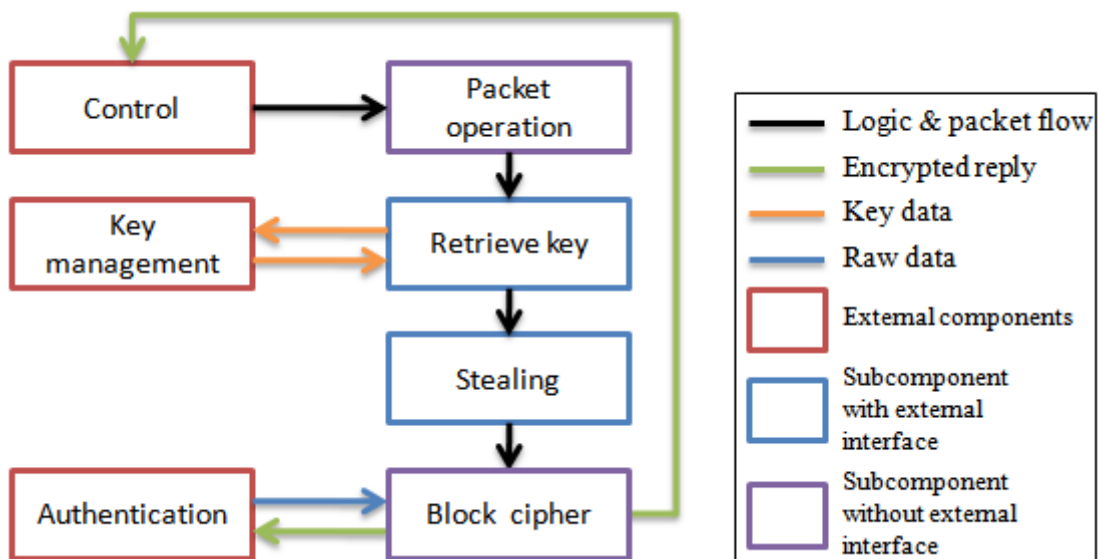


Figure 6: the subcomponents of the encryption component and their interfaces

Stealing subcomponent

Block ciphers require data input in blocks of a specified size, which is usually a multiple of 8. In cases where the actual data are not an exact multiple of eight, the data have to either be padded with a known value or ciphertext stealing has to be applied [4, 122].

SecRose uses a stealing subcomponent to achieve maximum efficiency. Although the method is inspired by the ciphertext stealing, it operates pre-emptively, before any encryption happens, and it primarily relies in the 4-byte MAC and the 4-Byte block size of XXTEA. The method operates more efficiently than traditional stealing solutions, as it does not require additional iterations of the encryption cipher.

SecRose steals selectively from the MAC field, the *type* field or the *group* field of the packet. The stealing subcomponent allows SecRose to refrain from using any padded data for packets carrying more than 1-byte data payload.

In addition to efficiency gains, the stealing subcomponent complements the computational complexity of transmitted encrypted and authenticated packets, effectively increasing the computational effort required to conduct a brute force attack. However, the exact cryptographic gains vary in accordance to the packet's payload and their true impact was not precisely analysed. Nevertheless, it remains a complementary feature that is achieved with minimal impact to energy requirements.

Operation of the stealing subcomponent

Generic stealing subcomponents follow a simple strategy; encrypt the “first” blocks and then encrypt the “last” block. Initially, the subcomponent encrypts as many full blocks as possible, given an arbitrary input size. This leaves few bytes that are not enough to form a full block unencrypted. These bytes are complemented with some part of the already encrypted bytes, as many as required to form a full block. Then encrypt this “last” block. The process is illustrated in Figure 7.

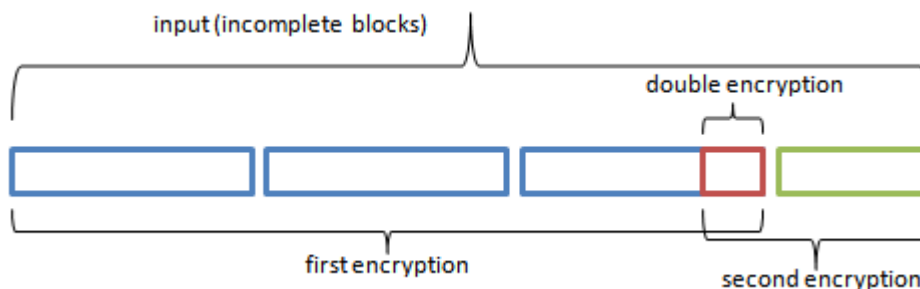


Figure 7: the principle of ciphertext stealing

SecRose improves this technique by treating the whole packet's data as an array of possible inputs to the encryption function while only the encryption of the *data* field is required. SecRose may include bytes from other packet fields in order to complete the last block. The selection happens before the encryption and the cipher is only called once, after complete blocks have been formed. Since there is no double encryption, the method consumes 50% less

computational resources than a typical ciphertext stealing subcomponent. The field selection process consumes insignificant resources.

The design of the SecRose stealing subcomponent is illustrated in Figure 8. The whole packet is the input. The process begins by stealing the MAC; if the last block of the *data* field is not a complete block, SecRose will add as many parts of the MAC as required to complete it. An example of the effect on a *long* packet with two bytes data payload is given in Figure 9.

Since both the block size of TEA and the MAC are 4 bytes, the MAC stealing method would be sufficient to complete any block. However, the minimum block size of TEA is 8 bytes. Therefore, packets with less than 4 bytes payload cannot be securely completed with MAC stealing only. In these cases, SecRose will selectively add as many bytes from the other fields of the packet as needed. In the extreme case when a packet's payload is 1 byte, SecRose will also add a padding byte, increasing the ciphertext. Table 2 illustrates the various scenarios for the first 13 bytes of data payload.

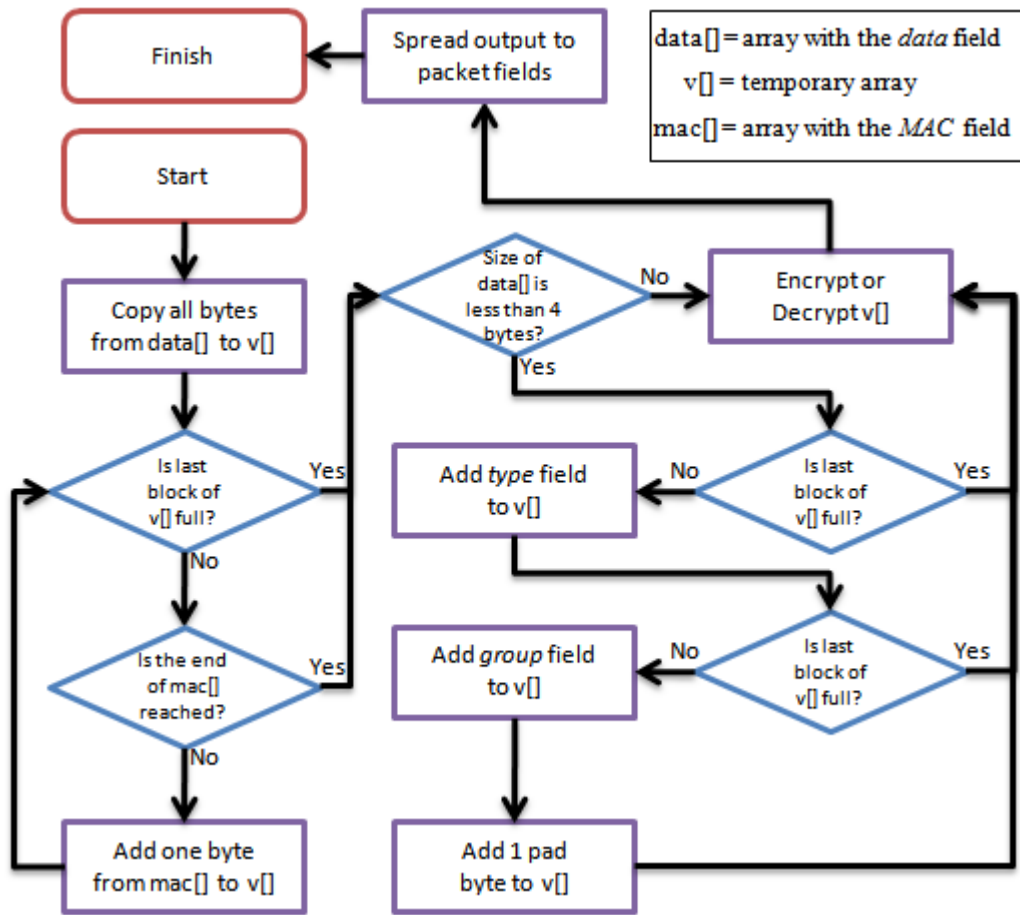


Figure 8: the stealing subcomponent of SecRose

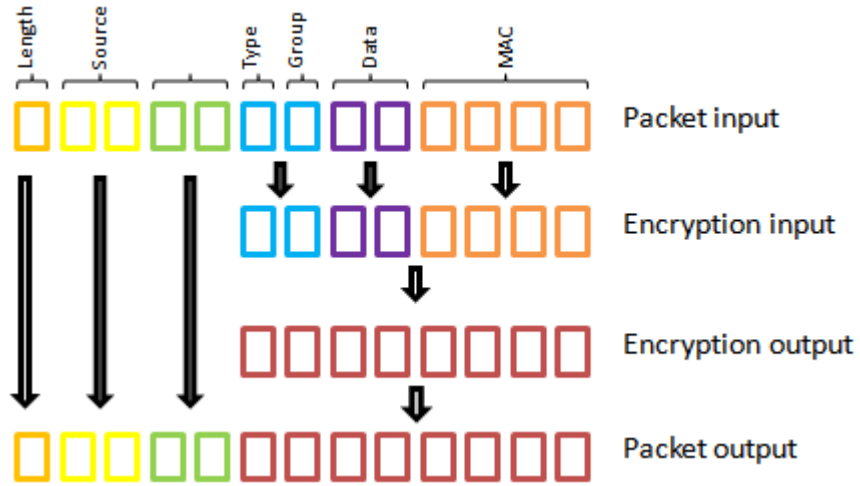


Figure 9: example of stealing applied to a *long* packet with two bytes data payload

Payload	Bytes stolen from				Total input to the encryption
	MAC	Type	Group	Pad	
1	4	1	1	1	8
2	4	1	1	0	8
3	4	1	0	0	8
4	4	0	0	0	8
5	3	0	0	0	8
6	2	0	0	0	8
7	1	0	0	0	8
8	0	0	0	0	8
9	3	0	0	0	12
10	2	0	0	0	12
11	1	0	0	0	12
12	0	0	0	0	12
13	3	0	0	0	12

Table 2: packet field utilisation by the stealing subcomponent.

All numbers express Bytes.

4.2.3 Key management component

Interfaces

The component can accept requests for a key to be provided or be informed about packet events by the control component. Both operations require a packet to be inputted. Although the key services interface does not directly interact with the authentication component, it does rely on information that has been calculated by it.

The control component uses an interface to inform key management when a packet event has happened. The possible packet events are:

1. A packet has been sent
2. A packet with a valid MAC has been received
3. A packet with an invalid MAC has been received
4. An acknowledgement packet has been received

Key management keeps an *active counter* and a *backup counter*. An *active counter* can be utilised to create a *final key*, which is sometimes referred to as the *active key*. If the *backup counter* is utilised the key is called the *backup key*.

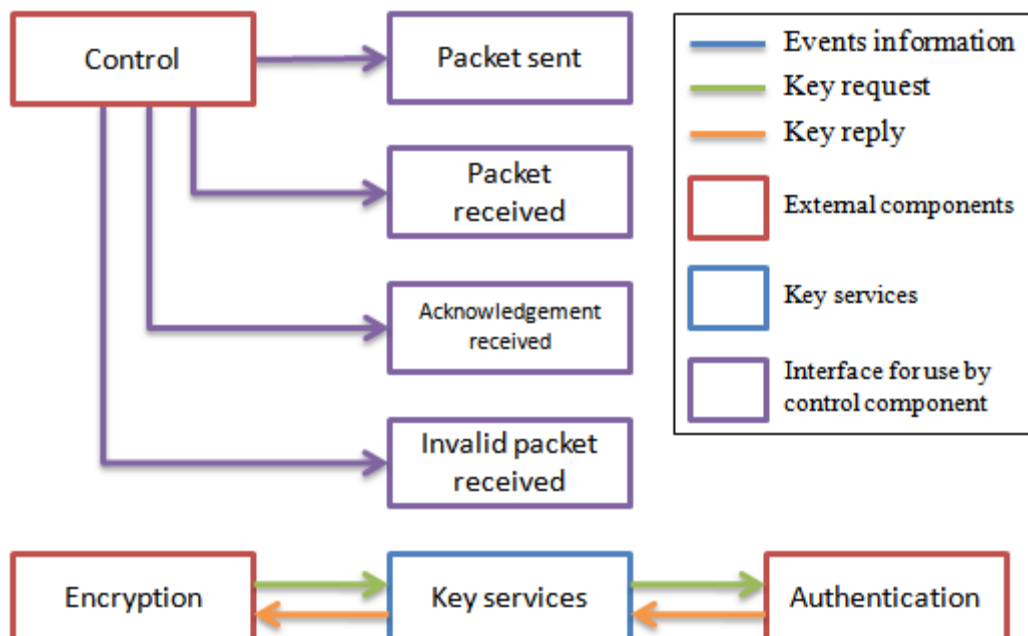


Figure 10: key management: interfaces, interactions and subcomponents

State information storage

The component keeps two tables; the *counter table*, which stores pair counters and the *acknowledgement table*, which stores acknowledgement values. The authentication component calculates update values and content for both tables and provides it to key management as part of the packet's data. The key management applies the information to the tables.

Keys are composite; they consist of a 96-bit *initial key* and a 32-bit counter. The *counter table* stores two counter values, *active* and *backup*, and the *counter update value*. These values are mixed with the *initial key*, which is uniform and preloaded onto all nodes. This way a unique, 128-bit *final key* is created.

The pair ID is stored, on both the acknowledgement and the *counter table*. On the *counter table*, the pair ID acts as the index of the table and allows the component to find and manipulate the values of the counters. On the acknowledgement table, the index is the authentication value and it is used to determine the pair ID associated with this an acknowledgement packet.

Key management requires 13 bytes of storage for every communication pair present in the *counter table*, in order to store *active* and *backup* counters. In addition, 32 bits of temporary storage for each communication are required until the acknowledgement is received.

Operation

Providing keys involves determination of the packet type and its destination. Based on the destination information the component consults its *counter table* to retrieve the *active counter*, mix it with the *initial key* and provide the *final key* for this particular destination.

When a packet is sent, the *counter update value* is stored in the *counter table*. When the acknowledgement is received and validated, the *counter update value* will be retrieved from the *counter table* and added to the *active counter*. Backups of the old values are also kept. *Broadcast* packets are exceptions to this process as they are not associated with acknowledgements. In this case, the key management component will immediately add the *counter update value* to the *active counter*.

When a validated packet is received, key management makes a permanent update on the *counter table*, noting the new *active counter* and backing up the existing *active counter*. The process will be repeated when the next valid packet is received and the *backup counter* will be completely overwritten then.

However, if the next received packet does not contain a valid MAC, the system can revert to the *backup counter* to allow the control packet to try MAC validation again. If this fails then communication in this pair is deemed broken, as the other end can send packets but it cannot receive the acknowledgements.

Finally, the acknowledgement table limits the maximum invalid acknowledgements it can receive to be no more than the amount of acknowledgements it awaits. If the maximum value is reached, the acknowledgement table is wiped. This limit is imposed for security reasons but it does not affect the normal operation of SecRose.

Note that for each communication pair, there is only one memory slot for the *backup counter*. Each time a valid acknowledgement or a valid packet is received, this slot is overwritten.

Evolving communication pairs

The key management system creates *evolving communication pairs* between the nodes of the network. The pairs are called “evolving” because they are constantly changing after each communication. This functionality allows the key management component to achieve powerful security characteristics like semantic security, weak freshness, authentication and integrity. In addition, it reduces the available cryptanalytic options and discourages node-capturing attacks¹¹.

The authentication component relies on the evolving keys to guarantee the identity of sending nodes to their receivers.

The subcomponent provides failover in case packets or their acknowledgements cannot be received by one end of the pair. The mechanism reverts from the currently *active counter* to the currently *backup counter*. Inevitably, this leads to the possibility that the same key might be reused.

¹¹ The security characteristics of the key management component will be discussed in Chapter 6.

Ideally, a system should never reuse keys in any case, under any condition but this requirement cannot be efficiently accommodated in a sensor network where repetitions of packets or acknowledgements are energy demanding.

Note that there are no actual *backup keys* stored in the system. The system holds backup values of the counters. The *backup key* is a *final key* that is created after the old counter has been restored.

Method rationale

Key management systems aim to agree a symmetric communication key between two communication points. Other solutions include public key infrastructure (PKI) systems like SSL [130].

The PKI solutions are much more effective than the described key management. They are able to provide randomised cryptographic keys, instead of the mathematically related keys that SecRose provides. In addition, the current key management is based on a pre-deployed shared key, which can become a security liability. However, sensor networks cannot support PKI solutions, as they demand too much communication, computation and storage resources.

The proposed method saves the few bytes introduced by other solutions for semantic security but it potentially consumes many more bytes in the preamble of the acknowledgement packets. This overhead is accepted as a side effect for the provision of the methods security properties.

Subcomponent diagrams

Figure 11 illustrates the process of retrieving the counter and mixing with the *initial key*. The process starts by an external component, shown in red, which is requesting a key. The whole packet is passed to the component as input. The component will then determine the destination by reading the destination address or the flag and consequently determine the pair's ID. Then the *active counter* value will be retrieved from the *counter table* and mixed with the *initial key*, which resides in memory. Then key management can deliver the *final key* to the external component that requested it.

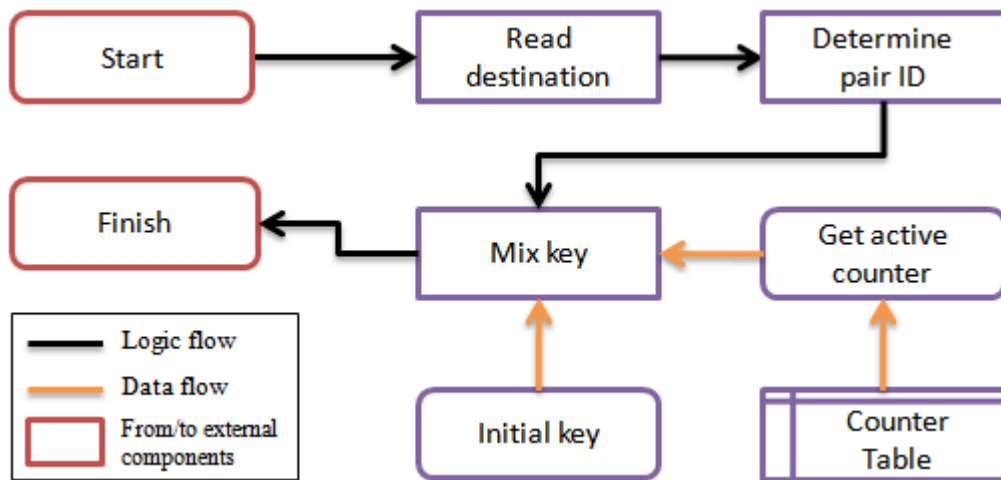


Figure 11: key request process

Figure 12 illustrates the post-transmission tasks for the key management component. The control component informs key management that a packet has been sent and it passes the packet to key management. The packet's data contains the flag, the destination address, and the full 8-byte MAC. Key management determines the packet type and acts accordingly. If the packet was as *broadcast* packet, it will advance the counter immediately after saving a backup of the *active counter* value. For all other packets, it will only store a temporary value and the acknowledgement value. Both values are included in the packet's data, in the 8-byte output of the authentication component. When the acknowledgement is received, the temporarily stored *counter advance value* will be used.

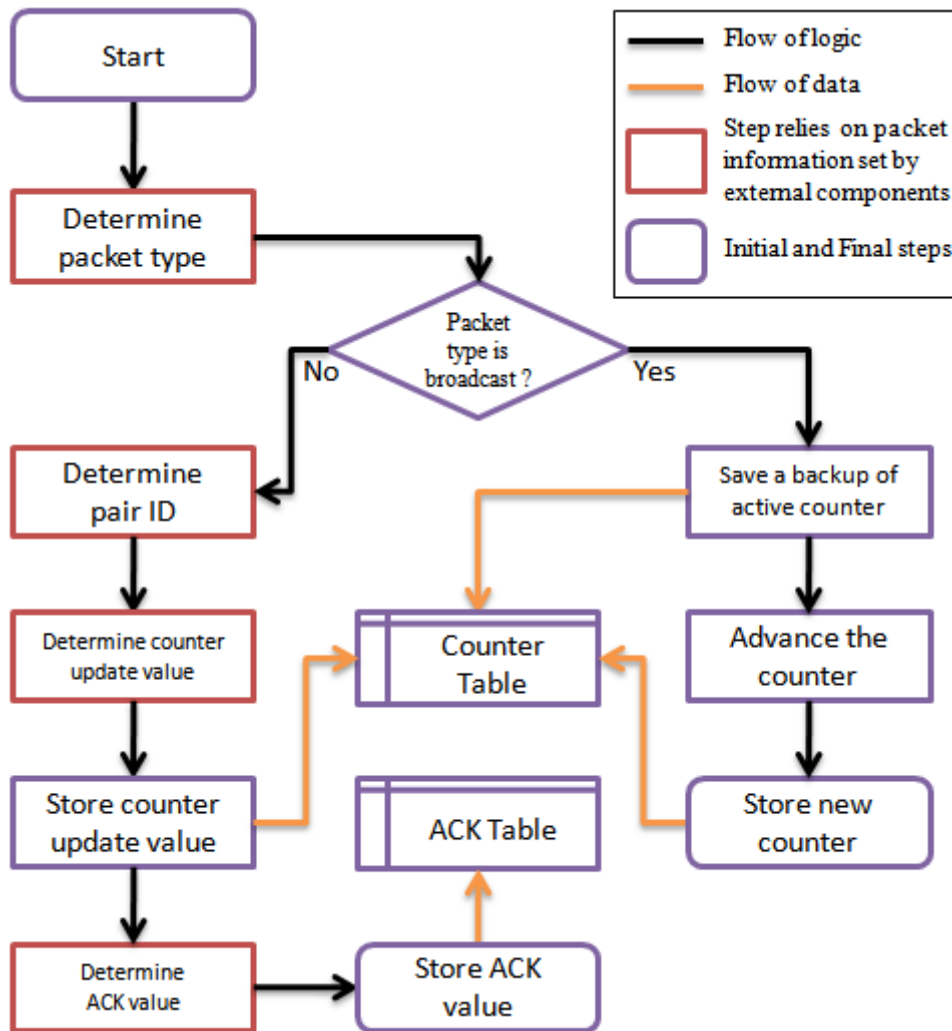


Figure 12: key management tasks after packet transmission

Figure 13 illustrates the process on the key management component after the reception of a valid acknowledgement. The control component informs the key management that an acknowledgement has been received and it passes the packet on. The packet contains two bytes of acknowledgement value, which is used as the index to query the acknowledgement table. When a match is found, the component can retrieve the pair ID, since the acknowledgement value acts as the index of the table. The component then retrieves the *active counter* value and the *counter advance value*, which is used to advance the counter. The next packet for this pair will use the new counter.

In case the received acknowledgement value is not found in the acknowledgement table, the component does not do anything. Note that if a value is found then the component will also remove it from the acknowledgement table. This activity is not shown in the diagram.

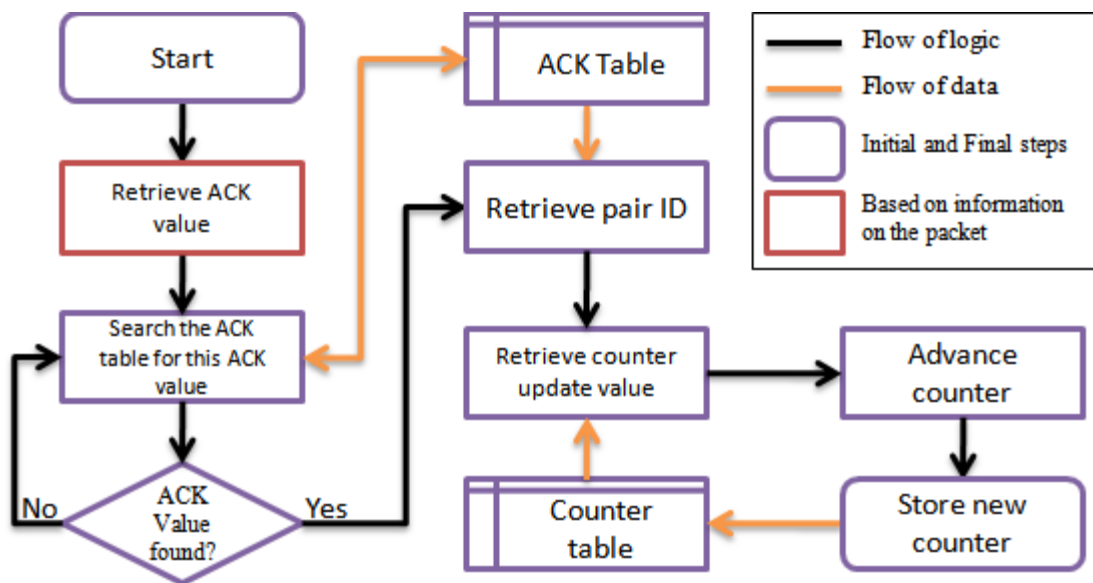


Figure 13: process after acknowledgement reception

Figure 14 illustrates the key management activities after a valid packet has been received. The process is initiated by the control component and the whole packet is passed to key management. The packet's data contain the destination and the full 8-byte MAC as calculated by the authentication component. Key management examines the destination to determine the pair's ID. After that is found, it will initially create a backup of the *active counter*. Then it will read the 5th MAC byte to determine the *counter advance value* and use it to advance the *active*

counter. The new *active counter* is finally saved on the *counter table*. The next packet for this pair will use the new counter.

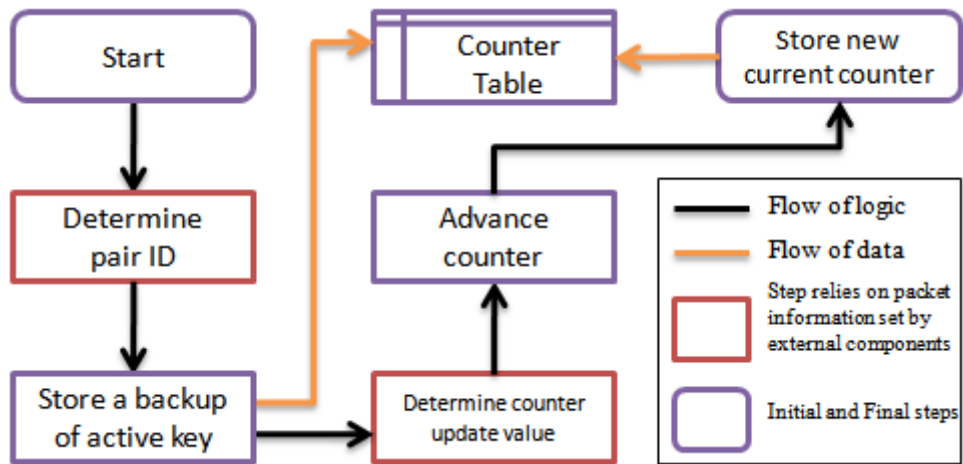


Figure 14: process after packet reception

Figure 15 illustrates how key management reverts to the *backup counter* after request of the control component. The control component should attempt packet validation using the *backup key* if it received a packet that cannot be validated normally. In order to do so it asks the key management to revert its currently *active counter* to the saved backup. The effect of the process is permanent and affects all future packets. The simple diagram explains this process.

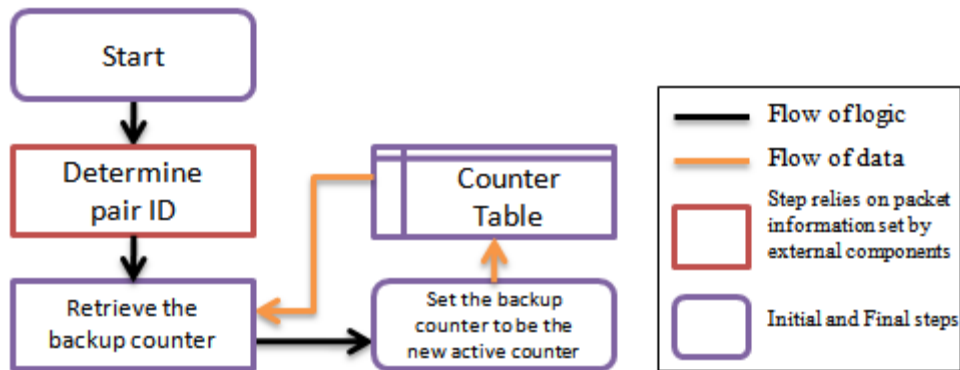


Figure 15: revert to *backup counter*

4.2.4 Authentication and integrity

Interfaces and their operation

This component interfaces with all the other components in various ways. Its primary functionality is to provide Message Authentication Codes (MAC) of outgoing packets and to validate the MACs of incoming packets. In order to achieve this functionality the component requests keys from the key management component and encryption services from the encryption component. In addition, the authentication component provides key update and acknowledgement values to key management after processing a packet. The component uses the CMAC [99] specification to calculate MACs.

The component relies on the key management component to provide authentication and trust. In particular, the authentication component guarantees that the sending node is a node that possesses the pair key while the evolving communication pairs guarantee that other nodes do not use the same keys.

In order to calculate a MAC the component requires provision of the related packet by the control component. Then it requests the *active* key from key management in order to request encryption services correctly. The MAC is then calculated and a string of eight signature bytes is produced. This stream is unique to distinctive input.

The first four of these bytes will act as the MAC of the packet. The fifth byte is the *counter increase value* and the sixth and seventh bytes will be used for the acknowledgement value. All the information is appended to the packet for utilisation of the control and key management components. The 8th byte is currently not used by SecRose.

A packet that has been altered in transit results in a different input to the MAC component of the receiving node. Since the resulting string of bytes is unique to each distinctive input, altered bytes will cause MAC validation to fail. Therefore, the authentication component guarantees packet integrity as a side effect.

The CMAC process

The CMAC [99] specification generates a message authentication code by invoking the encryption function to (a) derive three keys and (b) reduce the input text into the size of a cipher block by sequentially encrypting it. The command that facilitates this process is sometimes called the *MAC function*. The CMAC specification guarantees the creation of a MAC that is as secure as the encryption function used.

Collision probability and confidence

MACs are associated with a collision probability, or otherwise they provide guarantees with a confidence level. A 32-byte MAC facilitates 2^{32} distinctive outputs. The MAC guarantees that a received packet is authentic and intact with a probability of error equal to 1 in 2^{32} .

Therefore, a collision will occur, on average, after 2^{31} calculations that involved equal unique inputs. An attacker might attempt to send 2^{31} packets with a random MAC in order to forge authentication or integrity. The smallest possible packet is a *broadcast* packet with 0 data payload, which equals to a total of 7 bytes, without counting the preamble. A node powered by a typical AA battery will cease to operate before 2^{16} such packets are received [69]. The 4-byte MAC provides confidence that greatly exceeds the capabilities of the sensor nodes.

Selection rationale

There are two ways to provide digital signatures: use a cryptographic hash function or use a block cipher to generate them. The later method is selected primarily because it allows for code-reuse and subsequently provides better code size optimisation. Using a block cipher, there are only three options to generate MACs; CBC-MAC [131], OCB [100] and CMAC [99].

CBC-MAC is secure only for fixed-length messages and is therefore not recommended. The OCB mode of operation for block ciphers could be a promising solution but the security of this method is unclear.

CMAC is the secure successor of CBC-MAC and it is recommended by NIST [132] for all applications. Therefore, the CMAC mode is selected for the current version of SecRose.

Further evaluation and possibly redesign of authentication and key management is required to accommodate OCB in SecRose. Future work might evaluate and possibly use OCB in SecRose. This topic will be further discussed in Chapter 7.

Component interface diagram

Figure 16 illustrates the steps required to calculate a packet's MAC. The packet is the input. The process is initiated by the control component while it requires interaction with every other component. The general method is to derive three keys and feeding of the packet's data to the CMAC calculation function for sequential encryptions. MAC validation is illustrated in Figure 17 and it is simply a MAC generation and a comparison.

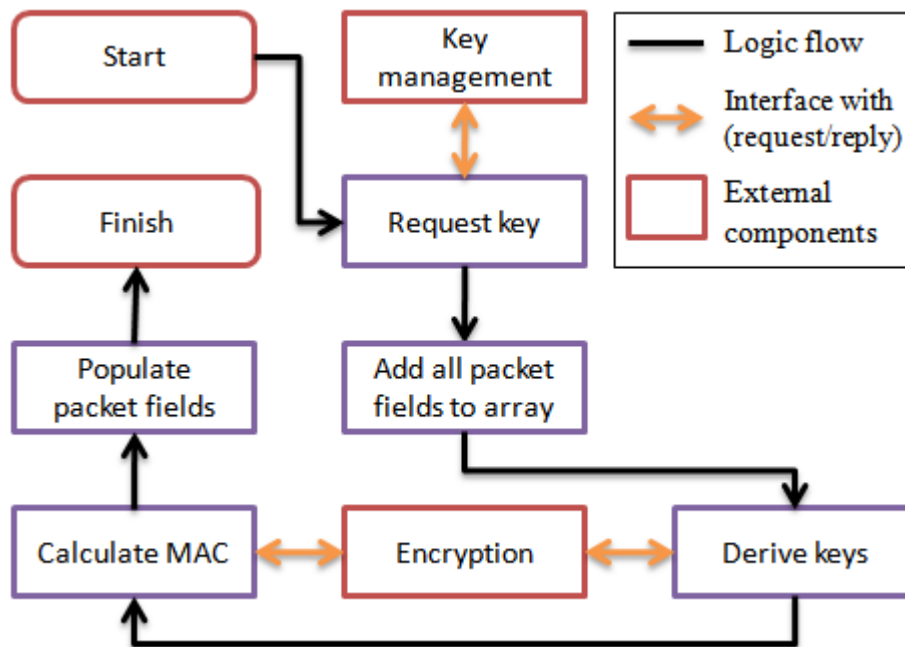


Figure 16: MAC generation process

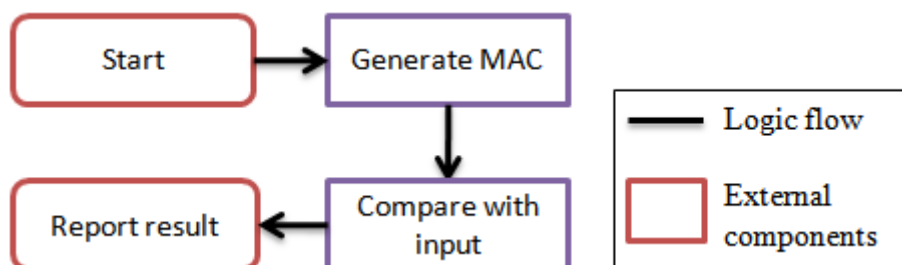


Figure 17: MAC validation process

4.2.5 Control component

Interfaces and operation

The control component interfaces with TinyOS and every other SecRose component in order to coordinate secure communication.

When the higher layers of TinyOS wish to send a packet, they pass it to the control component. The packet's contents are then built and processed to facilitate packet management, encryption and authentication. This process is done by requesting the related services of other SecRose components. Finally, the packet is passed to the lower TinyOS layers for transmission.

Packets are received by the lower layers of the TinyOS and are then passed to the control component. The component then decrypts and validates the packet. Valid packets are passed to the higher layers of TinyOS.

Note that packet data is a superset, which includes the packet fields and other information. Some of the packet data are not transmitted.

Packet management

The packet management system is an important subcomponent. It facilitates the three basic packet types: *normal*, *broadcast* and *long* packets. It also recognises and handles *acknowledgement* packets. Every packet has to be processed by packet management before any other action takes place.

Outgoing packets are given a *flag* value, depending on communication type. For incoming packets, the *flag* is examined and the actual destination address is derived from the *flag*. Outgoing packets are then passed back to the control component to continue the operations. Incoming packets continue to be handled by packet management until their reception is completed.

Packet management has to coordinate the actual transmission and reception because transmitted packet fields depend on packet type. Outgoing packets are given back to the packet management component when the other components have finished processing and the

packet is ready for transmission. Only the relevant packet fields are selected from the packet data for transmission.

The flags that are utilised by the packet management can be used for additional efficiency savings by facilitating early rejection of unwanted packets. The rejection happens in this component.

Outgoing packets

After packet management has processed the packet, the control component requests a MAC to be calculated by the authentication component. When this concludes, the packet is sent to the encryption component to encrypt it.

The packet is then passed back to packet management for transmission and when that finishes the control component informs the key management system that a packet transmission event has concluded. Finally, the control component is ready to process the next packet.

Incoming packets

When a packet arrives from lower layers, it is initially processed by the packet management subcomponent. The finalised packet is passed to the control component and the encryption component is invoked to decrypt the contents of the packet. Then the packet is passed to the authentication component to validate the authenticity and integrity of the packet. An acknowledgement packet is sent if the validation succeeds.

If validation fails, the control component asks the key management system to revert to the backup pair key and repeats the decryption and validation steps again. An acknowledgement packet is not transmitted in this case.

Acknowledgement packets

Authenticated acknowledgement packets are sent when packets are received and validated with the first attempt. The process is straightforward and invokes the control component and the packet management subcomponent only.

Acknowledgement packets do not contain any data and thus the encryption component is not invoked. The authentication component is not needed either, since it has already processed the received packet and has produced the required authentication data.

The control component feeds packet management with the acknowledgement packet data and the acknowledgement packet is generated on the fly by the field selection of packet management.

As it has been discussed already, there are cases where acknowledgement packets are not transmitted. This behaviour is a consequence of the original intentions of the acknowledgement packets and the subsequent system design.

The aim of acknowledgement packets is to maintain pair key state, not to confirm proper reception of a packet. Packet reception is a side effect of the acknowledgements. With the current SecRose operation specification, transmission of acknowledgement packets, when the initial MAC validation has failed, would result in loss of synchronisation.

If a receiver is required to use the *backup key* to validate a packet, it makes a permanent change in the *counter table*, promoting the *backup counter* to *active counter*. The system is not in ideal state but it is operational because the two *active counters* are synchronised. Transmission and reception of acknowledgement at this point would cause the sender to advance its *active counter* using the *counter update value*. It will therefore cause complete loss of synchronisation, since no counter will be synchronised after that.

SecRose intentionally operates in this way to avoid using energy-expensive acknowledgements when the key cannot be easily updated. It might be that both communication ends have calculated the *counter update value* and can therefore update the active key. However, the sender would have to acknowledge the initial acknowledgement in order to let the receiver know that it received that acknowledgement.

That would be an unnecessarily energy-expensive and complicated process with little benefit. Consider that the acknowledgement of the acknowledgment would also have to be acknowledged and so on. This research did not find a solution to this problem and thus SecRose is required to allow some key reuse to happen.

The SecRose proposal might be different from the usual but it is still consistent with the TinyOS acknowledgements. TinyOS includes a data-link layer acknowledgement mechanism, in which neighbouring nodes report that they have received the outgoing transmission. It does not give any information about what happened in the rest of the hops until the final receiver.

SecRose's acknowledgement operates in the transportation layer, informing the sender that the final receiver has received the acknowledgement or that some kind of error had occurred. Further work in SecRose might improve it to give acknowledgements with specific errors but they need to be carefully considered in terms of energy consumption.

4.2.6 Diagrams of system operation

Figure 18 and Figure 19 illustrates the steps and the system interactions initiated by the control component in order to achieve packet communication. The figures illustrate the process in the sender and the receiver covering every activity: packet transmission, reception, validation, sending of acknowledgement and the reception of it. Arrows indicate flow of data, logic or both. The colour of the boxes indicates the component in which the step is carried out.

The colour of the arrow represents the kind of flow. When the data of a packet is carried along in the system's logic, a black arrow is used. After transmission, the packet is copied on the receiving end but the packet's data remain in the sender as well, so the logic flow continues to complete the post-transmission tasks. The same applies after passing the packet to higher layers and before transmission of the acknowledgement. The actual transmission of data over the radio is indicated with a red arrow.

Orange coloured two-sided arrows indicate system component interactions. These are request/reply type of interactions and they are associated with data exchange between the components as well. A one-sided orange arrow represents an "informative" message between components. It is only present in Figure 19 when the control component informs key management that a packet with a valid MAC has been received.

The boxes describe the steps of the flow and their colour represents the components in which a step is taking place. This diagram focuses on the communication, which is coordinated by the control component, represented in purple.

All the other components are considered “external” and are illustrated with a red box. For clarity reasons, most interactions between external components are either not shown at all or simplified.

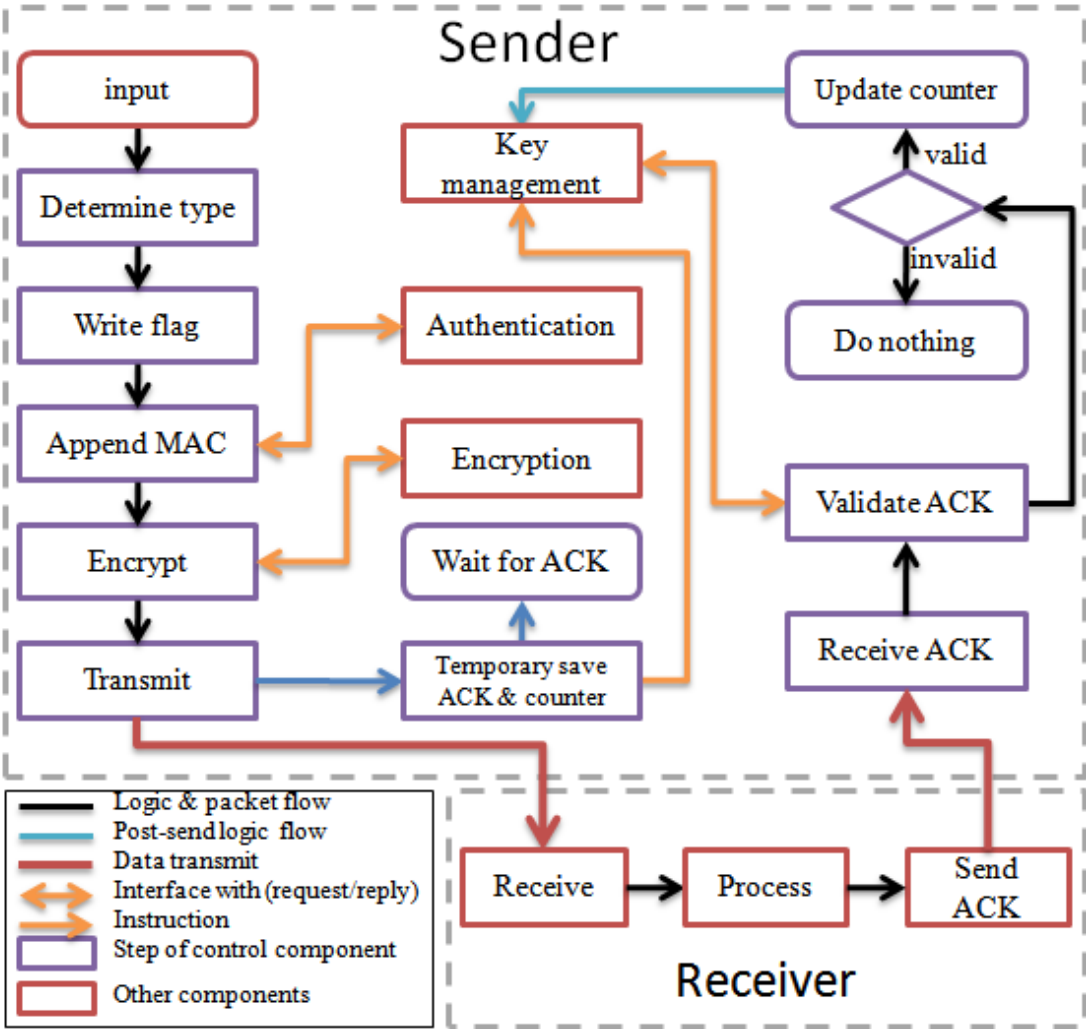


Figure 18: transmission of packets and reception of acknowledgements by the sender.

See Figure 19 for detailed receiver process.

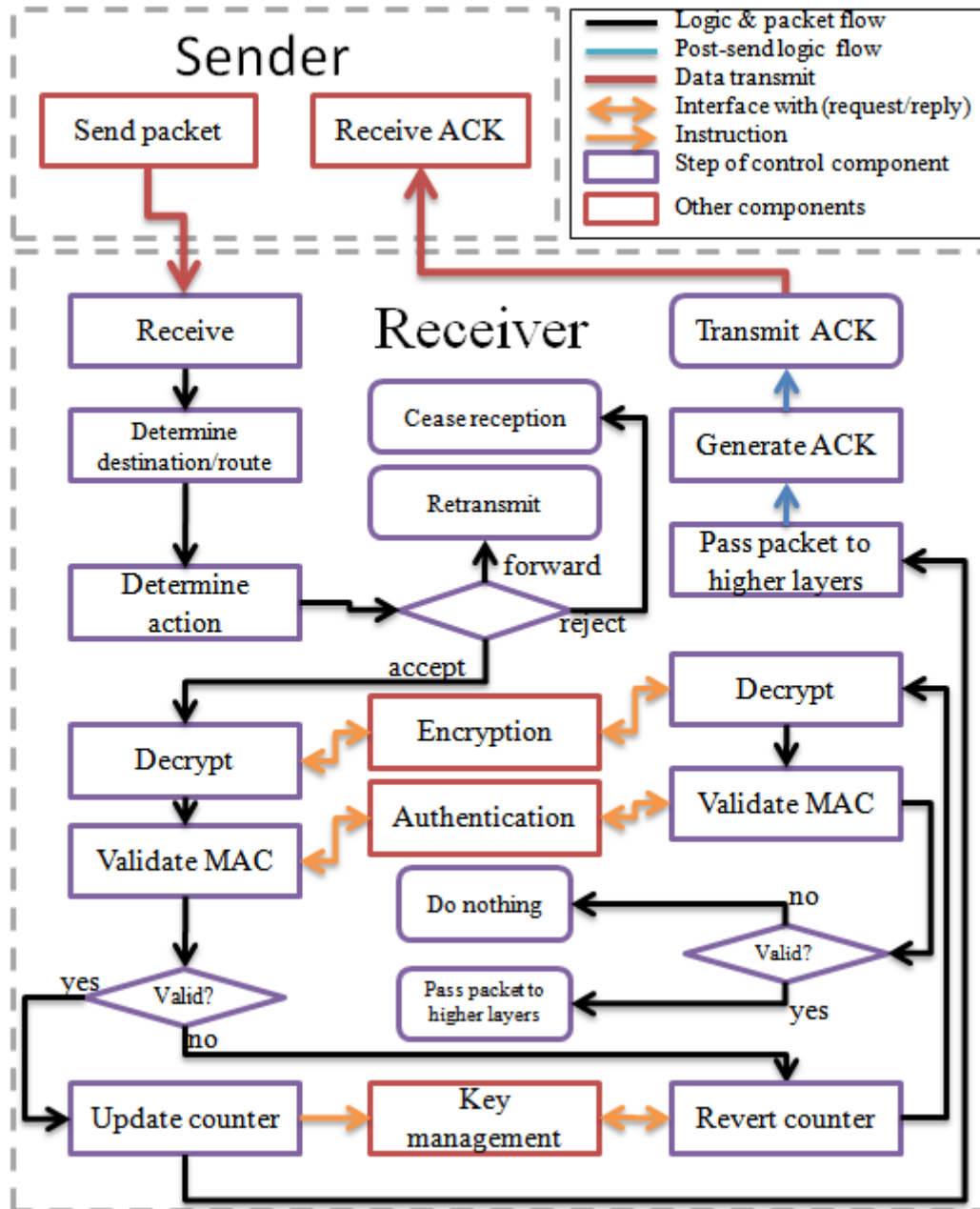


Figure 19: reception of packets, validation and transmission of acknowledgements.

Packets arrive from the sender, which is further illustrated in Figure 18.

4.3 Summary

4.3.1 Simple design, simple operation

Simplicity is a key quality of SecRose, which allows it to achieve both the security and energy efficiency targets. The same simplicity is adapted at every part of the design, from the way the cipher is coded to the MAC stealing mechanism and the acknowledgement validation.

Simplicity of operation

The basic operation of the SecRose mechanism is very simple and it can be described in a few lines:

- Sender: authenticate, encrypt, transmit, note future key, note bits of the un-transmitted MAC, wait for acknowledgement
- Receiver: receive, validate, decrypt, update key, send back bits of the un-transmitted MAC as an acknowledgement
- Sender: receive acknowledgement, validate, update key

The rest of SecRose has to do with the security level and the performance. The main unique security features of SecRose are visible; there is no other proposal that provides either authenticated acknowledgements or key management.

Clarity of design

SecRose consists of four, relatively small, components:

- Encryption; provides 128-bit encryption and the stealing techniques
- Authentication; provides digital signatures for packets
- Key management; changes the used keys as frequently as possible
- Control; controls the sequence of events and interfaces with other components

The existence of SecRose is invisible to the rest of TinyOS. The control component is implemented in the data transportation layer, mimicking the functionality of the part it replaces.

Relationship with existing proposals

TinySec was the only security mechanism for the data transportation layer of sensor networks when this research commenced. Inevitably, some of the TinySec features were good enough to not need changing and thus SecRose has retained some of TinySec's features.

Some of the TinySec features remain largely unchanged. For example, SecRose retains the 4-Byte MAC, and the CBC mode to calculate it, as it is deemed sufficient for most of the applications.

MiniSec was developed, probably in parallel with SecRose, as an alternative to TinySec. SecRose does not adapt any features introduced by MiniSec but the proposals share the data overloading technique. However, the idea is used it on different data; SecRose uses an overloaded flag while MiniSec overloads the counter part of the IV.

SenSec is an attempt to improve TinySec and build on this platform. It does not provide important new features other than the increased key size and the one-pass encryption and authentication method using a cipher in OCB mode. SecRose does not relate to SenSec any more than it relates to TinySec.

4.3.2 Innovative features

Innovations in security

SecRose provides unique security features designed to elevate the overall security provision.

SecRose's main security contributions are:

- 128-bit encryption strength supported natively
- Key management with frequent key changes
- Authenticated acknowledgements

Each of these features addresses an important security issue. The 128-bit security is the least acceptable strength. The key changes provide freshness and semantic security in an alternative way and the authenticated acknowledgements enable many routing protocols to function securely.

Innovations in energy efficiency

SecRose includes important energy features like:

- three packet types to fit communication patterns better
- utilisation of data overloading
- unique ciphertext stealing implementation
- key synchronisation by meta-information with no radio data exchange

4.3.3 Chapter conclusion

SecRose provides a simple, secure and efficient design, which retains good security features of other proposals while it solves problems that have not been addressed in the past.

The implementation of these features on code is discussed in the next chapter while the evaluation of the security provision and energy requirements is presented in Chapter 6.

Chapter 5

Implementation

5. Implementation

The implementation of SecRose in the TinyOS Operating System is discussed in this chapter. Technical information on TinyOS and its relationship with SecRose is given in the first subsection. The second subsection discusses the algorithms used to produce the code, pseudocode of the whole implementation and some examples of actual code. The final subsection illustrates the running of the application in TOSSIM and Avrora.

5.1 TinyOS and SecRose

5.1.1 Description of TinyOS

The TinyOS Operating System

TinyOS is a miniature operating system designed for use in sensor networks and other embedded systems written in the nesC programming language [31, 53, 64]. TinyOS is a flexible, application-specific operating system, which follows a component-based, event-driven model.

The system is built from a set of reusable components that are assembled into an application-specific executable, capable of running in a sensor board. Instead of multithreading, TinyOS provides an event-driven concurrency model, which utilises *components* that interface with each other.

Component model of TinyOS

Every TinyOS program is a graph of *components*, which interact together using three computational abstractions: *commands*, *events* and *tasks*. The programming model of nesC provides component creation and access in the form of services, which are called *interfaces*. An application connects *components* using a *wiring specification*, which defines which *components* will be used by the application. This mechanism excludes the *components* that are not required from the executable.

Commands are requests for a *component* to perform some service. When the operation concludes, an *event* informs the commanding *component* of its success or failure. *Commands* and *events* might also post a *task* to be executed at a convenient time by the operating system's scheduler. Both hardware and software resources are abstracted by TinyOS as *components*. The *component* system manages the underlying software and hardware interfacing at low level while the application is only waiting for an *event* to inform it that the *command* has finished.

Each *component* defines a number of *interfaces* that it *provides* or *uses*. The interfaces specify how the *component* may interact with other *components*. *Interfaces* contain both *commands*

and *events*. A *command* is a function implemented by an interface provider while an event is a function implemented by its user

Components can be *modules* or *configurations*. *Modules* contain the actual code of the *component* while *configurations* connect this *component's* code together with other *components*. The connection process is called *wiring*. Via this process, a *configuration* might *wire* a number of *components*, which in turn may be *configurations* that wire more *components* resulting to the creation of a tree of *components*. The tree is then called a *supercomponent* and in that case, the module of the *component* in the base of the tree is controlling the tree. Essentially, applications are *supercomponents*.

5.1.2 Operation of TinyOS

Programming and execution model of TinyOS

TinyOS itself and the applications that run on it are written in nesC; an event-driven dialect of C. NesC is a dialect of C, it is no different from C other than in the way that functions and libraries are defined and linked. As opposed to C, there is no linking in nesC. The linking process is replaced by the *wiring* of *components*.

TinyOS provides a scheduler, or execution model, which coordinates simultaneous *commands* and *events* that run between the *components*. The scheduler also manages energy consumption, puts hardware components into sleep mode and other housekeeping tasks. The scheduler allows for real-time task execution and for low priority execution.

NesC and the scheduler are designed to solve a number of engineering issues regarding event-driven execution. The system for example can detect data races statically. The execution model consists of run-to-completion tasks for computation and asynchronous interrupt handlers signalled by the hardware. The scheduler might execute tasks in any order but it must obey the run-to-completion requirement.

Simulators

Due to the nature of wireless sensor networks, development of complex applications might be a difficult process requiring expensive hardware [60-61]. A number of development tools and

methodologies have been proposed to assist the process and minimise the development constraints.

The most important are the simulation tools. They consist of software that attempts to represent the behaviour of the sensor hardware on a personal computer. They create instances of virtual hardware and deploy them on a virtual network. The development of SecRose used the TOSSIM [62] and Avrora [63] simulators and the `avr-gcc` compiler [133] and the associated debugging modes and tools.

TOSSIM [62] is the pc simulator that comes with TinyOS as a tool for development and testing using a desktop computer. It works by abstracting the hardware-specific functions, e.g. the memory, into C functions with similar input and output. TOSSIM is able to produce a pc executable, which simulates the WSN when run, allowing the sensor network applications to be run as if they were normal executables for the pc architecture.

Avrora [63] uses a different, slower but highly accurate approach to simulate sensor networks. Its authors describe it as an instruction-level sensor network simulator, which runs the actual microcontroller programs in a simulated environment. Avrora implements an event queue [134] to coordinate events. This preserves the accuracy and correctness of the simulator. In addition, it utilises a radio synchronisation technique, which allows the nodes to communicate with precision timing.

5.1.3 Communication model and SecRose's position

Communication facilities of TinyOS

TinyOS provides a networking architecture, implemented as a combination of communication components. These components define how packets are requested by the application, constructed by the operating system and handed to the hardware radio component for transmitting.

The communication facilities of TinyOS are much lighter and they look different from traditional communication protocols and specifications. However, it is apparent that the

design has been influenced by (a) the OSI model [66], (b) the TCP/IP protocols[9, 135] and (c) the 802.11 MAC protocols [67].

TinyOS uses a layering model but since it is extremely lightweight, some of the OSI definitions are missing and others are combined. One or more components are used to achieve the computational tasks involved. The following layers can be distinguished:

- Application layer, expressed by the application
- Transport layer, multiple components achieve the functionality of generating and pre-processing a packet with its headers and data. Key components are;
 - `AMStandard`, initially generates a packet with its header and data
 - `MicaHighSpeedRadioM`, for mica2 nodes, is responsible for sending the packet to lower layers on a byte-by-byte basis
- Network layer, also facilitated in a number of components, most importantly;
 - `ChannelMonC`, responsible for monitoring the channel for idleness and/or transmissions
 - `SpiByteFifoC`, is responsible for sending the packet to the hardware in a bit-by-bit stream

Note that there is no physical layer. Medium access is controlled by the transport layer and the network layer is responsible for the actual data transmission.

TinyOS packets are defined to have the following fields:

Destination	Type	Length	Group	Data	CRC
2	1	1	1	0...29	2

The numbers express size in bytes. The Type field, sometimes referred to as AM, is equivalent to the TCP port field, the Group field is intended to allow clustering or other type of node grouping. The data contain the packet's data payload, which ranges from 0 to 29 bytes. The exact size of the payload is defined in the length field. Note the lack of a source address.

The acknowledgement packet is a simple stream of four identical bytes, which operates in the physical layer, and is sent by every node that happens to receive any packet.

SecRose in relation with TinyOS

SecRose updates the transportation-layer components of TinyOS and it is therefore facilitated in the `MicaHighSpeedRadioM` component. It also introduces and interfaces with a new `nesC` component, called `SecRose`.

The `MicaHighSpeedRadioM` component accepts outgoing packets from the `AMStandard` component and sends them to radio via the `SpiByteFifoC` component. Incoming packets are intercepted by `ChannelMonC` and are sent to `MicaHighSpeedRadioM` as well.

SecRose's implementation replaces the most significant parts of `MicaHighSpeedRadioM` and communicates with `SecRose` for additional functionality. The control component of `SecRose`¹² is primarily implemented in `MicaHighSpeedRadioM` while all the other components reside in `SecRoseM`.

¹² Described in Section 4

5.2 Algorithms, code and pseudocode

Files

The nesC component `SecRoseM` provides a series of SecRose-specific commands, which are used either internally by this component or by the `MicaHighSpeedRadioM` component. These commands are self-contained and can be described individually. This section describes those self-contained SecRose commands. They are defined in file `SecRose.nc` and implemented in file `SecRoseM.nc`.

Notes

Most commands described in this section do not return anything. The implementation of SecRose is based on passing pointers, instead of passing data. Commands manipulate the data pointed by the pointer and return control to their parent command, which then utilises the data. This practice is more efficient and consistent with the rest of TinyOS and allowed by its race protection and by the AVR microprocessors.

In the cases where actual code is provided, effort is made to reduce its size. The actual code is retained in the appendix.

This section contains integer values expressed with alphanumeric constants, like for example `TRUE`, `FALSE`, `PKT_SENT` and many others. These are automatically generated using C-style enumerated list like:

```
enum { TRUE, FALSE }
```

Such code can be placed in header files or other a globally scoped location and the pre-processor will automatically assign integer 0 to `TRUE` and integer 1 to `FALSE`. The `enum{ }` definition can enumerate many items.

5.2.1 Encryption component

Commands

The encryption component, as described in Chapter 4, is implemented in code using four commands:

- `bteaCipher()` – the implementation of XXTEA
- `packetEncDec()` – called by the control component when it requires encryption of a packet with stealing
- `stealEncDec()` – called by `packetEncDec()`, facilitates stealing

XXTEA for TinyOS – `bteaCipher()`

The XXTEA [136] implementation for TinyOS and other AVR microprocessors is provided. This code differs from other published versions. The MC constant is hardcoded, the cycles are fixed to 8 and the $n-1$ is calculated only once, instead of once for every cycle. The code for decryption is not provided here since it is the exact reverse of encryption.

Input

- `vl` – the pointer to the plaintext data
- `n` – the length of the data
- `k` – a pointer to the key

Code

```
1  async command void SecRose.bteaCipher(uint32_t* vl, int32_t n, uint32_t* k) {
2  uint32_t z, y=vl[0], sum=0, e, DELTA=0x9e3779b9;
3  uint8_t n_minus_one; uint32_t p, q;
4
5  n_minus_one = n-1; q = 8; z = vl[n_minus_one];
6  while (q-- > 0) {
7  sum += DELTA; e = (sum >> 2) & 3;
8  for (p=0; p<n_minus_one; p++) {
9  y = vl[p+1];
10 z = vl[p] += (z>>5^y<<2) + (y>>3^z<<4) ^ (sum^y) + (k[p&3^e]^z);
11 }
12 y = vl[0];
13 z = vl[n_minus_one] += (z>>5^y<<2) + (y>>3^z<<4) ^ (sum^y) + (k[p&3^e]^z);
14 }
15 }
```

Remarks

Line 1; illustrates one of the main differences of nesC and C. The line declares an asynchronous command with void output named `bteaCipher`, which resides in the `SecRose` component. The rest of the code is essentially C code.

Lines 2, 3 and 5; declare and assign values to temporary variables and constants.

Line 6; the XXTEA cycle starts. Since $q=8$ it will be run 8 times.

Packet encryption or decryption – packetEncDec() and stealEncDec()

The commands `packetEncDec()` and `StealEncDec()` are part of the encryption component. `packetEncDec()` is called by the control component when packet encryption service is required. It accepts a packet's data as input, and asks the key management component to provide a key. Then the command `StealEncDec()` is called to examine the payload size, decide and apply any efficiency optimisations.

This pair of commands could have been one command but they are left as two commands to allow for easier transition to different ciphers and different stealing methods. The current code splits other tasks from the actual encryption task. However, the process given below describes both commands as a uniform process.

Input

`data` – a pointer to a packet struct

`action` – a Boolean. Can be set to ENCRYPT or DECRYPT to define the desired action

Process

1. If `data->length` is zero then exit.
2. Determine the destination from the flag and the `data->addr` field
3. Request the pair key from key management, assign it to array `fkey[]`
4. Declare array `v[]`, populate it with the whole array `data->data[]`
5. If the last block of `v[]` is not a full block¹³ then add up to four bytes from `data->mac[]` to `v[]` in order to make it complete. Note how many bytes were added
6. If the last block is still not full; add `data->type` to `v[]`. Note the action
7. If the last block is still not full; add `data->group` to `v[]`. Note the action
8. If the last block is still not full; pad `v[]` with zero. Note the action
9. Encrypt or decrypt `v[]` according to what the `action` variable was

¹³ The “full block is considered to be 4-bytes long in this case. See *Remarks*.

10. Take `data->length` bytes from `v[]` and place them on `data->data[]`
11. If `data->type` was stolen, take a byte from `v[]` and add it to `data->type`
12. If `data->group` was stolen, take a byte from `v[]` and add it to `data->type`
13. If padding byte was used, take the last byte of `v[]` and add it to the end of `data->data[]`

Remarks

Line 1 and 9; Padding will only occur in packets with 1 byte payload only. The complete 8-byte block is formed by; 1 data byte, 4 MAC bytes, 1 type byte, 1 group byte and 1 pad byte. Packets with zero data payload are obviously not encrypted since there is nothing to encrypt.

Lines 2 and 3; on reality `packetEncDec()` ends in line 2 and `StealEncDec()` starts in line 3

5.2.2 Authentication component

Commands

The authentication component, as described in Chapter 4, is implemented in code using four commands:

- `calcMAC()` – accepts a packet and calculates its MAC code, called by the control component
- `validateMAC()` – accepts a packet, calculates the MAC of it and compares the result with the MAC written on the packet

Mac calculation – calcMAC()

The command that implements the CMAC specification is algorithmically described here. This command is the vital subcomponent of the authentication component.

Input

`data` – a pointer to a packet struct

`node` – the true source/destination node (`data` might lack this information)

`action` – a Boolean set to `true` if the packet is outgoing.

Process

1. Request the pair key from key management, assign it to array `fkey[]`
2. Declare array `v[]` add `data->data[]`, `data->source`, `data->type` and `data->group` to it (these are all the available fields of the packet)
3. Declare array `kbuffer[]`. Populate it with the encrypted output a string of eight zeroes.
4. Note if the most significant bit (MSB) of `kbuffer[]` is 0
5. Shift the whole `kbuffer[]` one bit to the left
6. Declare array `kalpha[]`, copy `kbuffer[]` to `kalpha[]`, shift it 1 bit left
7. If the MSB of `kbuffer[]` was not 0, then XOR `kalpha[]` with `0x1b`
8. Declare array `kbeta[]`, copy `kalpha[]` to `kbeta[]`, shift it 1 bit left
9. If the MSB of `kbuffer[]` was not 0, then XOR `kbeta[]` with `0x1b`
10. If the last block¹⁴ of `v[]` is a full block then XOR it with `kalpha[]`
11. Else append 1 set bit after the last bit of `v[]`
12. XOR the last block of `v[]` with `kbeta[]`
13. XOR a string of 0's with the first block of `v[]`, place the result in `kbuffer[]`
14. XOR the next block of `v[]` with `kbuffer[]`, place the result in `kbuffer[]`
15. Encrypt `kbuffer[]` using key `fkey[]`
16. Repeat steps 14, 15 for all blocks of `v[]`
17. Copy the first four bytes of `kbuffer[]` to `data->mac[]`
18. Copy the next byte of `kbuffer[]` to `data->count_value`¹⁵
19. For outgoing packets, copy the next two bytes to `data->ack_value[]`
20. Else copy them to `data->ack[]`

Remarks

The block is considered 8-bytes long. `kbuffer[]`, `kalpha[]` and `kbeta[]` are all 8-byte long arrays. A “block” in the CMAC specification refers to a string of bytes equal in length to the minimum block size of the cipher. XXTEA’s block size is 4 bytes but the minimum input size cannot be less than 8 bytes. The CMAC process is invoking the cipher with the minimum input. Therefore, the minimum block size for this case is 8 bytes.

¹⁴ The “full block” is considered to be 8 bytes long in this case. See *Remarks*.

¹⁵ This is going to be the *counter update value*.

The process uses the data on `fkey[]` to create two more keys which are stored in `kalpha[]` and `kbeta[]`. Their creation depends on the input's size in relation to the block size.

`kbuffer[]` is a temporary array which SecRose uses to store output and process it later.

MAC Validation – validateMAC()

The command `validateMAC()` accepts a received packet as input and validates the Mac written on it against the MAC that can be calculated by the received packet fields.

Input

`data` – a pointer to a packet struct

Process

1. Create a copy of the `data->mac[]` as received, place it in `mac_tmp[]`
2. Call `calcMAC()`, with `data` as input, to calculate the received MAC. The result will be placed in `data->mac[]` as normal
3. Compare `data->mac[]` with `mac_tmp[]`
4. Return TRUE or FALSE accordingly

5.2.3 Key management

Commands

The key management component, as described in Chapter 4, is implemented in code using four commands:

- `mixKey()` – obtains the counter from the counter table and mixes it with the initial key. Called by any component that requires the key of a pair
- `counterHandle()` – handles the counter table, provides the three SecRose system interfaces that relate to the counter table. Also calls the `ackTable()` command in order to consult the acknowledgement table
- `ackTable()` – handles the acknowledgement table, called by `counterHandle()`

Structure of the counter table

The counter table is represented as an array of struct with the following fields:

Definition:

```
typedef struct ROSE_counter
{
    uint32_t active;
    uint32_t backup;
    uint8_t temp;
} ROSE_counter;
```

The active field stores the active counter, the field backup stores the backup counter and field temp, which is only 8 bits, stores the counter update value. The initialisation adds 1 to the total number of nodes to reserve a place for the broadcast pair.

After initialisation, the table is accessible as an array of structs via commands like:

```
counter[node].temp = cur_counter;
```

The node variable represents the node number. For a node sitting on one side of a pair, the ID of the other node is effectively the ID of the pair.

Structure of the acknowledgement table

The acknowledgement table is represented as an array of structs with the following fields:

```
typedef struct ROSE_acks
{
    uint16_t addr; /* 2 DESTINATION addr */
    uint8_t ack[2]; // 2 needed in all packets
} ROSE_acks;
```

The field `addr` stores the destination address to which the packet went. This is also equal to the pair ID, as discussed before. The array `ack[]` stores the two `uint8_t` integers which are provided by the authentication component and represent the acknowledgement value of the packet.

After initialisation, the table is accessible as an array of structs, like the counter table. However, there is no known index, like the node's number, as in the counter table. The array has to be parsed and the value stored in `ack[]` is checked against the received acknowledgement value in order to relate it with a node id (stored in the `add` field).

Key requests - mixKey()

Key requests from the encryption and the authentication components are handled by command `mixKey()` which retrieves the active counter from the counter table and mixes it with the initial key.

For the requirements of this proof-of-concept, the 96-bits initial key is hardcoded into the memory. A temporary placeholder for the final key is also defined as `fkey[]`. Both arrays are global and are defined with a random value.

The command accepts the destination of the node as input. The clear systemic design represents this command as an interface that should accept the packet as input. Precisely applying this to code would have resulted in unnecessary performance loss since the destination is readily known in all cases where this command is called.

Handling of the counter table – counterHandle()

The four interfaces defined in Section 4 are all implemented in command `counterHandle()`.

Input

`data` – a pointer to a packet struct

`action` – an integer representing the action

Process if action is PKT_SENT (system interface “Packet sent”)

1. Determine the value of the flag and put it in variable `flag`
2. If `flag = 0` then set variable `node = 0`
3. If `flag = 1` then do the following, otherwise go to step x
4. Set variable `node` equal to the maximum number of nodes (`SECROSE_MAX_NODES`)
5. Retrieve active counter and copy it to the backup counter:

```
counter[node].backup = counter[node].active;
```

6. Retrieve active counter and add the *counter update value* to it¹⁶:

```
counter[node].active += cur_counter;
```

7. Set: `counter[node].temp = cur_counter;`

¹⁶ The value is determined by the Authentication component in step 18 of command `calcMAC()`.

8. Call `ackTable()` with input the data packet struct, action set to `ADD_VALUE` and the node variable

Process if action is PKT_RECV or CNT_REVERT (system interfaces “Packet received” and “Invalid packet received”)

1. Examine the value of `data->addr`. If it is the broadcast address set `node = SECROSE_MAX_NODES`. Otherwise set `node = data->source`
2. If the action is `PKT_RECV` then back-up and increase the active counter (as in the steps 5 and 6 of the `PKT_SENT` process)
3. If action is `CNT_REVERT` then copy the backup counter to the active counter:

```
counter[node].active = counter[node].backup;
```

Process if action is ACK_TEST_START (system interface “Acknowledgement received”)

1. Loop around `acks[]` until a value that matches `data->mac[]` is found in `acks[].ack[]`.
2. If a match is found then:
 - a. increase the active counter by the temporary counter value¹⁷:

```
counter[node].active += counter[node].temp;
```
 - b. Call `ackTable()` with input the data packet struct, action set to `REMOVE_VALUE` and the node variable which contains the position in the `acks[]` table where the received acknowledgement value was discovered

Remarks

Step 2b passes the discovered position to the `ackTable()` command to save from repeating the search.

Handling of the acknowledgement table – counterHandle()

The command is called by the `counterHandle()` command. The command branches depending on the value of the action input, which can be `ADD_VALUE` or `REMOVE_VALUE`. The command uses the global variable `num_acks`, which stores the total number of acknowledgement values that the sender expects.

¹⁷ Stored in the temp field in step 7 of the `PKT_SENT` action.

Input

`data` – a pointer to a packet struct
`action` – an integer representing the action
`node` – the node/pair ID under handling

Process if action is ADD_VALUE

The concept is to push the whole array one place to the right, like a LIFO table.

1. Increase variable `num_acks` by 1
2. If `num_acks` has reached its maximum value then decrease it by one. This will cause the older acknowledgement value to be overwritten
3. Loop around the whole array and shift it one place to the left so that all the fields in `acks[0]` will be copied in `acks[1]` and so on. The location `acks[0]` will be freed and the location `acks[SECROSE_MAX_ACK]` will be overwritten if it exists
4. Add the values of `data->ack_value[]` to `acks[0].ack[]` and the value of `node` to `acks[0].addr`

Process if action is REMOVE_VALUE

The concept is to push the parts of the array that are on the right of the discovered acknowledgement value one place to the left. Decrease `num_acks` by 1

1. Start at the position `node` and copy the fields of position `node+1`
2. Repeat step 1 until the end of the table is reached

Remarks

Under normal operation, the position of the acknowledgement value of the last sent packet is in `acks[0]` and thus discovered immediately by `counterHandle()`

5.2.4 Flag manipulation

The following commands are used by any system component and any command that needs flag handling or querying. They facilitate reading, writing and deleting of the flag on the *length* field of the packet.

Determine flag from whole packet - findFlag()

Accepts a packet's data as input, reads the `data->addr` field and determines what the flag should be for this destination address. This command is suitable for flag determination of outgoing packets, needed at various stages of the mechanism.

Read flag from length only – readFlag()

Accepts the length field of a packet and determines the flag by examining size of the field. This command is used by the control component to understand the packet type immediately after receiving the length – the first field of the packet.

Determine and write – writeFlag()

This command is similar to `findFlag()` but it also facilitates overloading of the flag in the packet's *length* field. In case the packet is an acknowledgement, the flag cannot be read by the `addr` field but it was pre-set by the control component. The function will write the acknowledgement flag on the length regardless.

Determine and extract the flag - fixFlag()

The command examines the length of the inputted packet, determines the flag and then deletes it from the length field.

5.2.5 Control component

Files

The component `MicaHighSpeedRadioM` is the TinyOS component that facilitates the transportation layer. `SecRose`'s control component is implemented in this file. The component calls commands from the `SecRoseM` component. The component is defined in file `MicaHighSpeedRadio.nc` and implemented in file `MicaHighSpeedRadioM.nc`.

Notes

This subsection discusses the PC version of the component. `SecRose` was also implemented for the MICA2 architecture. There are minor differences in the component names and the overall flow of control between different architectures but the concept ideas remain the same.

The component is based on *events* and thus there are time delays in the execution of the flow.

The position of the *events*, *tasks* and *commands* in the file does not match the position of their explanation in this subsection. The subsection groups packet sending tasks and packet reception tasks. The file does not follow any particular grouping, since the order does not matter.

This description does not include radio-timing tasks, which are irrelevant to SecRose.

Global variables

SecRose defines a number of temporary global variables for this component. The important ones are described here:

States

There are two variables regarding states in this component. The variable `state` controls the various stages where the sending or receiving process is. The variable `send_state` determines whether an outgoing packet is being processed by `MicaHighSpeedRadioM`. This is used to deny higher layers from sending subsequent packets until the current packet finishes.

Other variables

- `next_data` contains the next byte, to be transmitted shortly
- `data` contains the byte that was just received
- `rec_ptr` is the pointer to the received packet's data struct
- `send_ptr` is the pointer to the outgoing packet's data struct

Interfaces

This component uses and provides many interfaces. Some of the provided interfaces are important to SecRose. They are defined on other components but implemented in `MicaHighSpeedRadioM` under the SecRose design specification. These are:

`BareSendMsg` (defined with the name `Send`) – the interface to send packets, SecRose implements the command `Send.send()` which is triggered when higher layers want to send a packet.

`SpiByteFifo` – the interface that controls this component with the radio component

`SecRose` (defined with the name `rose`) – the `SecRoseM` component described in the previous subsection

Pseudocode structure of `dataReady()`

The command `dataReady()` is both complicated and important. It is common to both sending and receiving packets. The exact action is controlled by the state variable. The command is called by the `SpiByteFifo` component continuously, after 8 bits could have been received or sent, regardless if they actually did. In receiving mode, the command is called with the received byte as the value of the `data` variable. In sending mode, the `data` variable is set to 0.

The structure of the `SpiByteFifo.dataReady()` command is as follows;

```
event result_t SpiByteFifo.dataReady(uint8_t data) {
    if(state == TRANSMITTING_START) { ... }
    else if(state == TRANSMITTING){ ... }
    else if(state == RX_STATE) { ... }
    else if(state == SENDING_STRENGTH_PULSE){ ... }
    else if(state == REC_STRENGTH_PULSE) { ... }
    return 1;
}
```

Depending on the state, a different process is initiated. This subsection has grouped the distinct processes of sending and receiving and it will only describe the relevant branch of `SpiByteFifo.dataReady()` for each one.

Outgoing packet

When a packet arrives from the higher layers, it triggers the command `Send.send()`. Then the process of sending a packet initiated, involving various events, tasks and commands. The input to the process is the pointer to the packet's data, which is stored in variable `send_ptr`. The process explanation follows.

Process of command `Send.send()`

1. Check if there is not another packet in the process of being sent. If yes then return `FAIL` and exit.
2. Set `state = SEND_WAITING`
3. Set the source of the packet (`send_ptr->addr`) to the ID of the local node

4. Calculate the MAC or the ACK of the packet (depending on type) by calling `rose.calcMAC()` or `rose.calcACK()`
5. Encrypt the packet by calling `rose.packetEncDec()` with action `ENCRYPT`
6. Determine the exact size of the packet's fields, and the total number of bytes in the packet note the values for use later
7. Write the flag on packet's length by calling `rose.writeFlag()`
8. For *normal* packets, set the destination field to the ID of the local node¹⁸
9. Call `ChannelMon.macDelay()` and exit

The command `ChannelMon.macDelay()` is listening to the radio medium for a gap in transmission. The command is implemented in another component but when a gap is found it triggers the *event* `ChannelMon.idleDetect()`, which is implemented in `MicaHighSpeedRadioM`.

Process of ChannelMon.idleDetect()

1. If state is `SEND_WAITING`, do the following;
 2. Set variable `rx_count = 0`
 3. Set `send_state = IDLE_STATE`
 4. Set `state = TRANSMITTING_START`
5. Call `SpiByteFifo.send()` to send the first byte of the preamble

The command `SpiByteFifo.send()` accepts one byte input and sends it to the radio. The command is implemented in another component but when it completes its task it triggers the *event* `SpiByteFifo.dataReady()` which is implemented in `MicaHighSpeedRadioM`. That means that the *event* `SpiByteFifo.dataReady()`, described in the following lines, will be triggered a number of times, at least once after `SpiByteFifo.dataReady()` is called and concluded.

Transmission process of SpiByteFifo.dataReady()

1. If state is `TRANSMITTING_START`, follow the steps to send the preamble
 - a. Send the next preamble byte by calling `SpiByte.Fifo.send()`

¹⁸ Normal packets do not need a destination address; their destination is the base station. However, they do need a source address. `SecRose` sends the source address in place of the destination address. Swapping of address in the packet's data struct simplifies the code.

- b. Note how many preamble bytes have been sent
- c. If all preamble bytes have been sent
 - i. set `next_data` to the first byte of `send_ptr`
 - ii. Set `state = TRANSMITTING`
2. If `state = TRANSMITTING`, follow the steps to send the packet's bytes
 - a. Send `next_data` by calling `SpiByte.Fifo.send(next_data)`
 - b. If there is more bytes, set `next_data` to the next packet byte¹⁹
 - c. Otherwise set `state = SENDING_STRENGTH_PULSE`
3. If `state = SENDING_STRENGTH_PULSE` and follow the steps to send the signal strength bytes:
 - a. Send two signal strength bytes one-by-one by setting them to `next_data` and then calling `SpiByte.Fifo.send(next_data)`
 - b. Set `state = IDLE_STATE`
 - c. Call `SpiByteFifo.idle()`
 - d. Call `ChannelMon.startSymbolSearch()`²⁰
 - e. Trigger event `packetSent()`

Process of event `packetSend()`

1. Set `send_State = IDLE` and `state = IDLE`
2. Inform higher layers about the packet send event
3. Call `rose.counterHandle(send_ptr, PKT_SENT)`

Incoming packet

When at its normal state, the radio component listens to the channel for packet preambles. When a preambles is found it triggers the event `ChannelMon.startSymDetect()`, which is implemented in `MicaHighSpeedRadioM`.

Process of `ChannelMon.startSymDetect()`

1. Set `state = RX_STATE`
2. Reset temporary variables to default values

¹⁹ Uses the information about the total size of packet's fields, as determined in step 7 of `Send.send()`.

²⁰ The command `ChannelMon.startSymbolSearch()` returns the radio component to its normal state. `SecRose`'s control flow continues at event `packetSent()`.

At this stage, SecRose is ready to receive the bytes of the packet. All bytes “appear” in the MicaHighSpeedRadioM component as triggers of the SpiByteFifo.dataReady () event. The reception and triggering is implemented in the SpiByteFifo component. SpiByteFifo.dataReady () in the MicaHighSpeedRadioM component is triggered every 8 radio bits.

The reception process of SpiByteFifo.dataReady () is executed when state = RX_STATE and it is as follows;

Reception process of SpiByteFifo.dataReady()

1. If this is the first byte then
 - a. Place the received byte at the beginning of rec_ptr
 - b. Call rose.readFlag () to determine the flag
 - c. Call rose.fixFlag () to extract the flag
 - d. Determine packet’s faith; accept/forward/reject²¹
 - e. Determine the exact packet fields size note for later use
2. If it was not the first byte and not the last byte
 - a. Determine the location that the next received byte should be placed in rec_ptr and place it there²²
 - b. Place the received byte at the next location
3. If it was the last byte, initially execute the following tasks
 - a. For *normal* packets, swap destination address with source address. Then set source address to the address of the base station
 - b. For *broadcast* packets, set destination address to the broadcast address. Then set source address to the address of the base station
 - c. For acknowledgement packets
 - i. Call rose.counterHandle () with action ACK_TEST_START to check the validity of the acknowledgement
 - ii. Set state = REC_STRENGTH_PULSE
 - iii. exit

²¹ The rest of the process assumes that the packet is accepted. SecRose does not currently implement forwarding or rejection.

²² Uses the information about the exact fields size, noted in step 1a.

4. Then execute the decryption and validation tasks (for all packets except acknowledgements)
 - a. Create a copy of the packet as received
 - b. Decrypt packet
 - c. Call `rose.validateMAC()` to validate the MAC
 - d. If the MAC is not valid, do the following
 - i. Call `rose.counterHandle()` with action `CNT_REVERT` to revert the counter to the backup counter
 - ii. Decrypt the copy of the packet
 - iii. Call `rose.validateMAC()` to validate the copy of the packet with the new counter
 - iv. If `mac_valid` is now `TRUE` then keep the copy of packet and discard the original
 - e. If the MAC was valid then note it down in order to send an ACK later
5. Set `state = REC_STRENGTH_PULSE`

Process of SpiByteFifo.dataReady() when state = REC_STRENGTH_PULSE

1. Trigger event `packetReceived()`
2. Set `state = RX_DONE_STATE`
3. Call `SpiByteFifo.idle()` to set radio state to idle

Process of packetReceived()

1. Call `ChannelMon.startSymbolSearch()`²³
2. If the received packet was an acknowledgement packet then do nothing
3. Otherwise do the following
 - a. Inform higher layers about a packet reception event
 - b. Call `rose.counterHandle()` with action `PKT_RECV`
 - c. Create an acknowledgement packet, call `TrueSend.send()` to send it

²³ The command `ChannelMon.startSymbolSearch()` returns the radio component to its normal state. SecRose's control flow continues at event `packetSent()`.

Chapter 6

Evaluation

6. Evaluation

This chapter evaluates the security provision and the performance of SecRose while it compares it with the existing proposals.

6.1 Evaluation of security provision

This subsection presents attacks on potential vulnerabilities of each security requirement and documented methods of conducting them. The operation and effectiveness of the provided countermeasures is then discussed and evaluated.

This subsection begins by discussing how the threat model is perceived in SecRose and how it relates to the requirements for confidentiality, authentication and freshness. Possible attacks are also discussed.

The provision for each of the requirements is analysed in subsection 6.1.2-6.1.4 while 6.1.5 discusses the additional security requirements. The correctness of the implementation of XXTEA and CMAC is also demonstrated briefly.

The subsection concludes with an evaluation of SecRose against other solutions.

6.1.1 The threat model again

This subsection relates the threat model, presented in the literature review, with SecRose's basic security requirements of confidentiality, authentication and integrity, freshness and availability. Vulnerabilities to each of the basic security requirements lead to a number of attacks documented in the literature but the attack nomenclature is inconsistent. In order to preserve consistency, this document refers to attacks by the name of the requirement in which they exploit vulnerability.

This subsection gives a short description of the attacks on the threat model, explains how they are perceived by SecRose's security requirements and relates them to the relevant requirement. The alternative attack names for known attacks in each sub-subsection are provided where applicable.

Attacks on confidentiality

Attacks on confidentiality, also known as eavesdropping attacks, lead to revelation of the plaintext input, usually after retrieval of the encryption key. Attacks on confidentiality that target the computational complexity of the encryption function are generally known as cryptanalytic attacks.

Resilience against cryptanalytic attacks is provided by the theoretical level of complexity, which is directly related with the length of the key, and by the mathematical perfection of the cipher. Every cryptanalytic attack is associated with some level of difficulty. The aim of the cipher is to provide an infeasible level of computational difficulty, which should be greater than the perceived computational ability of the attackers [3].

Known-ciphertext attacks

The attacker has access to the encrypted output of the cipher but the input, key and plaintext, are unknown. The attack aims to reveal the plaintexts, the key or partial information about the plaintexts. This attack is rarely used nowadays as modern ciphers are specifically designed against it, making it exceptionally difficult to conduct the attack.

Known-plaintext attacks

This is a widely used attack but its possibility on sensor networks depends on the kind of application that runs. The attacker must have knowledge of both the encrypted output and the plaintext input, while the key is the only unknown variable. The attacker then conducts an exhaustive search of all the keys in order to find the one that created this combination of input/output. The complexity of this attack is equal with the length of the key. The size of the key affects the number of required matching attempts and therefore the required time before the attack concludes. This attack is popularly known as the brute-force attack.

Chosen-plaintext attacks

The attacker can “query” the cipher with specific ciphertexts of their own choice and obtain a number of outputs that relate to the chosen inputs. If the cipher is not mathematically perfect, the attacker might then utilise some computation in order to obtain information about the key, reducing the theoretical complexity of the cipher. If the attack can be conducted with relatively low computation, then the attack is successful for the attacker and the cipher is considered insecure.

Related-key attacks

This attack requires the attacker to obtain two or more ciphertexts encrypted with different keys that are unknown to the attacker but share a known mathematical relationship. Similar to the Chosen-plaintext attack, the attacker might exploit weaknesses of insecure ciphers to reduce the theoretical complexity and reveal the used keys in a relatively easy way.

Semantic security

This attack requires the attacker to obtain enough ciphertexts that can be related with known or predicted plaintexts. After an initial observation period, the attacker can deduce plaintexts, or information about them, without knowing anything else than the resulting ciphertexts.

The attack is possible when there is a predictable relation between all given plaintexts and all resulting ciphertexts [29]. The use of a symmetric cipher normally enables this attack, unless the system introduces a non-constraint bit of information to the ciphertext.

Advantages of this attack include that it does not require any significant computational effort to be conducted and does not allow the security system to detect that it is under attack.

Semantic security is considered important to Public-Key Cryptography but there is no documentation regarding its importance in sensor networks or in other symmetric-key cryptosystems. The usefulness and feasibility of such attacks in sensor networks is highly dependent on the level of variability on the data that the sensor network reports.

Attacks on authentication and integrity

Attacks on the authentication and integrity of the system aim to forge malicious data or nodes that are then inserted into the network. If a system is vulnerable to these attacks, then a number of routing attacks is possible and under certain conditions, the cryptographic key might be revealed. The system is protected by the time complexity required for these attacks to happen. The time complexity property refers to the number of randomised attempts required before the attack succeeds. The computational complexity is insignificant if the attacker is equipped with a modern computer.

The resource-constrained nature of sensor networks allows a relatively small number of data that to be transferred before depleting the energy resources. In addition, data can only be transferred at a relatively slow rate. Protection that requires resources or time that exceeds the capabilities of WSNs is considered sufficient. Injection attacks are dangerous for any security mechanism and are particularly dangerous for the key management component of SecRose.

A successful packet injection or alteration attack allows the attacker to manipulate the routing protocol enabling a number of routing attacks. Currently documented attacks are; selective forwarding, sinkhole, black hole, wormhole and Sybil attacks [76]. In addition, the attacker might force illegitimate messages to be delivered to the base station.

An acknowledgement injection attack allows for acknowledgement spoofing; making a node appear healthy while it might have been destroyed or compromised [76]. In addition, due to the way SecRose's key management operates, such attack would lead to key de-synchronisation and thus communication failure.

An attacker who is capable of systematic packet injection has essentially managed to represent herself as a node injected in the network. Such an achievement enables the wormhole attack and practically renders the cryptographic functionality meaningless. In

addition, if the attacker can record all possible keys that allow packet injection, she might be able to work out the initial key.

Specifically for SecRose, packet injection attacks might lead to failure of subsequent communication. If an attacker manages to inject a *broadcast* packet then the receivers will change the key to a new value, which might be unknown to the base station. Therefore, the pair key will be de-synchronised and the receivers will be unable to receive further, legitimate, *broadcast* packets. The same problem is possible for *normal* or *long* packets but only if two or more packets are injected subsequently. Otherwise, the failover mechanism of the key management component can desynchronise the pair.

Attacks on freshness

These attacks involve recording of legitimate valid packets and replaying them later, at a time convenient to the attacker. The attacks aim to report falsified information to the base station or to fiddle with the routing information.

Apart from the routing attacks described in [76], there are no other documented attacks that exploit freshness vulnerabilities. However, the importance of preventing information replays is considered in every proposed data-transportation layer security mechanism [2, 21, 29-30].

A typical attack is described with an attacker who is able to replay packets with a powerful transmitter and has recorded the network's activity under "normal" conditions. When the sensor networks' sensing capabilities are truly needed, the attacker replays the "normal" pre-recorded packets with a strong transmitter, which is able to shadow the node's transmitters in order to prevent the network from reporting the real event.

There are two kinds of freshness. Weak freshness guarantees that messages are not replays but does not give any information on the time elapsed between transmission and reception of a packet. Strong freshness includes time information and it can confidently guarantee not only that a packet is not a replay but also that it is actually a recently transmitted packet.

6.1.2 Provision of confidentiality

Provision for known-plaintext attacks

Protection against a known-plaintext attack relies on the theoretical complexity provided by the cipher, which is directly related with the length of the key. The current agreed key length that would provide sufficient security is at least 128-bits [72]. This is also considered secure by references [4, 71]. SecRose utilises XXTEA to provide keys of that length. Longer keys would currently be an unnecessary waste of resources in a resource-deprived sensor network.

The 128-bits key length provides enough computational complexity to retain the secrecy of the encrypted data for the next 50-100 years [4, 72]. In addition, the cipher can operate in DES-style modes, allowing for longer keys to be used in the future if required, in similar fashion as in Triple-DES [129].

Provision for other cryptanalytic attacks

Other, more sophisticated, cryptanalytic attacks involve exploitation of fundamental cipher vulnerability in order to decrease the computational complexity of exhaustive search attacks. The mathematical integrity of the cryptographic cipher is responsible for protection against these attacks.

As a modern cipher, XXTEA is considered resilient against chosen-plaintext, known-ciphertext and related-key attacks. Even if these attacks are allowed by the rest of the SecRose system, the cipher does not have any known mathematical imperfections that would allow reduction of the computational complexity and thus are meaningless. Additional discussion on the security of the cipher follows.

Security of TEA

The security of a cipher against cryptanalytic attacks is a theoretical task, which cannot be tested practically. The security of TEA and its successors has been analysed thoroughly, since it was first published in 1995. Various problems were found with the first TEA and were promptly corrected. XTEA and XXTEA are currently considered secure. A summary of all the cryptanalytic work in XTEA and XXTEA is presented in reference [129].

All TEA variants demonstrate strong *diffusion* and *confusion* and they are surprisingly simple [129]. In addition, it was demonstrated that TEA resists various cryptanalytic attacks and differential cryptanalysis. The simplicity of the algorithm itself adds credibility to the cipher, since it is easy to analyse it mathematically.

Provision of semantic security

Semantic security is provided in SecRose by the frequent key changes introduced by the key management component. Even if the same ciphertext is input for two communications, a different key will be used and that change will be reflected in the resulting ciphertext.

However, semantic security is provided only if the previous communication was successful and an acknowledgement was received. An attacker who wishes to attack the semantic security of SecRose will initially have to block acknowledgements. This pre-requisite allows SecRose to detect that an attacker is present, depriving the attacker from the ability to remain undetected. Therefore, the attack is available to attackers but detectable and less significant.

Note that the above attack is not possible for *broadcast* packets, which always advance their counter after every transmission. SecRose can transmit up to 2^{32} *broadcast* packets without reuse of the same key. According to calculations, which will be discussed in 6.1.3, this amount of packets cannot be transmitted in typical sensor networks and therefore *broadcast* packets benefit from permanent and unconditional semantic security.

SecRose already keeps track of how many acknowledgements are in waiting, and this information could be passed to applications that can utilise it, via the packet's data struct. However, the facility to report on missing acknowledgements is not implemented in the proof-of-concept demonstration.

SecRose has chosen to provide semantic security via a mechanism that differs from the other proposals, which utilise an Initialisation Vector (IV). This choice is further discussed in 6.1.6.

Time complexity attack on TEA

A 2010 publication, in a non-scientific, non-peer-reviewed medium, claims to have reduced the time complexity of XXTEA under certain conditions [137]. The attack is a chosen-

plaintext attack requiring 2^{35} queries when applied to 6 XXTEA cycles. The time complexity increases to 2^{59} chosen plaintexts for 32 XXTEA cycles.

Our thesis regarding this claim is that SecRose is not vulnerable to these attacks, since the time complexity is always greater than the time complexity introduced by the size of a packet's MAC. As will be discussed in 6.1.2, it is impossible for an attacker to make 2^{35} queries on a typical sensor network.

Prof. S.J. Shepherd, the author of [129] and unofficial maintainer of TEA, and has dismissed the validity of this attack in a personal communication. Prof. Shepherd has repeated the views expressed on his paper; XXTEA is safe for all practical purposes.

Relation with requirements

The requirement of 128-bit long security has been met by utilising a secure cipher, which natively supports this key length.

Correctness of the implementation

Due to the asymptotic nature of exhaustive searches, the operation of the cipher cannot be tested conclusively. However, some experiments have been conducted to test the cipher. The experiment is set up so that node 0 is the attacker and node 1 is the node under attack. Node 0 knows part of the key and attempts to find the rest of the key by sequentially searching all combinations.

For the first experiment, the following key is loaded to node 0 (in hex format):

```
32A3D709 2A83F848 D2F6F4B3 AB211500
```

This is a randomly selected value except that the last 8 bits are set to zero. Node 0 will pretend to be the base station and attempt to send *broadcast* packets to node 1. The node increases its key monotonically. Node 1 is loaded with the following key:

```
32A3D709 2A83F848 D2F6F4B3 AB2115FF
```

This is the same value except that the last 8 bits are set to one. Node 1 should reject the first 255 packets and accept the 256th. Otherwise, the cipher has produced the same ciphertext from a combination of different plaintext and key. Discovery of the key will be tested by determining when a matching MAC is found and the packet is accepted.

The test was run on a modified code that would send the packets and (a) report when a MAC is found, (b) report the number of sent packets, (c) report the used key and (d) exit the simulation. The simulator outputted the following:

```

Elias@Natally /opt/port/apps/test
$ ./build/pc/main.exe -t=300000 2
i. Packet count: 256. MAC IS UALID.
i. Recv MAC: 19 33 87 228
i. Calc MAC: 19 33 87 228
i. Used key: 32a3d709 2a83f848 d2f6f4b3 ab2115ff
Elias@Natally /opt/port/apps/test
$
  
```

Evidently, all the objectives of the experiment were satisfied but further experiments were run to validate that the findings are not a product of chance. The experiment was repeated by using the non-randomly set bits at the ends of the remaining 32-bit chunks of the key:

Experiment	Node	Key
2	0	32A3D700 2A83F848 D2F6F4B3 AB211509
	1	32A3D7FF 2A83F848 D2F6F4B3 AB211509
3	0	32A3D709 2A83F800 D2F6F4B3 AB211548
	1	32A3D709 2A83F8FF D2F6F4B3 AB211548
4	0	32A3D709 2A83F848 D2F6F400 AB2115B3
	1	32A3D709 2A83F848 D2F6F4FF AB2115B3

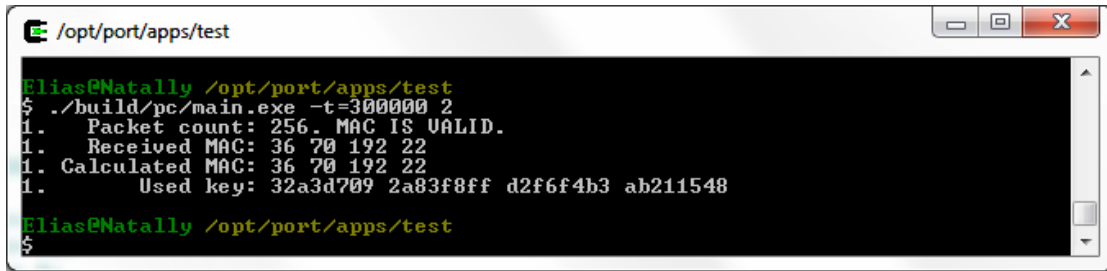
Node 0 was set to increase the relevant part of the key monotonically, for example, in attempt 2 it increased 2A83F800 to 2A83F801, then 2A83F802 and so on. After running the experiments, the simulator produced the following output:

Experiment 2

```

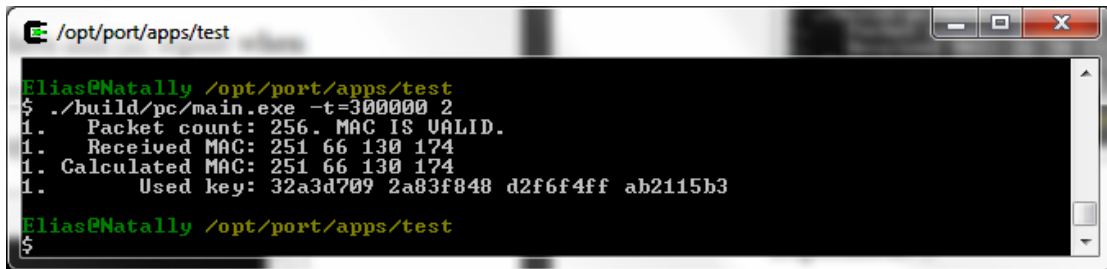
Elias@Natally /opt/port/apps/test
$ ./build/pc/main.exe -t=300000 2
i. Packet count: 256. MAC IS UALID.
i. Received MAC: 63 151 128 50
i. Calculated MAC: 63 151 128 50
i. Used key: 32a3d7ff 2a83f848 d2f6f4b3 ab211509
Elias@Natally /opt/port/apps/test
$
  
```

Experiment 3



```
Elias@Natally /opt/port/apps/test
$ ./build/pc/main.exe -t=300000 2
i. Packet count: 256. MAC IS UALID.
i. Received MAC: 36 70 192 22
i. Calculated MAC: 36 70 192 22
i. Used key: 32a3d709 2a83f8ff d2f6f4b3 ab211548
Elias@Natally /opt/port/apps/test
$
```

Experiment 4



```
Elias@Natally /opt/port/apps/test
$ ./build/pc/main.exe -t=300000 2
i. Packet count: 256. MAC IS UALID.
i. Received MAC: 251 66 130 174
i. Calculated MAC: 251 66 130 174
i. Used key: 32a3d709 2a83f848 d2f6f4ff ab2115b3
Elias@Natally /opt/port/apps/test
$
```

It is observed that Node 0 was always able to find the expected key after exactly 256 tries and remained completely consistent with the theoretical expectations.

6.1.3 Provision of authentication and integrity

Conditions for packet injection attacks

In order to inject *normal*, *broadcast* or *long* packets on the network, an attacker needs to guess a valid MAC, associated with the packet content. Attackers are able to attempt that by transmitting a packet with a seemingly random MAC and hope that the MAC was validated by chance. The theoretical probability of this happening is discussed.

The specified SecRose MAC length is 4 bytes and there are therefore 2^{32} possible MACs. Therefore, any MAC, even those that are randomly generated, would have a 1 in 2^{32} chance of matching a MAC that is generated by actual input and proper procedure. When this happens, a MAC collision is found. On average, a collision is found after $n = 2^{31}$ packets are exchanged.

SecRose can protect a network as long as this number of packets is not received. This level of complexity is selected in accordance with the requirements for the authentication component. The requirements state that the mechanism has to provide complex enough authentication to

render such attacks useless by forcing them to deplete energy resources in order to be successful. As will be discussed, reception of 2^{31} packets would require greater resources than what is available on sensor nodes, therefore making the node to cease to operate due to lack of available energy.

The utilisation of available energy resources needs to be calculated in practical terms in order to determine if the system can satisfy the security requirements. Assuming total energy E is required to carry out an attack on a node with capacity C , the system is secure as long as $E > C$. Failing to meet this condition means that the node's energy resource is drained out before the attack concludes.

Assuming packet length L , energy e is required to transfer one bit and total packets n need to be transferred, then the required total energy E to receive the packets is $E = L * e * n$. Therefore, the system is secure as long as $[E = L * e * n] > C$.

It has already been discussed that $n = 2^{31}$. The real-terms value of the other variables can also be found in the literature or drawn by SecRose's design:

- SecRose's minimum packet size $L = 152b^{24}$
- the required reception energy $e = 2028nJ/b$ (nanojoules per bit) [69]
- the node is provided²⁵ with capacity $C = 30,000J$ (joules) [138]

Under these conditions, $E = L * e * n = 152 * (2028 * 10^{-9}) * (2^{31}) = 661974J$, which is about 22 times than C and therefore the system is considered secure. Further calculations indicate that the energy resources will be depleted after about $2^{26.5}$ packets are transmitted. The above calculations do not account for the computational energy that is additionally required to validate the packets.

In addition to energy constraints, there are time constraints as well. For example, a MICA2 node communicates at 38.4Kbps and it can therefore receive about 253 packets per second. Without accounting for medium access delays, the 2^{31} packets required before a collision is found would take 98 days.

²⁴ As defined in SecRose's design, the minimum length of broadcast packet with 0 data payload is 7 bytes. There is also a 12-byte preamble defined in TinyOS for every packet. A total of $19B = 152b$.

²⁵ There is little published information on the exact capacity of commercial batteries. The 30,000 Joule figure for two AA batteries is a very generous estimation derived by a number of sources in addition to the referred article.

Therefore, even if unlimited power supply is assumed, the 4-Byte MAC can still provide some protection. However, the current configuration of SecRose, including the length of the MAC, is optimised for nodes with a limited power source. If a network enjoys unlimited energy resources then SecRose can be configured to use a 5-Byte MAC, which would increase time complexity by a factor of 256, resulting in time complexity of 68 years before a packet is injected.

A diagram of how a packet injection attack might be conducted is given in Figure 20. The attack begins with the attacker sending a packet containing a MAC calculated using the key K_1 . As this will probably fail, the attacker continues sending packets with MACs calculated with various keys, until a match is found with K_n . In this case, the receiver will update its active counter. Note that even MACs that are selected randomly could in fact have been calculated using a key. Therefore, the diagram notates all MAC as to have been calculated with some hypothetical key.

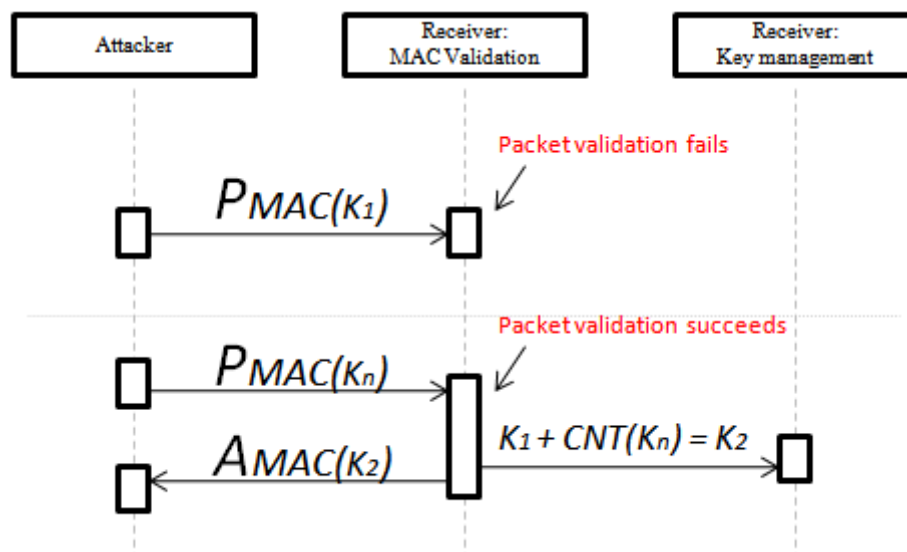


Figure 20: sequence diagram of packet injection attacks.

The attacker keeps sending invalid packets until packet encrypted with K_n is sent.

Conditions for acknowledgement injection attacks

SecRose's authenticated acknowledgements are protected by theoretical confidence guarantees and practical limitations. Acknowledgements contain two authenticated bytes and therefore there are 2^{16} distinct values. Forged acknowledgements can cause key desynchronisation and prevent subsequent communications. Since the 2^{16} complexity is not an

asymptotic value, SecRose does not rely on the complexity and it sets a limit to how many invalid acknowledgements it can receive.

An attacker would be able to transmit all distinct 2^{16} acknowledgements without consuming significant energy. However, a sender can only receive as many invalid acknowledgements as many packets it has sent and waits for an acknowledgement. Therefore, the sender can receive a theoretical maximum of 10 acknowledgements, as this is the maximum size of the acknowledgement table²⁶.

This limitation prevents the attacker from managing key de-synchronisation using brute force but a chance of achieving the attack remains. For a 2^{16} complexity and a sender that is constantly expecting 10 acknowledgements, the attacker has a 10 in 2^{16} chance of success when she sends random acknowledgement values after a communication. Therefore, de-synchronisation will occur after 3276.8 forged acknowledgements.

In practice, the chance is even smaller because the acknowledgement table is rarely expected to reach its maximum size of 10. The actual size depends on various factors but it can be calculated that as long as the senders do not send more than one packet every second then the size of the table would remain 1.

Conditions for node injection by packet injection

Most of the documented routing attacks that require what is described as “node injection” could be conducted by just injecting one authorised packet on the network. The actual feasibility depends on the routing protocol and its resiliency. Therefore, SecRose’s level of resiliency to node injection attacks equals the 1 in 2^{32} chance of managing a packet injection.

Conditions for key de-synchronisation by sequential packet injection

SecRose is susceptible to key de-synchronisation and communication disruption should the attacker manage to inject two packets with the same destination on the network. The worst-case scenario is when these packets are *broadcast* packets, where the whole network will be affected.

²⁶ Discussed in 4.1.4.

Such an attack can happen with half the probability of injecting one packet, since any packet has the same probability, two packets are required and there is hypothetically no restriction on how many attempts are made. Therefore, an attacker has, on average, 1 in $2^{32} + 2^{32} = 2^{33}$ chance of succeeding on this attack. This value is asymptotic enough to claim that the level of provided security is acceptable.

Such attack would require double the time or double the energy to succeed and upon completion would only affect the packet type used to launch it.

Conditions for key revelation

An attacker might conduct a coordinated attack, which aims to find and document all possible MAC collisions and the hypothetical keys that created them. The chances of succeeding are unrealistically low but this attack is discussed for completeness.

To conduct this attack, the attacker has to know when a packet was successfully injected. Therefore, the attack cannot be conducted with a burst of *broadcast* packets transmitted without time interval. The attack has to allow nodes to respond and send an acknowledgement. According to the TinyOS specification, nodes respond with a medium access delay, which is always greater than 100 msec. Even if the actual transmission did not take any time at all, the attacker would only be able to send 10 packets per second. At this rate, it would take 6.8 years to transmit 2^{31} packets and manage to inject one.

Even if this is not a problem, the attack cannot conclude until a great many packets are injected. Since a MAC is 32 bits and the minimum acknowledged packet is 72 bits, there are $2^{72}/2^{32} = 10^{12}$ possible collisions. The attacker has to collect about half of them to reach the average number of collected collisions before the initial key can be revealed. Under these conditions, it is assumed that SecRose is not vulnerable to this attack.

Relation with requirements

SecRose meets the authentication and integrity requirements. Indirect authentication was not needed as all packet fields are directly authenticated via the MAC.

Correctness of the implementation

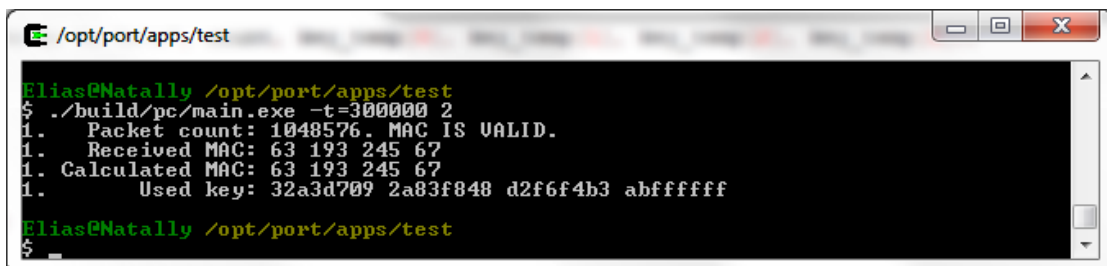
The protection against packet injection such attacks relies on the Authentication component, which in turn relies on the CMAC function and the encryption function. If the encryption function does not evenly diffuse key and plaintext then the CMAC function will not provide a homogenous output and thus the probability of an attacker finding a valid MAC by testing random values will be greater than the theoretical 1 in 2^{32} that SecRose should provide.

The full test of the MAC collision probability requires exchange of at least 2^{31} packets. Due to the limitations of medium access control, the full range of this experiment cannot be run by exchanging packets, as it would require 25 days to conclude. A number of smaller experiments were run to test the operation of both the CMAC and XXTEA simultaneously. The used keys were:

Node 0: 32A3D709 2A83F848 D2F6F4B3 AB000000

Node 1: 32A3D709 2A83F848 D2F6F4B3 ABffffff

Using these keys, the experiment should exchange 2^{20} packets. The experiment was run successfully and the expected output was produced. The output was:

A terminal window titled "/opt/port/apps/test" showing the execution of a program. The prompt is "Elias@Natally /opt/port/apps/test". The command executed is "./build/pc/main.exe -t=300000 2". The output is:

```
i. Packet count: 1048576. MAC IS UALID.  
i. Received MAC: 63 193 245 67  
i. Calculated MAC: 63 193 245 67  
i. Used key: 32a3d709 2a83f848 d2f6f4b3 abffffff
```

The prompt returns to "Elias@Natally /opt/port/apps/test" and the shell prompt "\$" is visible at the bottom.

6.1.4 Provision of freshness

Operation of the freshness provision

Freshness is provided to SecRose via the key management component. Each time a packet is successfully transmitted, the cryptographic state changes and attackers cannot learn any information about the new state, unless some other attack was successful.

The new state designates a new “era” and the first packet that is exchanged in this era cannot be a replay of a packet exchanged in a previous era. If that packet was received in the past, then the state would have changed back then and the current era would not be operative under this state.

The key change and state transition is illustrated in Figure 21. The attack begins with the attacker recoding a valid packet. The system progresses to a new era as the new key K_2 is stored and will be used for the next communication. Later, the attacker replays the packet. This packet is not valid anymore since it was generated with K_1 but now the receiver has advanced to K_2 .

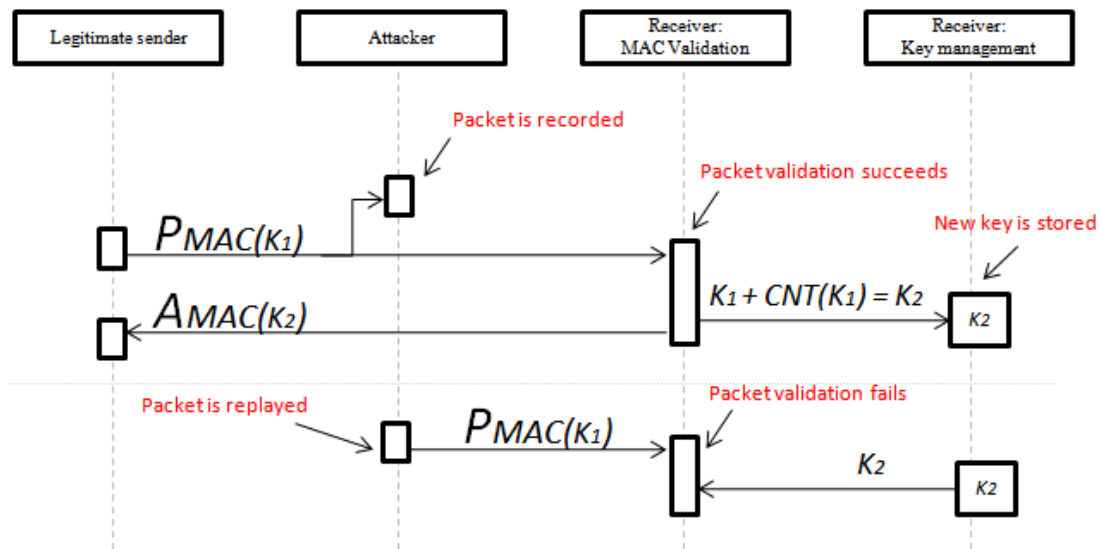


Figure 21: key changes and prevention of packet replays

Conditions and levels of provided freshness

The freshness mechanism provides weak freshness upon the condition that the last packet was successfully exchanged and the key was properly advanced. Therefore, various packets might enjoy different levels of freshness. This provision is in accordance with the requirements. The advantages and disadvantages of the SecRose solution against other proposals will be further evaluated 6.1.8.

Definitely fresh

The first packet exchanged after a successful communication is weakly fresh, with probability of error equal to the probability of packet injection²⁷, which equals 1 in 2^{32} . This is equal to the probability of packet injection, since a replayed packet in these cases is no different from a randomly generated packet in a packet injection attempt.

Broadcast packets are always definitely fresh, since they do not trigger acknowledgements and the broadcast key is always advanced. Effectively, each *broadcast* packet is the only packet that can be transmitted on a given era.

Unknown status

Subsequent packets transmitted in the same era, prior to key advancement have an undetermined freshness status. SecRose might loosely guarantee that these packets belong to this era but the system cannot tell if these packets are replays from the same era. Depending on the application, this might or may not introduce important consequences.

Regarding strong freshness

Inclusion of time information on a packet in order to provide strong freshness is not a requirement of SecRose and it is deemed an unnecessary resource-consuming feature since it is not required by every sensor network application²⁸. Applications that require strong freshness are able to include timing information on their packet and rely on the rest of the security properties of SecRose for authenticated and confident delivery of this information.

²⁷ As discussed in 6.1.3.

²⁸ As discussed in 2.2.1.

Relation with requirements

SecRose meets the freshness requirements as expected. Packets can be determined as confidently fresh or undetermined. In addition, packets can be associated with an era.

6.1.5 Additional security provision

Destructive attacks

Protection against destructive DoS attacks is out of the scope of SecRose and is therefore not provided. The following is discussed for completeness. These attacks have been discussed and addressed by other security research groups [109-110]. The typical sensor networks, which SecRose aims to operate at, are probably not suited to respond to such attacks and the SecRose mechanism is helpless in detecting or avoiding them.

Sensor networks are particularly vulnerable to radio jamming Denial of Service (DoS) attacks. An attacker can use a powerful transmitter, which is able to transmit at much higher energies than a sensor node. Such transmitter is neither difficult to make nor expensive.

In addition to radio attacks, the SecRose mechanism cannot detect or protect a sensor network from node destruction attacks.

Asymmetric DoS attacks

The requirements state that SecRose should not enable intelligent DoS attacks that an attacker can execute with little effort but would cause an asymmetrically great response from the sensor network, forcing it to consume high amounts of energy. To the best of our knowledge, this requirement has been met.

Transmission of a small, randomly generated, invalid packet does cause the receiver to undergo the effort of MAC validation twice, but this response is essential to the correct operation of the failover mechanism and it is not a greatly asymmetric effort. Nevertheless, the completion of double validation consumes less time than the time required transmitting the malicious packet²⁹.

²⁹ Further discussed and supported in 6.2.3.

The system design appears to have no loops. The failover mechanism is not repeating any packets and there are no acknowledgements to the acknowledgement. If such features were present, they might have allowed a malformed packet to cause an infinite loop on the system.

Packets with incorrect length do not have an adverse effect on the energy consumed by the receiver either. Firstly, input validation is a task that should be done at code level. Secondly, the flag-based packet management system does not allow packets of greater than 64-bytes payload to be received. Any packet with a malicious 64-byte data payload will be perceived as a *broadcast* packet with 1-byte payload. Similarly, packets with 129-bytes payload are perceived as *long* packets with 1-byte payload. Finally, packets with 193 bytes payload cause even less energy damage as they are perceived as acknowledgement packets with 0-payload and their processing does not involve any encryption operation.

Points of failure

SecRose's design does not designate any node to be of additional importance to the sensor network and it therefore does not introduce any single points of hardware failure. Note that the base station of any sensor network is inherently a single point of failure [81] and this status is not affected by SecRose.

In addition, SecRose does not appear to have any particular weakness in the design. It might be true that acknowledgement forging is easier than packet injection, but this attack is also the easiest to prevent as well.

Node compromise

Hardware security has been discussed in 2.3.2 and 2.3.5. The subsections conclude that SecRose should not aim to provide resiliency against node compromise attacks. Such research is primarily directed to electronic microprocessors and is not a research problem of our area. Therefore, SecRose should not be considered as a mechanism that provides protection against attack to hardware security.

However, SecRose includes a key management mechanism, which demonstrates potential to prevent node compromise attacks in the future. Currently, each node keeps the initial key in order to be able to initiate creation of new pairs. An attacker who gains access to the initial

key can initiate communication with the whole network and thus the system is vulnerable to node compromise.

A future version of SecRose, or an application built for SecRose, may implement a mechanism that would not require a pre-loaded initial key to be present on memory at all times. The existing control packet infrastructure³⁰ may support the introduction of this feature. Combined with the existing variable cryptographic state, the feature would create functionality that is similar to other key distributions schemes [23].

Security of the changing keys

It is stressed that the key management mechanism of SecRose does not provide additional cryptographic complexity, other than 128-bit key length provided by the cipher. Since the derived keys are mathematically related with their predecessors, ability to reveal one key grants access to all derived keys as well. For this reason, the computational complexity is not based in the changing keys feature.

Guaranteed deliveries

SecRose offers an authenticated guarantee of packet delivery via the authenticated acknowledgements. The guarantee is provided to the sender with 10 in 2^{16} probability of error. However, lack of acknowledgement does not necessarily mean lack of delivery. An acknowledgement might have simply been lost while in transit. Appropriate utilisation of this feature, e.g. retransmission of seemingly lost packets, is the responsibility of the application. This behaviour is backwards compatible with TinyOS.

Security of *broadcast* packets

Secure broadcast communication is very difficult to achieve efficiently in sensor networks [29]. Although SecRose provides secure broadcast packets, they are vulnerable to a DoS attack that might render them inoperable. The attack involves blocking some of the *broadcast* packets, which causes key de-synchronisation, and breakdown of further *broadcast* packets. Unfortunately, SecRose cannot guarantee the reliability of broadcast communication without compromising efficiency.

³⁰ Discussed in 4.1.1.

On the other hand, the provided level of security in *broadcast* packets is higher than in both *normal* and *long* packets. Since there are no acknowledgements, the base station will always advance the key after each transmission. This method guarantees unconditional semantic security, a sub-property of confidentiality, and weak freshness as it is not subject to the conditions introduced by the blocking of acknowledgements.

As a conclusion, it is guaranteed that the nodes will receive a *broadcast* packet through secure transmission, but the base station is not guaranteed that its broadcast attempt will reach the nodes. The topic will be further discussed and compared with other proposals in the next subsection.

This behaviour does not violate the availability requirements of SecRose and is consistent with other broadcast communication methods found in traditional computer networks, for example in the ARP [139] and ICMP [140] protocols. In these and many other cases, broadcast communication is a “cheap” mechanism of unreliably addressing multiple network nodes.

Relation with requirements

SecRose unconditionally meets the additional security requirements.

6.1.6 Evaluation against other solutions

Candidate alternatives

This subsection evaluates and compares the security of SecRose against the similar TinySec, MiniSec and SenSec. The 802.15.4 (ZigBee) [10, 105] and ContikiSec [93], are also discussed despite the fact that they are intended for high-end sensor nodes and therefore dissimilar to SecRose. The more recent proposal, FlexiSec [94] is critically evaluated as well. The SPINS [29] proposal was not accounted for since it seems partially complete and abandoned by its creators.

Confidentiality: ciphers and key lengths

As discussed, the confidentiality provision is directly related with the length of the key and the mathematical integrity of the cryptographic cipher. The systems under evaluation use either XXTEA or SkipJack. Both ciphers are considered secure if used properly [129, 141].

In particular, the best-known cryptanalysis of SkipJack is only computationally beneficial if the cipher is used with 31 rounds. However, the design of SkipJack recommends a minimum of 32 rounds to be used and therefore the imperfection does not affect real-world proper implementations. It is certain that TinySec uses SkipJack with 32 rounds. However, it is only assumed that the same is true for MiniSec and SenSec as well. On the other hand, there is no known and peer-reviewed cryptanalysis on XXTEA. The only known possible vulnerability is yet to be validated.

Therefore, the essential differences in regards with provision of confidentiality lie in the selected key length. SecRose uses a 128-bit key, which is natively supported by the cipher. TinySec and MiniSec use SkipJack in its 80-bit native mode while SenSec uses key whitening to operate SkipJack with a non-native key length, which is variable between 80-144 bits.

When whitening is used, the effective complexity is $2^{n+(m/p)}$, where n is the key length, m is the block size and p is the number of known plaintexts [4]. Under these conditions, an attacker who acquires one plaintext in order to conduct a known-plaintext attack would face an effective computational complexity of 2^{144} . However, this complexity is drastically reduced if the attacker can obtain more than one plaintexts. For example, if 10 plaintexts are known, then SenSec can only enjoy a complexity of $2^{86.4}$. The effective complexity reduction is less drastic after this point.

In addition to the complexity problem, whitening, and especially its use with SkipJack is not a well-documented technique and therefore it is open to potential cryptanalytic attacks. If proven secure, it can also be applied to SecRose and increase its complexity to a variable 128-160 bits.

SecRose is the only protocol that provides an unconditional solution that adequately satisfies modern complexity requirements. For these reasons, SecRose provides better confidentiality than the alternative proposals.

Confidentiality: semantic security

The alternative proposals utilise an Initialisation Vector (IV) to achieve semantic security. SecRose relies on its key management mechanism for the same task. These are significantly different approaches with different advantages and drawbacks.

The IV approach appears to have the advantage of unconditional operation; it will always provide a level of semantic security. However, IVs offer their security on condition that the IV value will not be reused. All the proposals arrange their IVs so that they appear long enough but in reality, the truly variable parts are much shorter than presented. For example, TinySec uses an IV that appears to be 32-bits long but the authors admit that for a particular pair, IV reuse will happen after 2^{16} packets are exchanged. Similarly, SenSec will reuse its IV after 2^{24} packets. Unfortunately, the authors of MiniSec do not clearly state their IV variability.

The proposals do not seem to have permanently tackled the problem of IV reuse and therefore their security is conditional. In addition, the IVs introduce a potential for vulnerabilities [111]. Since the IV is both transmitted in the clear and a part of the ciphertext input, it is essentially input data leaked to attackers, a piece of known-plaintext that is always available.

On the other hand, SecRose provides a 32-bit variable counter as part of the key. The counter's length is enough to prevent reuse but it only operates properly on the condition that packets and their acknowledgements are received. However, if attackers block the acknowledgements consistently, they reveal their presence to SecRose and therefore the attack is less serious as it loses the advantage of remaining passive.

Other proposals can also detect when IV reuse will happen, since it can be tracked on their system configuration. Therefore, all proposals offer semantic security, which is based on a detectable condition.

The true advantage of SecRose's approach is that the seriousness of the attack is reduced and that it does not introduce additional radio energy. For these reasons, SecRose provides a better overall performance, as far as semantic security is concerned.

Authentication

All proposals use a 4-Byte MAC, which provides the same probability of collision and therefore the same probability of packet and node injection. Differences exist in the way the MAC is calculated since SecRose, TinySec and MiniSec use two-pass CMAC authentication while SenSec uses a one-pass OCB authentication, which supposedly uses less energy but has no effect in the security of the MAC.

Both modes appear to be secure and therefore the security of the authentication for all proposals is considered the same.

Freshness

There are different levels of freshness provided by different proposals. MiniSec provides weak freshness and guarantees that a packet is not a replay of a previously transmitted packet. However, MiniSec cannot determine the time elapsed between transmission and reception in any way. TinySec does not provide freshness, and the authors of SenSec do not even mention freshness in their paper.

SecRose provides conditional weak freshness, which always gives some timing information. As explained, the key change mechanism clusters time in eras and SecRose can be confident about the freshness of the first packet transmitted in every era. For this first packet, SecRose

does not only guarantee its freshness, but can also guarantee that it could not have been originated at a previous era. It therefore provides some information on when the packet was transmitted. However, SecRose cannot guarantee that subsequent packets are not replays but it can still determine the era in which they were transmitted.

While it is clear that for the first packet of an era SecRose is better than any other proposal, subsequent packets are problematic. MiniSec also uses an epoch-based system, which can guarantee the freshness of up to 256 packets on each epoch. Until that point, MiniSec performs better than SecRose and then it performs similarly.

However, if attackers wish to block the freshness guarantee of SecRose, they cannot remain passive. The block of acknowledgements is a detectable situation. Thus, MiniSec offers a higher level of freshness only when a detectable attack is conducted. On the other hand, if the attacker wishes to remain passive, both protocols offer the same level of freshness.

Each solution has its own advantages and both might be of equal importance but SecRose has opted for the conditional solution since it introduces zero energy overhead and potential Attacks on freshness generate a detectable condition.

Secret state versus open state

As evident by the evaluation of freshness and semantic security, SecRose has opted for a secret-state type of mechanism, which has certain advantages and drawbacks. The primary advantage of a secret state is that it leaves less room for potential attacks while the drawback is that some security features are conditional. However, this problem is less important since attacks can be detected.

All proposals offer a fair level of semantic security and freshness, and they all operate conditionally. The essential difference between SecRose and the other proposals is that the conditions for correct operation differ. SecRose maintains its security as long as the attacker wishes to remain passive. Other proposals maintain their security for a number of packets.

Additional advantages of the SecRose approach is that it leaks less information to attackers while it consumes less energy and always allows for detection of attacks. Future applications

will be able to make the most of SecRose, fully accommodate the detection capabilities and benefit from a full range of security features while consuming less energy.

SecRose constitutes an alternative solution, providing the benefit of choice to the field and therefore making an important contribution. Security features that are currently “conditional” are due to SecRose’s choice to provide a closed cryptographic state and offer the associated different advantages.

Key management

SecRose provides a limited key management mechanism in order to provide for its own functionality. The mechanism lacks some of the functionality provided by complete key management solutions but makes SecRose comprehensive and allows it to be evaluated as such.

In addition, the key management mechanism of SecRose demonstrates great potential for future versions to update and achieve further increases in the security and efficiency of SecRose.

Other proposals, like TinySec rely on external key management and are therefore functionally incomplete.

Routing

SecRose is the only proposal to provide authenticated acknowledgements. The provision introduces significant amount of radio energy but it is essential for both the security and integrity of the mechanism. The authenticated acknowledgements enable SecRose to secure all messages, including the routing messages and therefore allow it to provide a highly elevated level of security.

SecRose’s requirement is to secure all exchanged packets, including acknowledgements and therefore including all routing messages. SecRose enables sensor network developers to choose from any of the available routing protocols, which in turn could allow utilisation of energy-efficient routing that suits their network.

All other proposals are vulnerable to routing attacks that exploit vulnerabilities in the acknowledgement. As described in [76], these attacks are all kinds of selective forwarding attacks; nodes that may or may not forward a packet while pretending that they did. Most of the existing routing protocols rely on the acknowledgement mechanism and are therefore are vulnerable to selective forwarding. Sensor networks that use any of the following routing protocols are vulnerable; TinyOS beaconing [64], Directed Diffusion [18], geographic routing protocols like GPSR [17], cost-efficient protocols like [142], clustering protocols [89, 143] and rumour routing [88].

The only protocols that are resilient to these attacks require special capabilities from the sensor node. For example, SIGF [106], which natively uses authenticated acknowledgements, requires location awareness while topology maintenance routing protocols, SPAN [25], GAF [87], CEC [144] and AFECA [145], are not intended for typical sensor networks.

While it is true that other proposals increase the security and authenticity of outgoing messages, their failure to protect the acknowledgements is detrimental to the security of all typical networks. No sensor network that utilises typical, low power, low capability and cheap nodes, like an out-of-the-box MICA2 node, is secure against routing attacks, unless it uses SecRose.

Delivery guarantee

The packet delivery guarantee offered by SecRose is a unique feature provided by the authenticated acknowledgement mechanism. The feature makes selective DoS attacks on the radio medium easier to detect. There is currently no other proposal offering this additional functionality.

In fact, every other proposal is vulnerable to attacks in this area. All proposals use a link-layer acknowledgement; the first forwarding node informs the sender that their packet is received. This acknowledgement is not authenticated; it is simply a stream of four specific bytes. No proposal provides any way of confidently determining what happened next and if the packet finally arrived to the destination.

Other proposals are vulnerable to acknowledgement injection [76], selective forwarding and sinkhole attacks [76]. These attacks are not detectable but can cause disruption or total communication breakdown, depending on the routing protocol used.

SecRose can detect such attacks and most importantly is not vulnerable to communication breakdown under any circumstance due to this problem. However, SecRose only utilises this feature for self-synchronisation. Actual exploitation of this feature and implementation of it as a complete transmission control mechanism, as the one found in TCP, is left to the application.

Security of broadcast communication

SecRose provides a special *broadcast* packet, which benefits from a higher level of confidentiality, authentication and freshness than the level offered by *normal* and *long* packets. On the other hand, this packet does not offer availability.

Other proposals offer broadcast communication in the same way, and the same level of security, as with any other packet type. They are therefore not subject to the availability problems of SecRose but they are subject to attacks on confidentiality and freshness.

Neither SecRose, nor the other proposals were designed to accommodate the property of availability and it is not in their requirements. The attack was discovered on SecRose because it was the only proposal evaluated against availability, despite the fact that it was not designed to provide the property. Whether availability attacks exist in other mechanism is unknown.

On that basis, a comparison would only be fair if vulnerabilities on availability are excluded. Under this assumption, SecRose provides a much higher level of confidentiality, semantic security and freshness than the other proposals.

As with other packet types, SecRose offers a different approach, which is subject to different conditions.

Summary for TinySec, MiniSec and SenSec

The following table presents a summary of the security properties of SecRose, TinySec, MiniSec and SenSec. The table is compiled by the information discussed up to now in this

subsection. Note that authentication complexity is not listed since it is the same for all proposals.

	Effective key length	Semantic security	Freshness	Key management	Authenticated Acknowledgements	State
TinySec	80	Limited at 2^{16} packets	Not provided	Not provided	Not provided	Open
MiniSec	80	Limited to 2^8 packets/epoch	Weak freshness	Not provided	Not provided	Partly secret
SenSec	Theoretical: 80-144 Actual: $\cong 85$	Limited at 2^{24} packets	Not provided	Not provided	Not provided	Open
SecRose	128	Unlimited, conditional	Weak freshness	Yes, limited	Yes	Secret

Table 3: comparative summary of the provided security for each mechanism

The table illustrates how SecRose provides equal or better security than the other proposals. The effective key length of SecRose remains consistent at 128-bits and while MiniSec might provide a better theoretical limit, it is not proven that this is either necessary or truly more secure.

The semantic security of SecRose is not limited by reachable packet counts but it is conditional to whether the attacker wishes to remain passive or not.

A similar condition applies for the freshness requirement when comparing SecRose with MiniSec. On the other hand, TinySec and SenSec do not provide freshness at all.

SecRose provides a basic key management with potential for future radical improvements that could capitalise on the energy savings of other areas. Other proposals do not provide key management at all.

Finally, SecRose keeps its security state secret and avoids potential, documented, risks associated with revealing part of the security state of secret key cryptography systems.

Comparison with FlexiSec

Reference [94], published in June 2009, describes the FlexiSec configurable link-layer security architecture. This is a recent work and thus it is evaluated separately.

FlexiSec recognises the importance of 128-bit computational complexity and the authors have evaluated XXTEA and AES as possible ciphers. The final solution is using AES at a speed optimised or a size optimised mode. This level of complexity is equal with SecRose. However, the SecRose evaluation has shown that AES uses more energy than XXTEA.

For authentication, the authors propose the use of a MAC with a variable size between 4 and 8 bytes. Our evaluation has proven that it is unnecessary to use MACs that are longer than 4 bytes, unless the network benefits from an unlimited power source. In this case, SecRose is ready to use a 5-byte MAC, which would be sufficient for all realistic applications. To provide freshness, the authors use a technique similar to the technique used by MiniSec. FlexiSec does not provide authenticated acknowledgements or any form of key management either.

It appears that the contribution of FlexiSec is on the provided flexibility only and that the mechanism does not provide any new features or solves any existing problems.

Comparison with 802.15.14 and ContikiSec

The IEEE 802.15.14 (ZigBee) [10, 105] standard is a generic, secure, communication protocol for small devices. The ContikiSec [93] is proposed as the security mechanism for the Contiki OS [57]. They are both suitable for sensor networks consisting of nodes with high-end capabilities, like the Imote2 [47].

SecRose's assumption was that it operates in low-end devices like the MICA2 node and it is optimised for the requirements of such devices. A comparison of SecRose and 802.15.4 is presented for completeness only.

The Imote2 benefits from features like a scalable processor capable of 416MHz and 250Kbps communication rate. SecRose was designed for nodes like the MICA2 node, which operates at 4MHz and can communicate at 38.4Kbps. The Imote2 is more than ten times more capable than the MICA2 node and, obviously, there are similar differences in pricing and energy consumption.

ContikiSec does not provide freshness, key management or authenticated acknowledgements. In addition, the mechanism operates via a random IV and thus it is considered to have an open

cryptographic state. For this reasons, ContikiSec is inferior to SecRose and will not be discussed further.

On the other hand, the security of SecRose is directly comparable with 802.15.14 especially if the protocols are used in low-end devices. The following table compares the security of SecRose, 802.15.14 and ContikiSec:

	SecRose	802.15.14	ContikiSec
Key in bits (cipher)	128 (XXTEA)	128 (AES)	128 (AES)
MAC length (Byte)	4	4/8/16	4
Semantic security	Unlimited, conditional	Unlimited, Conditional	Limited to 2^{16} packets
Freshness	Weak freshness	Weak freshness	Not provided
Key management	Yes, limited	Yes, full	Not provided
Authenticated Acknowledgements	Yes	Not provided	Not provided
State	Secret	Partly secret	Open

Table 4: comparison of the security provision of SecRose and 802.15.14

As illustrated in Table 4, there are mainly three differences of SecRose with 802.15.14 (a) supports multiple MAC lengths, (b) provides a fully functional key management scheme with key provision authority and (c) does not support authenticated acknowledgements.

As discussed in 6.1.3, the length of the MAC in SecRose is optimised for nodes that operate on finite power sources and communicate at low rates. The less confident authentication provided by SecRose is not a drawback but an optimisation choice for a low requirement design. Both SecRose and 802.15.4 provide authentication that is secure for their assumed operational environment. SecRose could have provided 8 or 16 bytes long MACs if that had been a requirement.

However, SecRose does lack in the key management area, as it does not provide a scheme for key distribution. However, it does provide a limited key change mechanism, which might form the base for a complete key management scheme in the future.

On the other hand, 802.15.4 does not provide authenticated acknowledgements. In our opinion, this is a serious drawback for a generic, high-end security mechanism and it does not appear consistent with other provisions of the protocol.

Our thesis is that although SecRose was never intended to contest with 802.15.4, the evaluation shows that the two mechanisms provide comparable security when used with low-end sensor networks.

6.2 Performance evaluation

This section discusses the performance of SecRose and compares it with TinySec. Unfortunately, we were unable to find working implementations of MiniSec, SenSec, SPINS and FlexiSec. These mechanisms are either abandoned by their research groups or not yet implemented. For this reason, our comparative results are limited to experiments between SecRose and TinySec.

This section begins by a presentation of the energy requirements from CPU and Radio. Then the latency issues are addressed and the section concludes with comparisons of SecRose, TinySec and TinyOS.

Notes on scale

All figures have been scaled on the y-axis to improve the visible information. The scale is different for all figures and thus it is important to note the value range when reading the figures.

6.2.1 Explanation of methodology

Application for measurements

In order to conduct the measurements of this section, a test program was created, which could be run on both the Avrora [63] and the TOSSIM [62] simulators. The program was adapted for each measurement type in a way that it would conduct a reflective simulation. Each subsection begins by describing how the measurement was taken.

Effort has been made to standardise and minimise the impact of the external variables that would affect our measurements. All measurements were taken using two nodes and no routing protocol. This allows SecRose to be the only truly “variable” variable while measurements were taken.

Attention has also been paid to the validity, integrity and consistency of measurements. When possible, the same application was used with only the simulation parameters changing. This

allows a wider picture to be built for the overall performance of the simulator. All results are directly copy-pasted from the simulator output to excel where the diagrams were created. This minimises possible errors in the illustrations.

Validity of simulators

Both TOSSIM and Avrora were used for double validation where possible but the presented results were generated with Avrora. Both simulators are highly respected for their accurate operation and validity of results. TOSSIM has been used by the majority of the research community and has been cited by many publications; including [1, 63, 146]. Avrora, on the other hand, enjoys less success, as it is more recent but it is used by sensor networks researchers [147-148].

The Avrora simulator is highly accurate and can provide energy usage analysis based on the energy model presented on [149]. Comparative results that demonstrate SecRose's performance against TinySec are deemed fair since any simulation inaccuracies would equally affect both mechanisms.

6.2.2 Energy requirements

Description of experiment

The Avrora simulator can report the energy used by the CPU and the energy used by radio to execute a simulation. A sensor network application was created to run the experiments.

The application sends one packet with a set data payload. Thirty applications, one for each of the 0...29 data payloads, for each packet type were created.

Results: CPU energy requirements

The following figures illustrate the CPU energy consumption. The x -axis represents the increasing data payload while the y -axis is the CPU energy consumption in Joules. The red line is for *broadcast* packets, the green line is for *long* packets and the purple line is for *normal* packets. The diagrams also illustrate the different CPU energy requirements for each packet type.

Figure 22 shows the CPU energy requirement for the transmitting node. The energy is increased in steps, which relate with either the block size of the cipher or the performance of TinyOS. For example, the difference of payload 5 and 6 is due to one additional call to the encryption cipher, which is initiated by the MAC component. The same is visible after every four bytes – the length of the cipher’s minimum block size. The difference between payload 8 and 9 is due to one additional block of data input on the block cipher due to the payload’s size.

On the other hand, the odd difference between 14 and 15, where 15 bytes payload consume less energy than 14 is an artefact of when TinyOS decides to put the radio on sleep. Note that radio consumes energy at both the CPU and the transmitter itself.

Figure 23 shows the CPU energy requirement for the receiving node. In this Figure, TinyOS is less influential on the shape of the line and the actual differences SecRose’s in data handling are more visible.

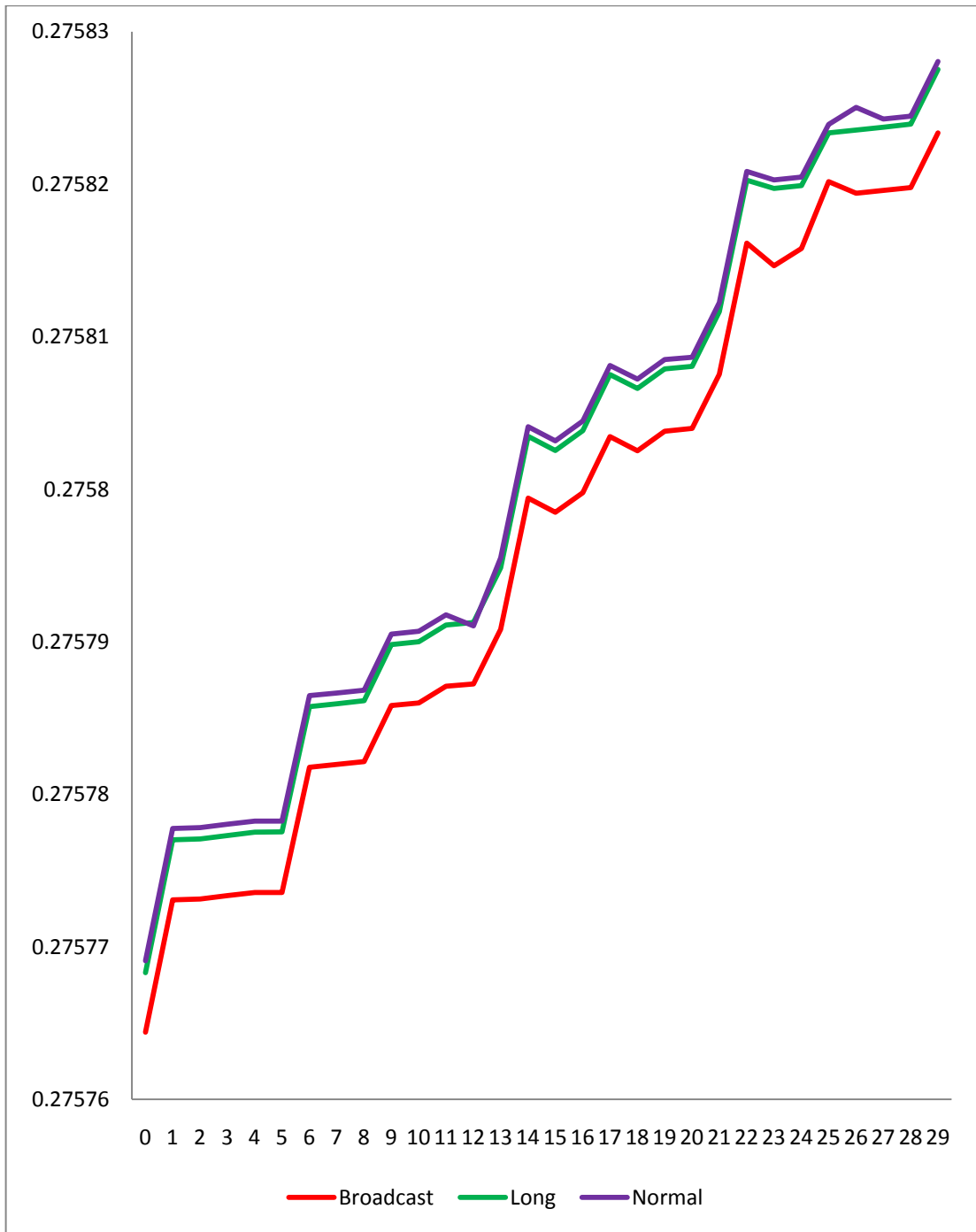


Figure 22: CPU energy requirements for a SecRose transmitter
 Values express energy in Joules (y), versus data payload in Bytes (x).

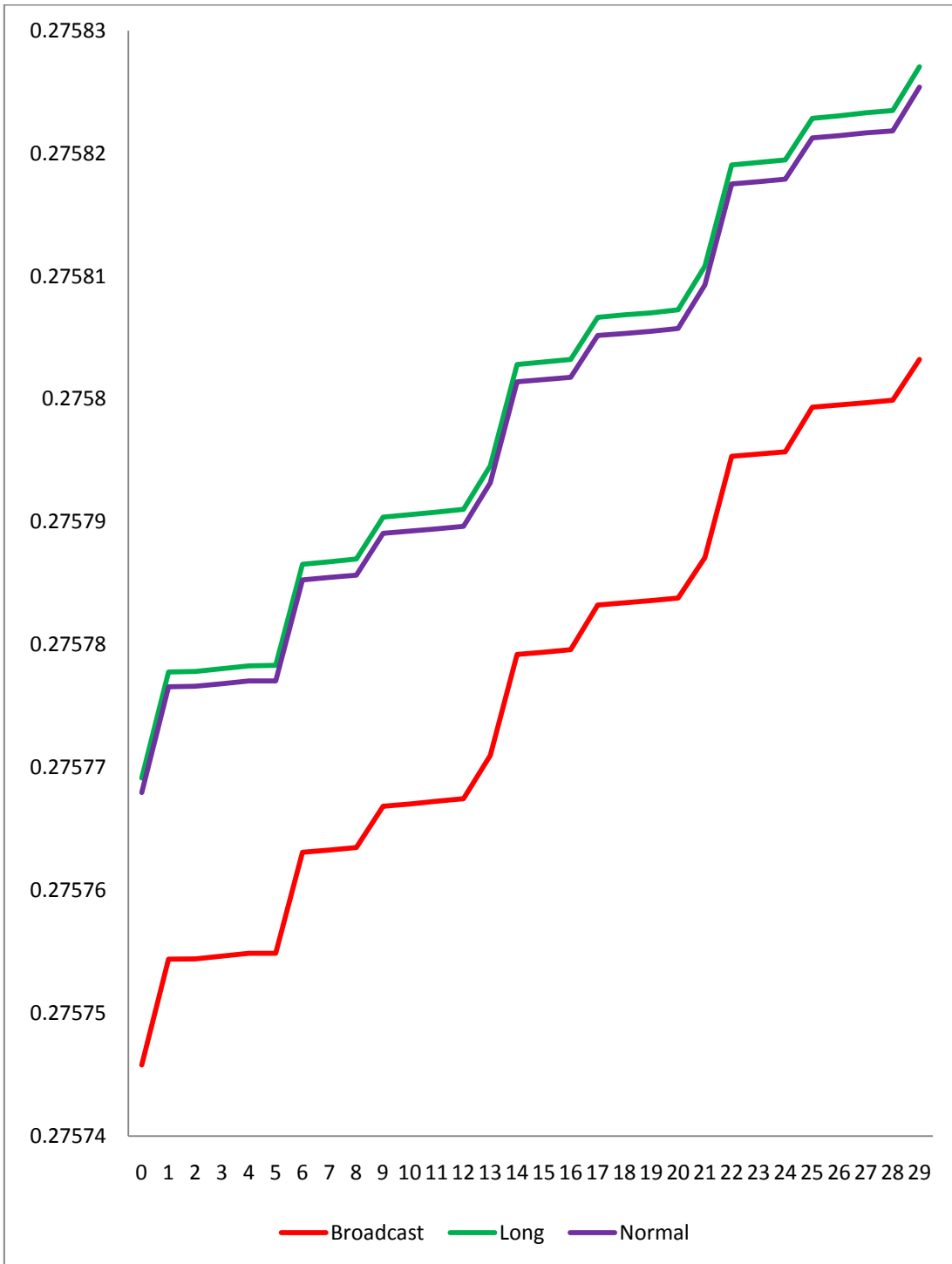


Figure 23: CPU energy requirements for a SecRose receiver. Values express energy in Joules (y), versus data payload in Bytes (x).

Results: radio energy requirements

Figure 24 illustrates radio energy consumption. The x -axis represents the increasing data payload while the y -axis shows the energy consumed by the radio in Joules. The red line is for *broadcast* packets, the green line is for *long* packets and the purple line is for *normal* packets. The diagram illustrates the differences in radio energy consumption when sending various packet types.

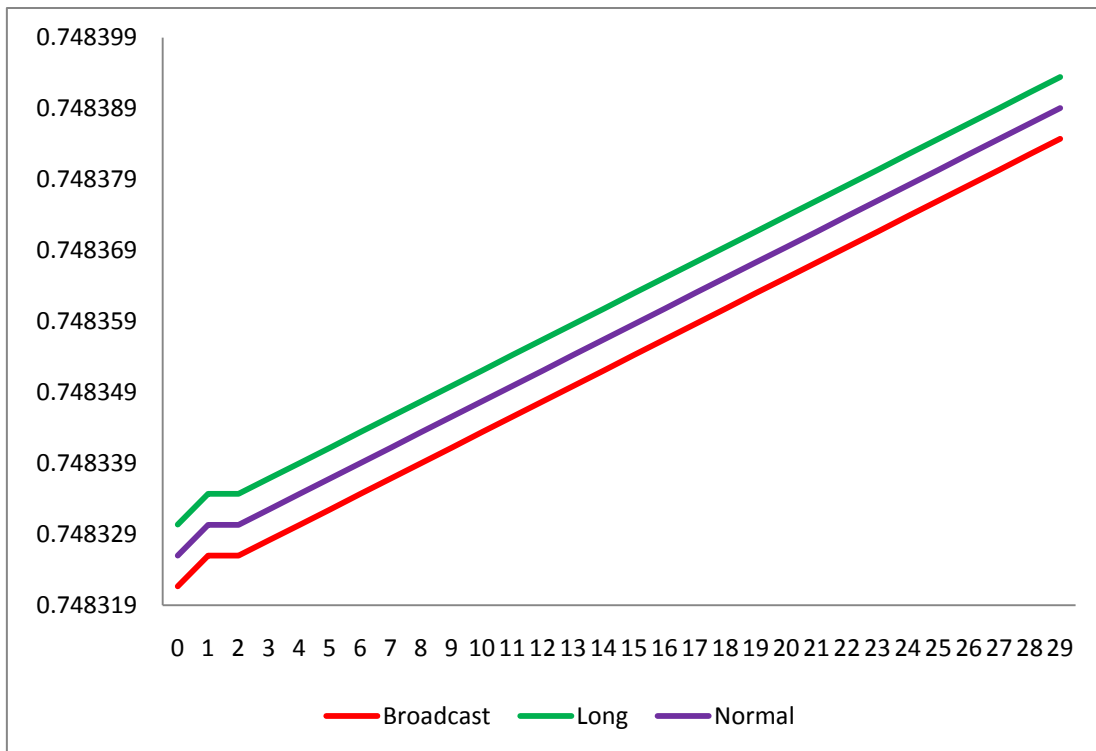


Figure 24: radio energy requirements for a SecRose transmitter.

Values express energy in Joules (y), versus data payload in Bytes (x).

Note that the transmitter energy does not include the acknowledgements.

The difference between payload 0 and 1 is actually two bytes, since there is one added padding byte for payload 1. This is also the reason why payloads 1 and 2 appear the same. At payload 2, both bytes are useful information and there is no padded byte.

The diagrams for reception energy are omitted since they do not provide significant information. Radio reception consumes the same listening energy regardless if set or unset bits are received.

6.2.3 Latency

Description of experiment

This experiment aims to prove that the latency introduced by SecRose, or any other mechanism, is not an important factor in the overall time required for an event to arrive from the sensor network to the base station.

The Avrora simulator was used for this experiment since it keeps extremely accurate timing information of the simulated seconds. An application that sends one *long* packet with 29 bytes payload is used to run the latency measurements at various stages of packet reception and transmission.

The experiment showed that 0.0244 seconds were required to process the packet, while 0.2431 seconds were required to transmit it over the medium. Therefore, SecRose could prepare 9.9 packets for every packet it transmits.

TinyOS does not implement a packet queue and so SecRose has time to prepare more packets than the radio can transmit. In addition, the TinyOS medium access introduces a delay between 100 and 163 milliseconds. In the minimum of 100 milliseconds, SecRose would have prepared 41.2 packets. Finally, SecRose can prepare one packet for as much time as required to send 5.2 bytes³¹. This is about 1/3 of the minimum packet length and thus SecRose would not have any problems sending bursts of 0-payload *broadcast* packets either.

Results: acknowledgement

The acknowledgement is transmitted and received separately as a packet. Its existence does not affect the sensor network application and therefore its delay is less important. The latency of the acknowledgement is not important for most WSN applications, as it is an internal SecRose feature that does not interfere with the rest of the application.

³¹ A 29-byte *long* packet requires actual transmission of 52 bytes. $52 / 9.9 = 5.2$.

Unless the sensor network sends bursts of packets very quickly, acknowledgement latency does not matter. However, bursts of packets might affect the frequency of key changes, since acknowledgements will not have time to arrive.

Conclusion

The results indicate that SecRose does not introduce significant latency in a sensor network. Considering that the medium access latency is at least 100 milliseconds and the longest SecRose packet is prepared, exchanged and validated in less than 25 milliseconds, any latency introduced by SecRose does not constitute more than 25% of the whole packet transmission process in any case.

6.2.4 Memory

Program sizes

The following table compares SecRose, TinySec and TinyOS program sizes. The same simulation application is used in all cases.

	Executable size (bytes)	ASM lines
SecRose	28868	6802
TinySec	28362	7478
TinyOS	28366	7477

Table 5: executable size of various proposals

Although there are small differences, all proposals are easily accommodated at the 128KB program flash memory of a typical sensor node like MICA2.

Runtime memory

The Avrora simulator reports that the runtime memory for all proposals peaks at 4351 bytes. The homogenous value, regardless of which proposal is measured, means that there are factors that affect the stack size other than the security mechanism.

Memory energy

Since sensor networks use Flash memory, the percentage of it that is occupied is irrelevant to the energy consumption. Flash memory consumes the same energy for every stored bit regardless of whether this it set or unset. Therefore, this metric is irrelevant to SecRose.

Conclusion

SecRose and TinySec do not impose any significant limit to the operation of the sensor network if compared with TinyOS. Further evaluation of memory usage in regards to scalability is discussed in 6.3.

6.2.5 Comparisons

Methodology

The same applications used for the discussed measurements were also used to measure the performance of TinySec and the plaintext TinyOS, for reference and further conclusions. To make differences more visible, the applications used for comparisons exchanged 1000 packets instead of 1. This reduced the influence of other tasks, like node boot-up, to the measurement results. Note that TinyOS does not transmit packets with zero data payload but SecRose and TinySec do. The 0-payload values for TinyOS are therefore out of scale in many of the figures and should be ignored.

This subsection presents accumulative diagrams for both nodes in contrast with the other proposals. Since *long* packets constitute the least well performing packets of SecRose, they were selected in order to represent the worst-case scenario. Similarly, the TinySec-AE packet was selected, as the only packet that provides sufficient security to be comparable with SecRose.

Energy consumption: CPU

Figure 25 illustrates the energy consumption of the CPU of the nodes. SecRose is represented in red, TinySec in green and TinyOS in blue. The advantage of SecRose in CPU consumption is clearly visible in this figure.

The difference between SecRose and the insecure TinyOS can be explained by the optimised and streamlined code in the whole network stack, which SecRose replaces. These engineering solutions have not been referred to in the past since they are out of the scope of this document.

The differences of SecRose with TinySec are mainly in the performance of the cipher and the stealing mechanism. TinySec uses padding to complete the first 8 bytes – one SkipJack block – therefore it appears to make the same effort for all the first 8 bytes.

The conclusion from this figure is that SecRose is not only a security mechanism; it is a good replacement for the network stack of TinyOS.

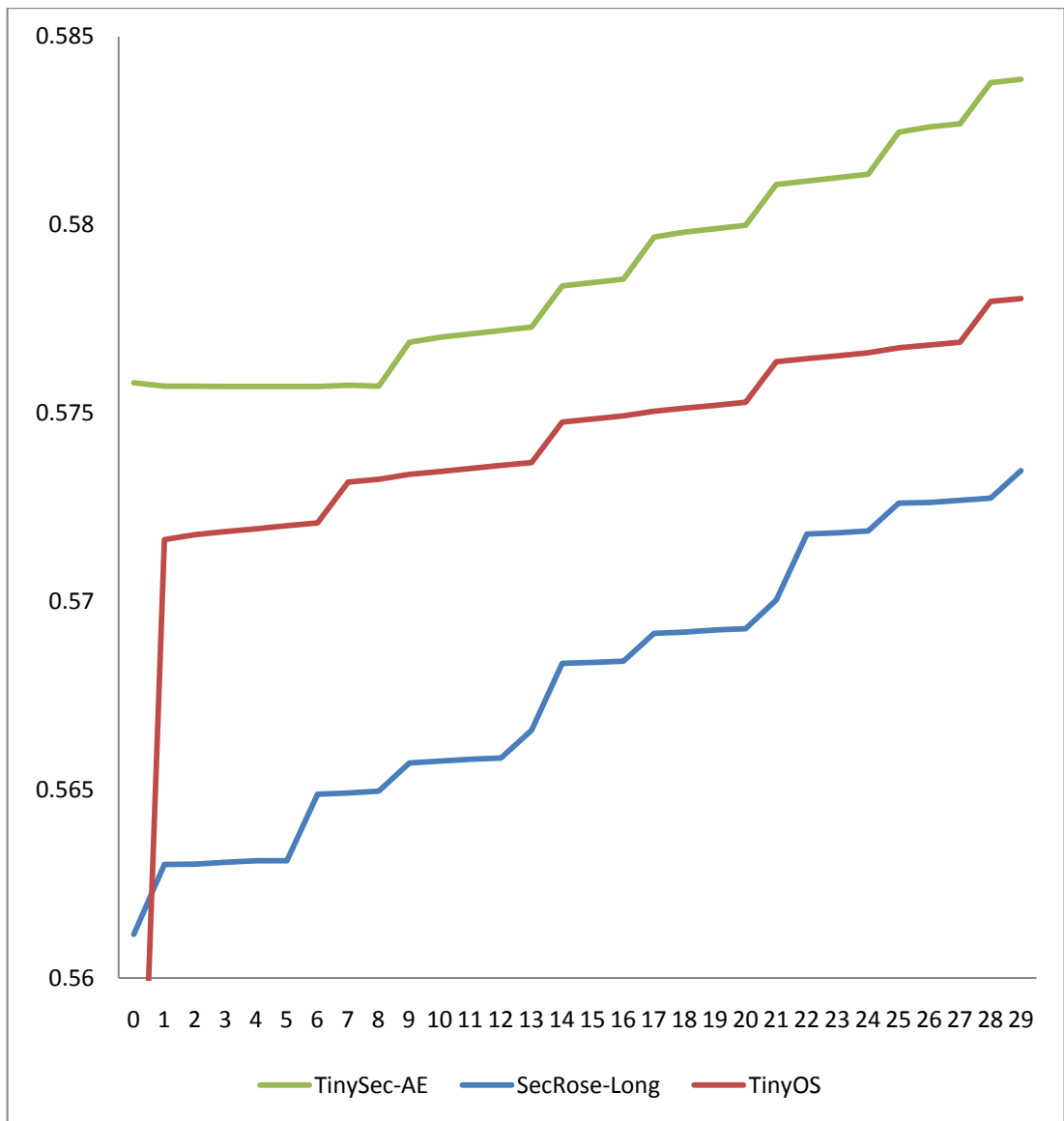


Figure 25: CPU energy requirements for both nodes.
 Values express energy in Joules (y), versus data payload in Bytes (x).

Energy consumption: radio

Figure 26 illustrates the energy consumed by the radio. SecRose is represented in red, TinySec in green and TinyOS in blue. Note that the figure is accumulative of the energy consumed for both reception and transmission.

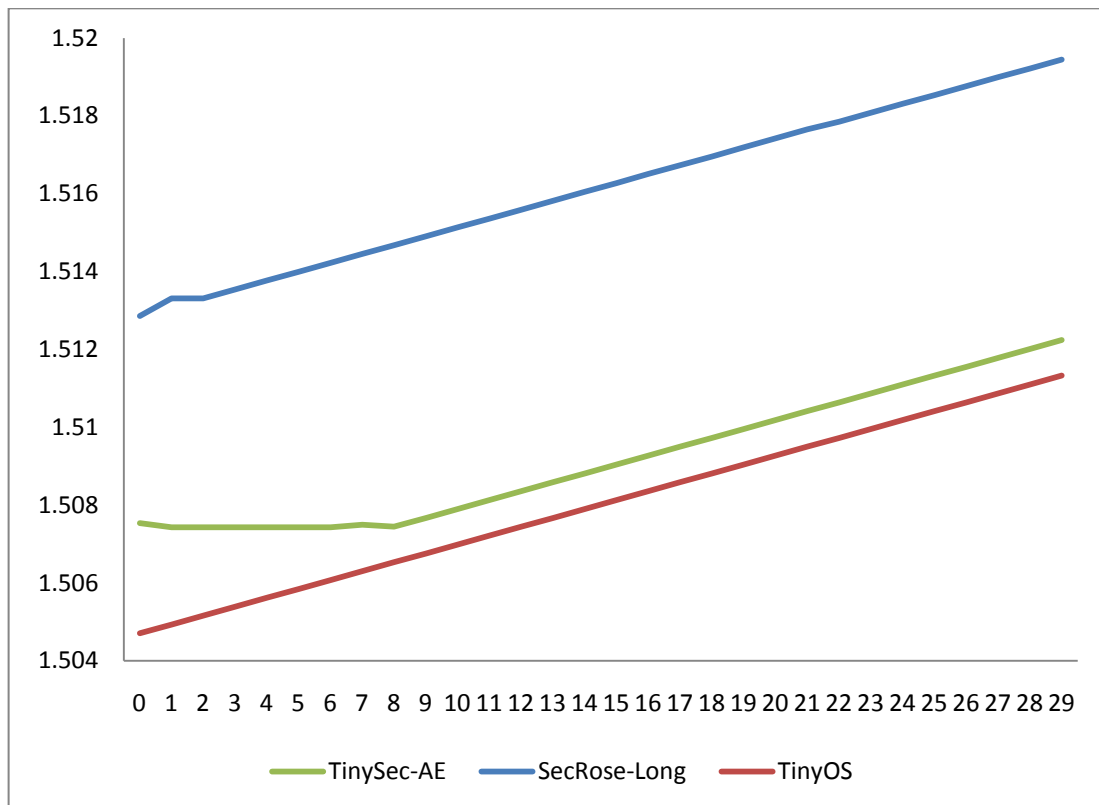


Figure 26: radio energy requirements for both nodes.

Values express energy in Joules (y), versus data payload in Bytes (x).

The disadvantage of the authenticated acknowledgements is visible. SecRose is required to send 16 acknowledgement bytes for each packet it sends, while SecRose and TinyOS will only send 4.

The operation of the stealing mechanism is also visible in the figure. TinySec's padding requires a whole block to be transmitted; regardless of how many of the data is useful information.

Energy consumption: total

Figure 27 illustrates the total energy consumption. SecRose is represented in red, TinySec in green and TinyOS in blue.

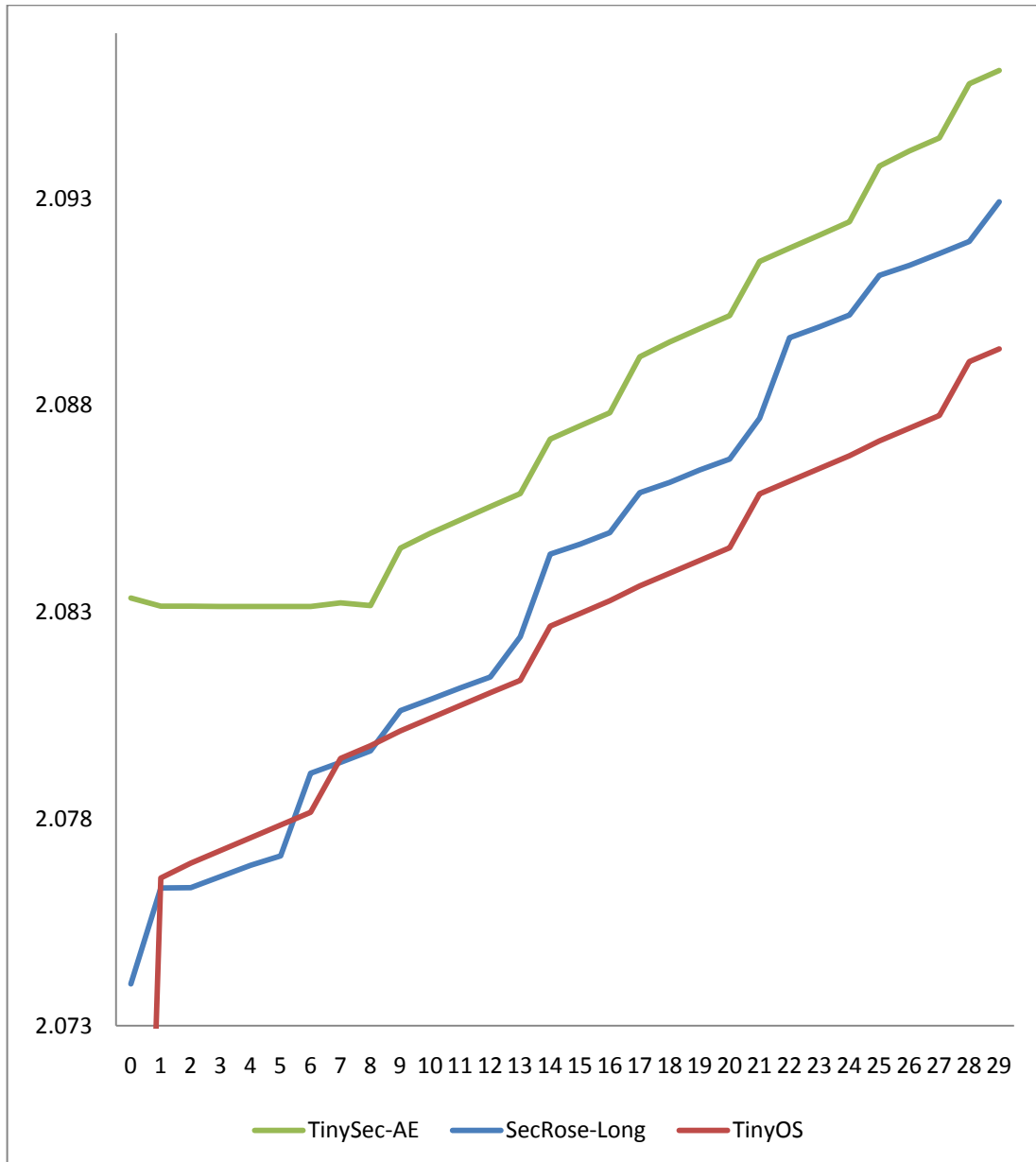


Figure 27: energy requirements for both CPU and radio.
Values express energy in Joules (y), versus data payload in Bytes (x)..

Energy consumption: normalised

Figure 28 is the same as

Figure 27 but normalised to illustrate the differences in relation with TinyOS, which is presented as the 100% baseline. Note that this graph does not include packets of 0 bytes since TinyOS does not send these packets and thus it consumes significantly less energy.

This figure illustrates energy consumption in relation with TinyOS for each different payload. For example, a packet with 14-bytes payload requires 0.08% more energy than TinyOS. The figure also illustrates that the difference between TinySec and SecRose is greater for small payloads.

The figure highlights the exceptional performance of SecRose, which can even consume *less* energy than the unsecured TinyOS for payloads of up to 5 bytes.

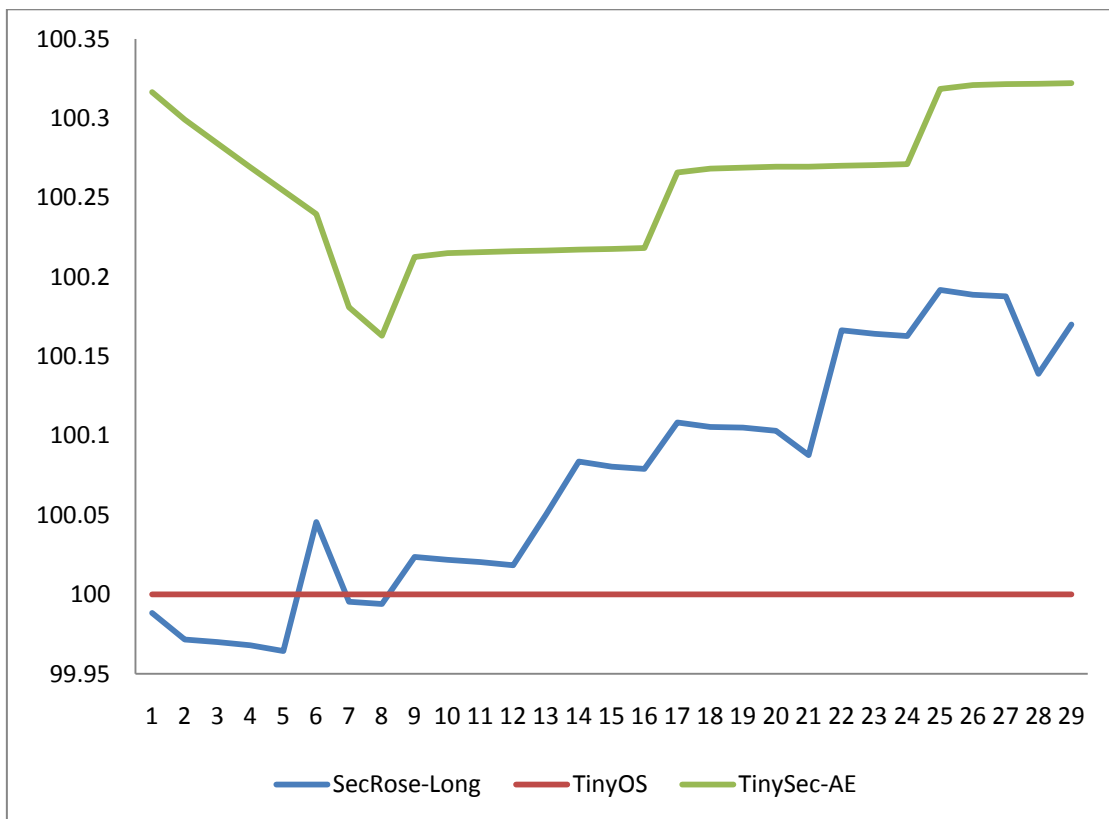


Figure 28: normalised energy requirements for both CPU and radio.

Values express energy in Joules (y), versus data payload in Bytes (x).

Note: 0-payload is not displayed.

Average performance differences

The following figures illustrate the accumulative total of reported energy consumption for all packets, all data payloads and both nodes. Energy in Joules is indicated on the vertical axis.

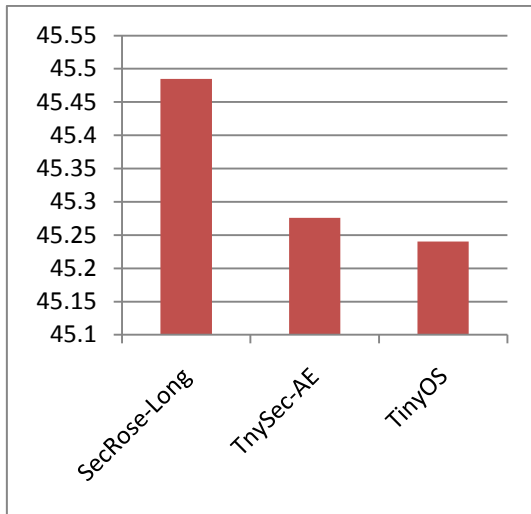


Figure 29: accumulative radio energy consumption for both nodes

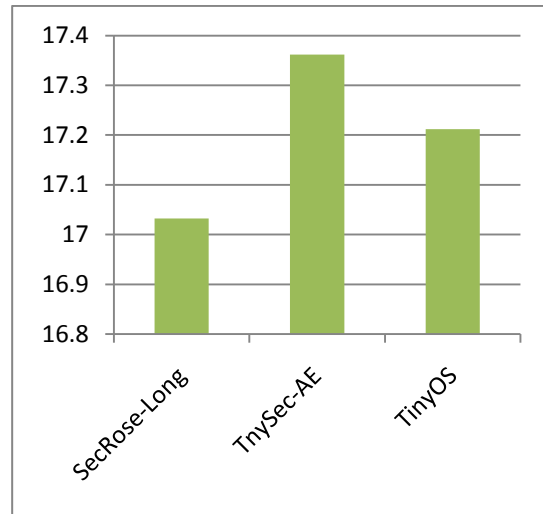


Figure 30: accumulative CPU energy consumption for both nodes

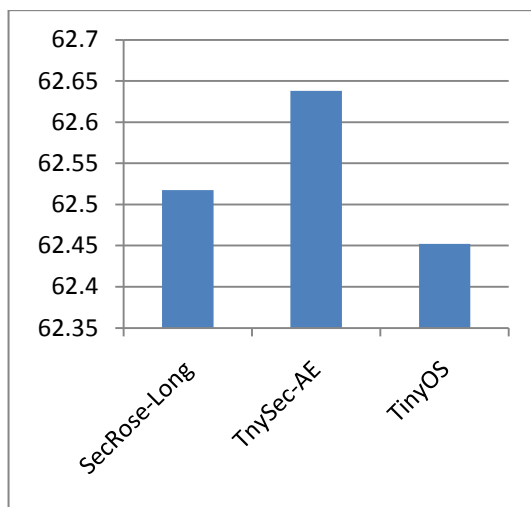


Figure 31: accumulative energy consumption for both nodes

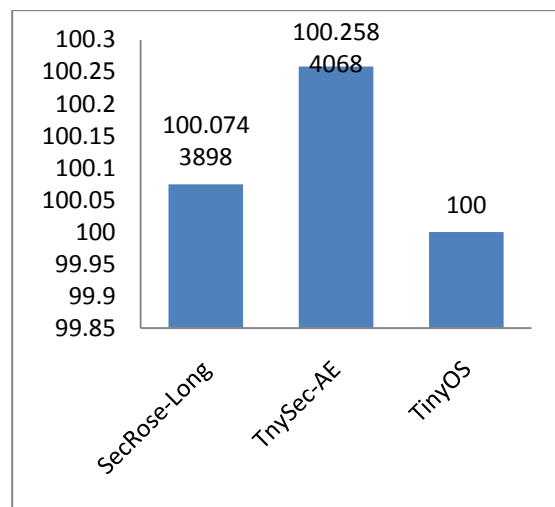


Figure 32: normalised accumulative energy consumption for both nodes

Figure 32 is the same data as Figure 31 normalised in relation with TinyOS, which is presented as the 100% baseline. On average, TinySec utilises 0.074% more energy than TinyOS while TinySec utilises 0.258% more. As with Figure 28, this figure does not include packets of 0-payload.

Impact of security on energy

The results indicated that any differences between the proposals are minimal if compared with the energy of the rest of a sensor node. For example, consider the two extremes of the results; the difference between 1000 unsecured TinyOS packets with 1-byte data payload and equal number of TinySec-AE packets with 29 bytes payload. TinyOS would consume 2.076574791 Joules while TinySec would require 2.096097628 Joules. TinySec requires only 0.94% more energy than TinyOS, when everything else is taken into consideration.

6.2.6 Discussion

Importance of efficiency

SecRose appears to be more efficient than TinySec and it enables further energy savings via the routing choice freedom that it provides by its authenticated acknowledgements. However, SecRose is a security protocol and it must be evaluated as such.

For these reasons, the actual energy performance figures are not important. After all, the differences are not great. As illustrated, the difference of SecRose with TinyOS is 0.074% while the difference of TinySec with TinyOS is 0.258%.

Our thesis is that the 0.074% difference with TinyOS is not significant in any way compared with the advantages of security provision. In addition, SecRose introduces less energy overhead than TinySec.

6.3 Evaluation of non-functional requirements

6.3.1 Essential deployment requirements

Backwards compatibility

SecRose has been developed as a replacement of the TinyOS network stack. As shown in other parts of this chapter, the replaced code performs better and achieves the security targets.

As explained in Chapter 5, SecRose is perfectly compatible with existing applications that have been designed for TinyOS and TinySec. The only requirement is re-compilation of the application with SecRose's files and re-deployment. No change is required in the application code. However, the current code of SecRose is only a proof-of-concept. Additional engineering work is needed to make it fully usable in a real environment.

SecRose introduces extra security features, which are optional and may be used by future applications. The existence of the features does not affect existing applications in any way. However, it is recommended that applications be updated to be able to communicate with SecRose better.

Note that the alternative proposals are not fully compatible with TinyOS because they do not include the Group field in their packet. SecRose has retained the Group field and thus it can transparently operate with existing applications.

Preconfigured distribution

SecRose is pre-configured with appropriate security parameters that would allow it to operate efficiently and securely by operators who have little connection with the area of security. SecRose has pre-selected values like the cipher's cycle count etc and there is no need to manipulate them unless a specific requirement is present.

Customisation

Sensor network operators who have specific security requirements can change the security level provided by SecRose dramatically. The code is written with appropriate constants, which reside on one header file, and can be changed at will.

6.3.2 Other requirements and desirables

Proof of concept

SecRose's proof of concept was developed and it is presented in Chapter 5 of this document. The evaluation in this subsection is done using the proof-of-concept.

Scalability

As per the requirements, SecRose does not pose any fundamental difficulty in scaling it up for use in larger networks. Although the mechanism is currently optimised to work on small networks that do not require clustering, it is not difficult to substitute the role of the base station with a cluster head and continue using SecRose in a clustered, scalable environment.

SecRose can scale up to large network sizes already. The state information required for each communication pair is 13 bytes³² on each node. The typical MICA2 node offers 128KB program memory, of which about 30KB is used by the program itself. SecRose can accommodate $\cong 7,000$ pairs on the remaining memory. Additional scalability is also possible, depending on the application, if the 512KB serial flash memory can be used as well. The upper scalability limit is about 40,000 pairs, meaning that networks of 80,000 nodes would be operational.

The current addressing scheme of TinyOS uses 16-bit address. Therefore, SecRose can cope with more nodes than TinyOS itself. Even if the serial flash memory is excluded, and assuming that nodes form pairs with only 10% of the rest of the network, SecRose can operate in networks with $\cong 70,000$ nodes, still higher than TinyOS.

³² Assuming 16-bit memory addressing, which is what is required to manage 512KB of memory.

However, all these calculations are indicative and real networks of such sizes, if they ever come to existence, should not use SecRose in an out-of-the-box basis.

Free of charge

SecRose is provided in an “as is” basis and free of any charges. This practice follows both ethical considerations and legal restrictions of the TinyOS licence upon which SecRose is based. In addition, care has been taken not to include any proprietary algorithm in SecRose, as that would cause legal problems and introduce royalty costs.

6.4 Summary

This section is a generic summary for SecRose comparing where it stands against the other proposals. The presented information is compiled from all previous chapters.

6.4.1 Critical security evaluation

Identical features

SecRose was developed after TinySec and it retained some of its features, as they were deemed adequate. The most important of these are the provided authentication guarantee and the assumption that hardware is secure.

The level of authentication guarantee is equal in all proposals, a 4-Byte MAC. The differences are in the ways the MAC is produced, SecRose and TinySec uses the cipher in CBC mode under the CMAC [99] recommendation by NIST [132] while SenSec and MiniSec use one-pass OCB by Rogaway *et al.* [100]. The two methods may provide different level of diffusion but any differences become much less important when a, relatively short, 4-Byte MAC is used.

Therefore, the only important characteristic is the Mac's size. Since this is identical in all proposals, the error rate of authentication is also identical and equal to 1 in 2^{32} .

In regards to hardware, all proposals assume that their non-transmitted secrets can remain secret, or in other words, that they operate in secure hardware.

Our thesis is that SecRose provides equal level of authentication as other proposals while it is designed to operate under the same assumptions.

Controversial features

SecRose provides data confidentiality in a radically alternative way. That includes all aspects of confidentiality: encryption strength, semantic security and freshness.

SecRose's 128-bit encryption is provided natively by a cipher designed and evaluated for this key length. In contrast, SenSec uses key whitening to provide a theoretical maximum of 144-bits encryption strength. However, this level of security is (a) potentially unnecessary and (b) reduced radically if the attacker obtains more than one plaintext/ciphertext sets in a known-plaintext cryptanalytic attack and (c) untested in SkipJack. On the other hand, TinySec and MiniSec do not provide an acceptable level of cryptographic strength.

Our thesis is that the cryptographic strength provided by SecRose is better than the other proposals since it satisfies the acceptable level unconditionally.

Data confidentiality is also about semantic security. SecRose provides semantic security in an alternative way, via its key management, which operates on a different condition than other proposals. Other proposals are limited by the number of exchanged packets while SecRose is unlimited but vulnerable to a detectable attack, making it less important and allowing a reaction by SecRose.

On the other hand, the solution is provided by SecRose without leaking any cryptographic state data to the attackers. It is difficult to draw a conclusion on which method is potentially weaker, as the conditions are not formally comparable.

A similar condition applies for freshness. Freshness is provided on SecRose and is effective under a different condition than MiniSec. SecRose's packets are all guaranteed to be weakly fresh as long as the attacker does not block acknowledgements, which is a detectable condition³³. Note that TinySec and SenSec do not provide freshness at all.

Our thesis is that SecRose uses alternative methods to provide semantic security and freshness. The solutions are characterised by different advantages and operate under different conditions.

³³ Broadcast packets are not subjected to this condition and are always fresh unconditionally.

Unique features

SecRose provides authenticated acknowledgements, which is a feature unique to it and is not provided by any other proposal. This feature is very important since it secures various routing protocols, which are currently unusable in a secure environment due to their security weaknesses.

In addition, this feature can be utilised by future applications and allow them to enjoy greater confidence that their message was received. If used appropriately, the feature can also provide trustworthy information on the health of the network.

Our thesis is that authenticated acknowledgements are an important feature, which constitutes a significant contribution of SecRose.

The second unique feature of SecRose is key management system and the way it preserves and synchronises its state, without leaking any information to attackers. It is desirable to keep the cryptographic state secret because there have been documented attacks on other cryptosystems that exploited problems like initialisation vectors [111].

Our thesis is that SecRose's key management provides comparable security but demonstrates greater potential for future developments.

6.4.2 Energy efficiency and non-functional requirements

SecRose provides packet categorisation with three different types that optimise the communication patterns of sensor networks. The types are implemented using an overloaded flag to characterise them and different packet fields, enabling energy-efficient addressing using default source/destination fields where available. In addition, the flag enables early rejection of unwanted packets, allowing further energy savings.

The whole packet type creation and categorisation is completely transparent to the application or the routing layers and fully compatible with existing applications.

Other proposals provide some categorisation between *normal* and broadcast communication but they do not provide the resolution of SecRose. In addition, no other proposal is fully compatible with all existing applications as they have not retained the Group field of TinyOS and thus they cannot operate in clustered applications, which work by partitioning the network in groups.

SecRose provides a radically new ciphertext stealing implementation as well. The solution is evidently more efficient than typical ciphertext stealing, like the one used in TinySec.

Finally, SecRose benefits from important engineering optimisations in the network stack of TinyOS, using much more efficient and streamlined code. All the parts of the code that consumed a significant portion of energy are heavily optimised in a per-line basis. The resulting code is much more efficient than the original TinyOS network stack and this is evident in the simulated results.

6.4.3 Final comparison

The following table is a rough illustrative estimation of the security provision of each proposal using a generalised scoring system. This table is provided as a summary and overview only. In reality, many features are difficult to quantify and compare.

To preserve fairness, the table is compiled supposing a worst-case scenario for SecRose and a best-case scenario for the other proposals, where the most impactful attack are launched against SecRose while the least impactful attacks are launched against the other proposals. Other advantages of SecRose’s proposal have not been accounted either.

The mechanisms score:

- 0 points for non-protection: the attack will succeed easily
- 1 point for inadequate protection: the attack is difficult but not impossible
- 2 points for limited or conditional protection; the attack may succeed
- 3 points for full protection: the attack is infeasible
- 4 points for more than adequate protection: the attack is infeasible and will remain so in the future

	Confidentiality		Authentication	Weak Freshness	Authenticated Acknowledgements	Total
	Encryption strength	Semantic security				
TinySec	1	3	3	0	0	7
MiniSec	1	3	3	2	0	9
SenSec	4	3	3	0	0	10
SecRose	3	1	3	1	3	11

Despite the worst-case assumption for SecRose, it appears to be the less vulnerable mechanism, primarily due to its authenticated acknowledgements. However, and most importantly, all mechanisms are still vulnerable under the right –for the attacker – conditions. Conclusively, more work is required to improve the security provision.

Chapter 7

Future Work and Conclusion

7. Conclusion and Future Work

7.1 Conclusion

SecRose is presented as an alternative to existing security mechanisms that operate in the data transportation layer. The mechanism benefits from a simple and efficient design, which provides for the basic security requirements of confidentiality, authentication and weak freshness.

The mechanism provides unique features like authenticated acknowledgements and key management, alternative solutions to existing features, like secret cryptographic state, and improvements to existing features, like 128-bits encryption strength. The mechanism secures all communication and allows developers to utilise routing protocols that are currently insecure. The design follows different directions than existing proposals and is therefore subject to different conditions and limitations.

SecRose introduces an average overhead of 0.074% compared with TinyOS while it is always more energy-efficient than the widely accepted TinySec proposal. SecRose requires less energy than TinyOS for packets with small data payloads.

Our thesis is that SecRose offers a security solution, which provides better overall security, is subject to different limitations and introduces less energy overheads than similar solutions.

7.2 Future work

7.2.1 Provision against current vulnerabilities

SecRose is vulnerable to certain low-impact attacks, affecting freshness and semantic security that need to be addressed in the future. This is essential in order to provide a better and more complete security mechanism, which would not be vulnerable in any computationally feasible way.

Freshness

SecRose provides conditional freshness. The feature is based on correct operation of the acknowledgements and on the fact that the attacker cannot block them and remain passive. There are a few possibilities that would improve this situation but they would all affect at least the energy consumption. A method that would provide unconditional freshness without energy overheads is the ideal target.

A straightforward solution would be to add an ordering counter, which would be transferred with the packet and advance with each sent packet. The counter can also be reset after each acknowledgement, since the key advances and a new era begins. However, the presence of the counter would consume radio energy and SecRose is already heavily influenced by the radio energy consumed by the acknowledgements.

A different key mixing method would also provide better freshness. Key mixing is currently done in the same way every time. The mixing can change so that it would be determined by how many packets were transmitted since the last key update. That would work efficiently under normal conditions but would require multiple validation attempts if something goes wrong. An attacker could deliberately cause errors and possibly create an asymmetric DoS attack by causing the nodes to undergo multiple packet validations in vain.

Semantic security

The ordering counter proposed above does not improve semantic security, unless the counter is included in the cryptographic input. On the other hand, if that happens then plaintext information is leaked to the attacker and this is unacceptable for SecRose.

A possible solution would be to include a highly diffused and encrypted IV. A potential candidate for this kind of IV is the last meta-byte generated by the MAC of the previous communication. This is a potent feature but it needs careful consideration, design and evaluation. If possible then it would provide semantic security at the expense of little additional cryptographic effort and slightly higher memory requirements.

7.1.2 Alternatives on authenticated acknowledgements

Authenticated acknowledgements are a completely contradictive feature. They elevate the security provision greatly, but consume significant amounts of energy. This is the first version of SecRose and thus the first design of the authenticated acknowledgements. A number of ideas that would improve this feature are presented.

Alter preambles

Packet preambles are a set string of bytes used to designate the start of a packet. They are 12 bytes long in the MICA2 nodes but their size is not necessarily fixed. Improvements on the preambles are possible but further research on preambles is required.

The size of the preamble can be greatly reduced if they are combined with cryptographically strong randomisation. Our research has proven that a 4-byte MAC is variable enough to authenticate a packet and thus there is no reason why a similarly sized stream of bytes could not authorise the start of a packet as well. However, the string has to be both randomised and uniformly known to the network.

Alternatively, the size might be reducible without provision. Our research has not found justification on the current preamble size since the literature does not explain the criteria behind the selection of that size. On this basis, an academic study on the proper preamble size

might be required. However, this research is out of the scope of security and thus we were unable to conduct it in this context.

If none of the above ideas is effective and preambles have to remain that long, then maybe SecRose can utilise them to provide additional features. For example, a preamble that is created via a cryptographic authenticated process might include an initialisation vector or an ordering counter. This option could improve freshness or semantic security or both.

Multilevel solution

SecRose could use a combination of link and network layer acknowledgements. Under normal conditions, nodes would use a short link-layer acknowledgement to designate that a packet was forwarded. If that does not happen, then the last node can send a long network-layer message similar to the “host unreachable” message of the IP protocol.

In order for this technique to work securely, both messages need to be authenticated. This increases the length of the “short” message. The “short” message would be longer than TinyOS-style link-layer acknowledgements, as it would be required to include both a credible message and information on the ID of the originator.

In addition, such system would introduce slightly more processing for all forwarding nodes. On the other hand, this potent feature might be the solution to the authenticated acknowledgement problem. The exact details regarding operation and energy consumption of this idea need to be evaluated.

7.2.3 Improvements on key management

Key management incorporated with data transportation is also a new idea introduced by this version of SecRose. As with the authenticated acknowledgements, there might be room for improvement in this area as well.

Better mixing

As discussed, the current mixing is invariable. Better mixing might improve freshness but it should also be designed in a way to improve counter diffusion as well. The *counter value* is currently initialised to a random number to address this problem but since the *counter update value* is a limited 8-bit value, the mixing is not ideal. Design and evaluation is required to provide semantic security, freshness and better diffusion in the same time.

Key update provision on key management

Key management may benefit from a key provision protocol, which would allow the nodes to request a key from an authoritative node or the base station, in similar fashion as SSL.

On the other hand, such functionality might be unnecessary for the data transportation layer. Effectively, if implemented, it would complete the key management mechanism. However, would that be useful enough to all applications to justify its energy expenses?

7.2.4 Regarding energy and performance

SecRose is already highly optimised and this is evident in the performance evaluation. Further optimisations might be possible after more engineering or research work.

One-pass encryption and authentication

The possibility of implementing OCB encryption and authentication in one-pass should be studied. The current mechanism uses more than one encryption interactions to encrypt and authenticate 8 bytes while an OCB style solution would be use only one interaction plus some additional processing

It is unknown what the impact of the additional processing is, compared to an encryption interaction. The security of OCB does not enjoy NIST recommendation status, as does the currently used CMAC, and therefore any such system should include a study on the security as well.

Further software engineering work

The proof-of-concept implementation does include optimisations in energy-demanding parts of the code but the rest of the implementation does not enjoy the same level of functionality. The related work should be conducted before SecRose is publicly released.

7.2.5 Flexibility and customisation features

Some features might be added to provide flexibility and a wider variety of options to match different sensor networks.

Acknowledged broadcast

Currently SecRose does not provide acknowledgements for *broadcast* packets as it was deemed unnecessary and energy consuming. The optional provision of these acknowledgements will improve the fitness of SecRose to some applications.

However, proper introduction of the feature would require appropriate research before the design and careful evaluation of the outcome. Special attention must be given to the evaluation of the impact on security and energy consumption if the feature is provided.

The semantic security and freshness advantages will be lost. The base station will require greater memory resources and there will be a significant impact in the energy requirements of the base station and its neighbouring nodes.

Backward compatibility

SecRose has retained the Group field of TinyOS in order to maintain backwards compatibility. We believe this feature should gradually be phased out and eventually replaced by other grouping mechanisms.

Making the feature optional is mainly an engineering challenge but the field is used by the stealing mechanism as well. Redesign of the stealing mechanism and subsequent evaluation of the effects is required.

Longer MACs

SecRose does not utilise one of the eight bytes of the MAC in any way. Our research has shown that utilisation of this byte will increase the security provision in networks that benefit by an unlimited power source. Functionality to use a 5-Byte MAC, when this is beneficial, would be a minor improvement in SecRose.

APPENDICES

IMPLEMENTATION CODE

The two important files of the proof-of-concept implementation are appended for reference and completeness reasons. The file that contains the control component is named `CC1000RadioIntM.nc` on the MICA2 implementation and `MicaHighSpeedRadioM.nc` for the pc target. The file listed here is the native MICA2 version since the code is better structured. The second important file is the `SecRoseM.nc` file, which contains every other component of `SecRose`. Note that some comments may be truncated.

File CC1000RadioIntM.ns

```
// $Id: CC1000RadioIntM.nc,v 1.23.2.8 2003/08/26 22:33:30 philipb Exp $
includes crc;
module CC1000RadioIntM {
  provides {
    interface StdControl;
    interface BareSendMsg as Send;
    interface ReceiveMsg as Receive;
    command result_t EnableRSSI();
    command result_t DisableRSSI();
    command result_t SetListeningMode(uint8_t power);
    command uint8_t GetListeningMode();
    command result_t SetTransmitMode(uint8_t power);
    command uint8_t GetTransmitMode();
    interface RadioCoordinator as RadioSendCoordinator;
    interface RadioCoordinator as RadioReceiveCoordinator;
  }
  uses {
    interface PowerManagement;
    interface StdControl as CC1000StdControl;
    interface CC1000Control;
    interface Random;
    interface ADCControl;
    interface ADC as RSSIADC;
    interface SpiByteFifo;
    interface StdControl as TimerControl;
    interface Timer as WakeupTimer;
    interface Leds;
    interface SecRose as rose;
    interface SendMsg as TrueSend;
  }
}
implementation {
  enum {
    IDLE_STATE,
    TX_STATE,
    DISABLED_STATE,
    POWER_DOWN_STATE,
    PRETX_STATE,
    SYNC_STATE,
    RX_STATE,
  };

  enum {
    TXSTATE_WAIT,
    TXSTATE_START,
  }
}
```

```

    TXSTATE_PREAMBLE,
    TXSTATE_SYNC,
    TXSTATE_DATA,
    TXSTATE_CRC,
    TXSTATE_FLUSH,
    TXSTATE_DONE
};

enum {
    SYNC_BYTE =          0x33,
    NSYNC_BYTE =         0xcc,
    SYNC_WORD =          0x33cc,
    NSYNC_WORD =         0xcc33
};

enum {
    TX_HEAD, TX_DATA
};

uint8_t RadioState;
uint8_t RadioTxState;
uint16_t txlength;
uint16_t rxlength;
TOS_MsgPtr txbufptr; // pointer to transmit buffer
TOS_MsgPtr rxbufptr; // pointer to receive buffer
TOS_Msg RxBuf; // save received messages
uint8_t NextTxByte;
uint8_t lplpower; // low power listening mode
uint8_t lplpowertx; // low power listening transmit mode
uint16_t preamblelen; // current length of the preamble
uint16_t PreambleCount; // found a valid preamble
uint8_t SOFCount;
union {
    uint16_t W;
    struct {
        uint8_t LSB;
        uint8_t MSB;
    };
} RxShiftBuf;
uint8_t RxBitOffset; // bit offset for spibus
int8_t RxByteCnt; // received byte counter
int8_t TxByteCnt;
uint16_t RSSISampleFreq; // in Bytes rcvd per sample
bool bInvertRxData; // data inverted
bool bTxPending;
bool bTxBusy;
bool bRSSIValid;
uint16_t usRSSIVal;
uint16_t usSquelchVal;

```

```

int16_t sMacDelay;    // MAC delay for the next transmission
volatile uint16_t LocalAddr;

// Secrose variables:
uint8_t flag;                // the flag of the packet
bool ack_send = FALSE;      // packets that fail to be MAC'ed with primary key must not send ACKS. This var controls the process
struct TOS_Msg ack_ptr;
int j, k;                    // temp
uint8_t rec_stop;           // when to stop receiving data
bool handle_flag = TRUE;    // to know if flag needs attention
bool handle_index = FALSE;  // to know when to "jump" the index of the recv ptr to a more appropriate pos
uint8_t header_len = SECROSE_HEADER_LENGTH; // total size of packet headers (source, dest etc), 56 by default
uint8_t max_data_len = TOSH_DATA_LENGTH;    // maximum data length, 29 by default
uint8_t mac_len = SECROSE_MAC_LENGTH;        // length of the mac address, 4 by default
uint8_t actual_data_len = 0;                 // length of data to be sent in the current packet (variable)
uint8_t stream_len = 7;                      // length of actual stream = header_len+actual_data_len+mac_len
uint8_t partial_stream_len = 0;              // the first part of stream length = header_len+actual_data_len
uint8_t mac_tmp[SECROSE_MAC_LENGTH];         // for storing the mac
uint8_t mac_pos;                             // for looping around the mac while storing
bool mac_valid;                              // for checking the MAC
TOS_MsgPtr rxbufptr_copy;                    // copy of pointer to receive buffer
char Byte;                                   // instead of locally redefining this variable for every byte, define it once.

// thsi has to be implemented, although we don't need it to do anything. It is triggered when an ACK is sent.
event result_t TrueSend.sendDone(TOS_MsgPtr m, result_t s) { return s; }

task void PacketRcvd() {
    TOS_MsgPtr pBuf;
    atomic {
        if ( flag != 3 ) {
            // do normal TinyOS tasks
            atomic {
                rxbufptr->time = 0;
                pBuf = rxbufptr;
                // EWMA to determin squelch values
                usSquelchVal = (((5*rxbufptr->strength) + (3*usSquelchVal)) >> 3);
            }
            pBuf = signal Receive.receive((TOS_MsgPtr)pBuf);
            atomic {
                if (pBuf)
                    rxbufptr = pBuf;
                rxbufptr->length = 0;
                //RadioState = IDLE_STATE;
            }
            call SpiByteFifo.enableIntr();

            // SecRose tasks
            if ( (rxbufptr->crc==1) && (ack_send) ) {
                // update counter
                call rose.counterHandle(rxbufptr, PKT_RECV);
            }
        }
    }
}

```

```

        // send ACK only if the packet was not broadcast
        if ( rxbufptr->flag != 1 ) {
            ack_ptr.source = TOS_LOCAL_ADDRESS;
            ack_ptr.flag = 3;
            call TrueSend.send(rxbufptr->source,0,&ack_ptr);
        }
    }
} // end atomic
}

task void PacketSent() {
    TOS_MsgPtr pBuf; //store buf on stack

    atomic {
        txbufptr->time = 0;
        pBuf = txbufptr;
    }
    signal Send.sendDone((TOS_MsgPtr)pBuf,SUCCESS);
    atomic bTxBusy = FALSE;

    // SEC-ROSE counter handling
    // Call counterHandle() in order to store the counter value
    if ( txbufptr->flag != 3 ) {
        call rose.counterHandle(txbufptr, PKT_SENT);
    }
}

command result_t StdControl.init() {
    bool temp;

    atomic {
        RadioState = DISABLED_STATE;
        RadioTxState = TXSTATE_PREAMBLE;
        rxbufptr = &RxBuf;
        rxbufptr->length = 0;
        rxlength = MSG_DATA_SIZE-2;
        RxBitOffset = 0;

        PreambleCount = 0;
        RSSISampleFreq = 0;
        RxShiftBuf.W = 0;
        bTxPending = FALSE;
        bTxBusy = FALSE;
        bRSSIValid = FALSE;
        sMacDelay = -1;
        usRSSIVal = -1;
        lplpower = lplpowertx = 0;
        usSquelchVal = PRG_RDB(&CC1K_LPL_SquelchInit[lplpower]);
    }
}

```

```

call SpiByteFifo.initSlave(); // set spi bus to slave mode
call CC1000StdControl.init();
call CC1000Control.SelectLock(0x9); // Select MANCHESTER VIOLATION
temp = call CC1000Control.GetLOSstatus(); //Do we need to invert Rcvd Data?
atomic bInvertRxData = temp;

call ADCCControl.bindPort(TOS_ADC_CC_RSSI_PORT,TOSH_ACTUAL_CC_RSSI_PORT);
call ADCCControl.init();

call Random.init();
call TimerControl.init();

// don't enable SPI interrupts until the radio is running
//call SpiByteFifo.enableIntr(); // enable spi and spi interrupt

LocalAddr = TOS_LOCAL_ADDRESS;

return SUCCESS;
}

command result_t EnableRSSI() { return SUCCESS; }
command result_t DisableRSSI() { return SUCCESS; }

command uint8_t GetTransmitMode() {
return lplpowertx;
}
command result_t SetTransmitMode(uint8_t power) {
result_t Result = SUCCESS;
if ((power >= CC1K_LPL_STATES) || (power == lplpowertx))
return FAIL;

atomic {
// check if the radio is currently doing something
if ((!bTxPending) && ((RadioState == POWER_DOWN_STATE) ||
(RadioState == IDLE_STATE) ||
(RadioState == DISABLED_STATE))) {
lplpowertx = power;
preamblelen = ((PRG_RDB(&CC1K_LPL_PreambleLength[lplpowertx*2]) << 8)
| PRG_RDB(&CC1K_LPL_PreambleLength[(lplpowertx*2)+1]));
}
else {
Result = FAIL;
}
}
return Result;
}

command result_t SetListeningMode(uint8_t power) {

```

```

result_t Result = SUCCESS;
// valid low power listening values are 0 to 3
// 0 is "always on" and 3 is lowest duty cycle
// 1 and 2 are in the middle
if ((power >= CC1K_LPL_STATES) || (power == lplpower))
    return FAIL;

atomic {
    // check if the radio is currently doing something
    if ((!bTxPending) && ((RadioState == POWER_DOWN_STATE) ||
        (RadioState == IDLE_STATE) ||
        (RadioState == DISABLED_STATE))) {

        // change receiving function in CC1000Radio
        call WakeupTimer.stop();
        if (lplpower == lplpowertx) {
            lplpowertx = power;
        }
        lplpower = power;

        // if successful, change power here
        if (RadioState == IDLE_STATE) {
            //RadioState = DISABLED_STATE;
            call StdControl.stop();
            call StdControl.start();
        }
        if (RadioState == POWER_DOWN_STATE) {
            //RadioState = DISABLED_STATE;
            call StdControl.start();
            call PowerManagement.adjustPower();
        }
    }
    else {
        Result = FAIL;
    }
}
return Result;
}

command uint8_t GetListeningMode() {
    return lplpower;
}

event result_t WakeupTimer.fired() {
    uint8_t oldRadioState;
    uint16_t sleeptime;
    bool bStayAwake;
}

```

```

if (lplpower == 0)
    return SUCCESS;

atomic {
    oldRadioState = RadioState;
    bStayAwake = bTxPending;
}

switch(oldRadioState) {
case IDLE_STATE:
    sleeptime = ((PRG_RDB(&CC1K_LPL_SleepTime[lplpower*2]) << 8) |
        PRG_RDB(&CC1K_LPL_SleepTime[(lplpower*2)+1]));
    if (!bStayAwake) {
        atomic RadioState = POWER_DOWN_STATE;
        call WakeupTimer.start(TIMER_ONE_SHOT, sleeptime);
        call CC1000StdControl.stop();
        call SpiByteFifo.disableIntr();
    }
    else {
        call WakeupTimer.start(TIMER_ONE_SHOT, CC1K_LPL_PACKET_TIME*2);
    }
    break;

case POWER_DOWN_STATE:
    sleeptime = PRG_RDB(&CC1K_LPL_SleepPreamble[lplpower]);
    atomic RadioState = IDLE_STATE;
    call CC1000StdControl.start();
    call CC1000Control.BIASOn();
    call SpiByteFifo.rxMode();           // SPI to miso
    call CC1000Control.RxMode();
    call SpiByteFifo.enableIntr(); // enable spi interrupt
    call WakeupTimer.start(TIMER_ONE_SHOT, sleeptime);
    break;

default:
    call WakeupTimer.start(TIMER_ONE_SHOT, CC1K_LPL_PACKET_TIME*2);
}
return SUCCESS;
}

command result_t StdControl.stop() {
    atomic RadioState = DISABLED_STATE;

    call WakeupTimer.stop();
    call CC1000StdControl.stop();
    call SpiByteFifo.disableIntr(); // disable spi interrupt
    return SUCCESS;
}

command result_t StdControl.start() {

```

```

uint8_t chkRadioState;
atomic chkRadioState = RadioState;
if (chkRadioState == DISABLED_STATE) {
    atomic {
        rxbufptr->length = 0;
        RadioState = IDLE_STATE;
        bTxPending = bTxBusy = FALSE;
        sMacDelay = -1;
        preamblelen = ((PRG_RDB(&CC1K_LPL_PreambleLength[lplpowertx*2]) << 8) |
            PRG_RDB(&CC1K_LPL_PreambleLength[(lplpowertx*2)+1]));
    }
    if (lplpower == 0) {
        // all power on, captain!
        call CC1000StdControl.start();
        call CC1000Control.BIASOn();
        call SpiByteFifo.rxMode();           // SPI to miso
        call CC1000Control.RxMode();
        call SpiByteFifo.enableIntr(); // enable spi interrupt
    }
    else {
        uint16_t sleeptime = ((PRG_RDB(&CC1K_LPL_SleepTime[lplpower*2]) << 8) |
            PRG_RDB(&CC1K_LPL_SleepTime[(lplpower*2)+1]));
        atomic RadioState = POWER_DOWN_STATE;
        call TimerControl.start();
        call WakeupTimer.start(TIMER_ONE_SHOT, sleeptime);
    }
}
return SUCCESS;
}

command result_t Send.send(TOS_MsgPtr pMsg) {
    result_t Result = SUCCESS;
    atomic {
        if (bTxBusy) {
            Result = FAIL;
        }
        else {
            bTxBusy = TRUE;
            txbufptr = pMsg;
            txlength = pMsg->length + (MSG_DATA_SIZE - DATA_LENGTH - 2);
            // initially back off a message + [0,127] radio bytes
            sMacDelay = MSG_DATA_SIZE + (call Random.rand() & 0x7F);
            bTxPending = TRUE;
        }
    }
    if (Result) {
        uint8_t tmpState;
        atomic tmpState = RadioState;
        // if we're off, start the radio
        if (tmpState == POWER_DOWN_STATE) {

```



```

        // disable wakeup timer
        call WakeupTimer.stop();
        call CC1000StdControl.start();
        call CC1000Control.BIASOn();
        call CC1000Control.RxMode();
        call SpiByteFifo.rxMode(); // SPI to miso
        call SpiByteFifo.enableIntr(); // enable spi interrupt
        call WakeupTimer.start(TIMER_ONE_SHOT, CC1K_LPL_PACKET_TIME*2);
        atomic RadioState = IDLE_STATE;
    }

    // SecRose tasks to prepare packet
    atomic {
        txbufptr = pMsg;

        TxByteCnt = 1;
    }

    // from ChannelMon.startSymDetect() :
    atomic {
        // from ChannelMon.idleDetect()
        flag = call rose.findFlag(txbufptr);
    }

    // normal packet. always set the source address so that it will be included in the MAC calculation
    txbufptr->source = TOS_LOCAL_ADDRESS;

    if ( txbufptr->flag == 3 ) call rose.calcACK(txbufptr, txbufptr->addr); // For ACK packets calculate the ACK
    else call rose.calcMAC(txbufptr, txbufptr->addr, SEND); // For all other packets calculate the MAC

    // encrypt packet
    call rose.packetEncDec(txbufptr, ENCRYPT);

    atomic {
        handle_index = TRUE;
        actual_data_len = txbufptr->length + txbufptr->pad_size; // calculate data+pad length
        partial_stream_len = header_len+actual_data_len; // calculate header+data+pad subtotal
        stream_len = header_len+actual_data_len+mac_len+1; // calculate grand total
        // overload the len with the flag
        flag = call rose.writeFlag(txbufptr);
        // normal packet. we have to replace destination addr with source addr
        if ( flag == 0 ) { txbufptr->addr = TOS_LOCAL_ADDRESS; }
    }
}
return Result;
}

async event result_t SpiByteFifo.dataReady(uint8_t data_in) {
#ifdef ENABLE_UART_DEBUG

```

```

UARTPutChar(RadioState);
#endif

if ( RadioState == IDLE_STATE ) {
    if ( data_in ) {
        if (((data_in == (0xaa)) || (data_in == (0x55)))) {
            PreambleCount++;
            if (PreambleCount > PRG_RDB(&CC1K_LPL_ValidPrecursor[lplpower])) {
                PreambleCount = SOFCnt = 0;
                RxBitOffset = RxByteCnt = 0;
                rxlength = MSG_DATA_SIZE-2;
                RadioState = SYNC_STATE;
            }
        }
    }
    else if (bTxPending && (--sMacDelay <= 0)) {
        bRSSIValid = FALSE;
        call RSSIADC.getData();
        PreambleCount = 0;
        RadioState = PRETX_STATE;
    }
    return;
}

switch (RadioState) {
case TX_STATE:
    {
        call SpiByteFifo.writeByte(NextTxByte);
        TxByteCnt++;
        switch (RadioTxState) {

        case TXSTATE_PREAMBLE:
            if (!(TxByteCnt < preamblelen)) {
                NextTxByte = SYNC_BYTE;
                RadioTxState = TXSTATE_SYNC;
            }
            break;

        case TXSTATE_SYNC:
            NextTxByte = NSYNC_BYTE;
            RadioTxState = TXSTATE_DATA;
            TxByteCnt = -1;
            signal RadioSendCoordinator.startSymbol(); // for Time Sync
            break;

        case TXSTATE_DATA:
            if ( TxByteCnt <= stream_len ) {
                if ( flag != 3 ) {
                    if ( handle_index ) {
                        // normal packet, only has source addr

```

```

        if ( (flag==0) && (TxByteCnt==3) ) { TxByteCnt += 2; handle_index = FALSE; }
        // broadcast packet, does not have source or dest
        if ( (flag==1) && (TxByteCnt==1) ) { TxByteCnt += 4; handle_index = FALSE; }
    }

    if(TxByteCnt == partial_stream_len){
        TxByteCnt = MAC_POS;
        stream_len = MAC_POS+SECROSE_MAC_LENGTH;
    }

    NextTxByte = ((char*)txbufptr)[TxByteCnt];
}
else { // control packets are less generic in mica2 due to lack of time
    if ( SECROSE_MAX_NODES > 255 ) {
        switch (TxByteCnt) {
            case 3: NextTxByte = txbufptr->mac[0]; break;
            case 4: NextTxByte = txbufptr->mac[1]; break;
            case 5: RadioTxState = TXSTATE_FLUSH; break;
            default: NextTxByte = ((char*)txbufptr)[TxByteCnt];
        }
    }
    if ( SECROSE_MAX_NODES < 255 ) {
        switch (TxByteCnt) {
            case 2: NextTxByte = txbufptr->mac[0]; break;
            case 3: NextTxByte = txbufptr->mac[1]; break;
            case 4: RadioTxState = TXSTATE_FLUSH; break;
            default: NextTxByte = ((char*)txbufptr)[TxByteCnt];
        }
    }

    }
    signal RadioSendCoordinator.byte(txbufptr, (uint8_t)TxByteCnt);
}
else {
    RadioTxState = TXSTATE_DONE;
}

break;

case TXSTATE_FLUSH:
    if (TxByteCnt > 3) {
        RadioTxState = TXSTATE_DONE;
    }
    break;

case TXSTATE_DONE:
    call SpiByteFifo.rxMode();
    call CC1000Control.RxMode();
    bTxPending = FALSE;
    if (post PacketSent()) {

```

```

        // If the post operation succeeds, goto Idle
        // otherwise, we'll try again.
        RadioState = IDLE_STATE;
    }
    break;
default:
    break;
}
}
break;

case PRETX_STATE:

{
    if (((data_in == (0xaa)) || (data_in == (0x55)))) {
        // Back to the penalty box.
        sMacDelay = (((call Random.rand() & 0xf) + 1) * (MSG_DATA_SIZE));
        RadioState = IDLE_STATE;
    }
    else if (bRSSIValid) {
        if (usRSSIVal > PRG_RDB(&CC1K_LPL_SquelchInit[lplpower])) {
            // ROCK AND ROLL!!!!
            call CC1000Control.TxMode();
            call SpiByteFifo.txMode();
            TxByteCnt = 0;
            RadioState = TX_STATE;
            RadioTxState = TXSTATE_PREAMBLE;
            NextTxByte = 0xaa;
            call SpiByteFifo.writeByte(0xaa);
        }
        else {
            // Russin frussin freakin frick o frack
            sMacDelay = (((call Random.rand() & 0xf) + 1) * (MSG_DATA_SIZE));
            RadioState = IDLE_STATE;
        }
    }
}
}
break;

case SYNC_STATE:
{
    uint8_t i;
    if (bInvertRxData) data_in = ~data_in;

    if ((data_in == 0xaa) || (data_in == 0x55)) {
        // It is actually possible to have the LAST BIT of the incoming
        // data be part of the Sync Byte. SO, we need to store that
        // However, the next byte should definitely not have this pattern.
        // XXX-PB: Do we need to check for excessive preamble?
        RxShiftBuf.MSB = data_in;
    }
}
}

```

```

}
else {
  uint16_t usTmp;
  switch (SOFCCount) {
  case 0:
    RxShiftBuf.LSB = data_in;
    break;

  case 1:
  case 2:
    // bit shift the data in with previous sample to find sync
    usTmp = RxShiftBuf.W;
    RxShiftBuf.W <<= 8;
    RxShiftBuf.LSB = data_in;

    for(i=0;i<8;i++) {
      usTmp <<= 1;
      if(data_in & 0x80)
        usTmp |= 0x1;
      data_in <<= 1;
      // check for sync bytes
      if (usTmp == SYNC_WORD) {
        if (rxbufptr->length !=0) {
          call Leds.redToggle();
          RadioState = IDLE_STATE;
        }
        else {
          RadioState = RX_STATE;
          call RSSIADC.getData();
          RxBitOffset = 7-i;
          signal RadioReceiveCoordinator.startSymbol(); // Time sync

          // reset some variables for proper packet reception
          j = 1;
          RxByteCnt = 0;
          ack_send = FALSE;
          handle_flag = TRUE;
        }
        break;
      }
    }
  }
  #if 0
  else if (usTmp == NSYNC_WORD) {
    RadioState = RX_STATE;
    RxBitOffset = 7-i;
    bInvertRxData = TRUE;
    break;
  }
  #endif
}
}

```

```

        break;

    default:
        // We didn't find it after a reasonable number of tries, so...
        RadioState = IDLE_STATE; // Ensures we wait till the end of the transmission
        break;
    }
    SOFCount++;
}

}
break;
case RX_STATE:
{
    if (bInvertRxData) data_in = ~data_in;
    RxShiftBuf.W <<=8;
    RxShiftBuf.LSB = data_in;

    Byte = (RxShiftBuf.W >> RxBitOffset);

    ((char*)rxbufptr)[RxByteCnt] = Byte;
    RxByteCnt++;

    signal RadioReceiveCoordinator.byte(rxbufptr, (uint8_t)RxByteCnt);

    if (handle_flag) { // we have just received the first byte. this is allways the length+flag
        handle_flag = FALSE;
        flag = call rose.readFlag((uint8_t)Byte); // find the overloaded flag
        Byte = call rose.fixFlag(rxbufptr); // remove the overloading
        rxbufptr->length = Byte; // set the correct length of data_in

        if ( flag == 0 ) { // this is a normal packet
            /* TODO: broadcast or discard packet (depending on route) */
            rxbufptr->addr = 0; // manually set the destination addr
        }

        else if (flag == 1) { // this is a broadcast packet
            /* tasks to implement:
            1. re-send packet
            */
            rxbufptr->addr = TOS_BCAST_ADDR;
            rxbufptr->source = 0; // only the base station might send broadcast pkts. source HAS to be 0.
        }

        else if (flag == 2) { // this is a long packet
            /* tasks to implement:
            1. forward packet (depending on route) */
        }
        if ( flag == 3 ) {

```

```

else { ; } // reserved, we should never reach here

// now determine how much is there to be received
handle_index = TRUE;
actual_data_len = rxbufptr->length; // length as reported
stream_len = header_len+actual_data_len+mac_len+1; // add header length and mac length

if ( rxbufptr->length == 1 ) actual_data_len++;
partial_stream_len = header_len+actual_data_len; // note the length before MAC
rec_stop = MSG_DATA_SIZE; // when to stop receiving data
} // end flag handling

if ( handle_index ) {
// Jumps regarding the packet headers, determined by the packet type and reported by the flag:
if ( (flag==0) && (RxByteCnt==3) ) { RxByteCnt += 2; handle_index = FALSE; } // normal packet, only has dest addr
if ( (flag==1) && (RxByteCnt==1) ) { RxByteCnt += 4; handle_index = FALSE; } // broadcast packet, does not have source or dest
if ( flag==3 ) { // control packet does not have type, group, source
    if ( (SECROSE_MAX_NODES > 255) && (RxByteCnt==3) ) {
        if ( rxbufptr->length == 0 ) { RxByteCnt = partial_stream_len; }
        else { RxByteCnt = DATA_START_NUMBER; }
        handle_index = FALSE;
    }
    if ( (SECROSE_MAX_NODES < 255) && (RxByteCnt==2) ) {
        if ( rxbufptr->length == 0 ) { RxByteCnt = partial_stream_len; }
        else { RxByteCnt = DATA_START_NUMBER; }
        handle_index = FALSE;
    }
}
}

// we have reached the end of headers+data. Now we need to jump to receive the mac
if ( RxByteCnt==partial_stream_len ) {
    RxByteCnt = MAC_POS;
    if ( flag == 3 ) { rec_stop-=2; } // MAC of control packets is 2 bytes less
}

if ( RxByteCnt==rec_stop ) { // just received the last byte. now we do the MAC validation etc
    if ( flag == 0 ) {
        rxbufptr->source = rxbufptr->addr; // rxbufptr->addr actually contains the source addr on normal pkts
        rxbufptr->addr = 0; // and the dest addr should be zero
    }
    else if ( flag == 1 ) {
        rxbufptr->source = 0;
        rxbufptr->addr = TOS_BCAST_ADDR;
    }
    else if ( flag == 2 ) { ; } // long packets. No need to do anything
    else if ( flag == 3 ) { ; } // control packets are better handled below
    else { ; } // reserved
}

```

```

// ACK Implementation (counterHandle() will do most of this for us)
if ( (flag==3) && (rxbufptr->addr==TOS_LOCAL_ADDRESS) ) {
    // update the key AND the rxbufptr->source field on data
    call rose.counterHandle(rxbufptr, ACK_TEST_START);
    RadioState = IDLE_STATE;
    RxByteCnt = 0;
    return 1;          // Control packet STOPS here.
}

// make a copy of rxbufptr to rxbufptr_copy
rxbufptr_copy = malloc(sizeof(TOS_Msg));
memcpy(rxbufptr_copy, rxbufptr, sizeof(TOS_Msg));

// decrypt with active key
call rose.packetEncDec(rxbufptr, DECRYPT);

// MAC validation, first set some vars, assuming that MAC is not valid
mac_valid = FALSE; ack_send = FALSE; rxbufptr->crc = 0;    // the crc is left for compatibility

mac_valid = call rose.validateMAC(rxbufptr); // attempt to validate

if ( !mac_valid ) {    // if MAC is not valid
    call rose.counterHandle(rxbufptr, CNT_REVERT);          // revert to backup key
    call rose.packetEncDec(rxbufptr_copy, DECRYPT);          // decrypt the copy of rxbufptr with backup key
    mac_valid = call rose.validateMAC(rxbufptr_copy);        // attempt to validate again
    if ( mac_valid ) { rxbufptr = rxbufptr_copy; }          // make rxbufptr point to rxbufptr_copy then
}
else { ack_send = TRUE; }    // first validation attempt was successful, so send ACK
// end of MAC validation. mac_valid now contains TRUE/FALSE info if MAC is valid or not

// respond to the result of MAC validation, send ACK etc
if ( mac_valid ) {
    if ( flag != 3 ) {    // was not a control packet, prepare for higher layers
        rxbufptr->crc = 1;    // left here for the shake of compatibility
    }
}
else { // MAC still not ok! (but we do nothing about it here, packetReceive() will discard the packet)
    dbg(DBG_USR1, "%i. Cannot validate MAC for received packet from node %i.\n", TOS_LOCAL_ADDRESS, rxbufptr->source);
}

free(rxbufptr_copy);
// end SecRose RX. Continue with CC1000 stuff

call SpiByteFifo.disableIntr();
RadioState = IDLE_STATE; //DISABLED_STATE;
rxbufptr->strength = usRSSIVal;

```



```

    if (!(post PacketRcvd())) {
        // If there are insufficient resources to process the incoming packet
        // we drop it
        rxbufptr->length = 0;
        RadioState = IDLE_STATE;
        call SpiByteFifo.enableIntr();
    }

    #if 0
    if (bTxPending) {
        sMacDelay = (((call Random.rand() & 0xf) + 1) * (MSG_DATA_SIZE));
    }
    #endif
}
}
break;

case DISABLED_STATE:
    break;

default:
    break;
}
}
}
break;

case DISABLED_STATE:
    break;

default:
    break;
}
}
}
break;

#if 0
RSSISampleFreq++;
RSSISampleFreq %= (preamblelen + 2 + offsetof(struct TOS_Msg, data) + 3) >> 1);
if ((RSSISampleFreq == 0) && (bTxPending || (RadioState == RX_STATE))) {
    call RSSIADC.getData();
}
}
#endif
return SUCCESS;
}

async event result_t RSSIADC.dataReady(uint16_t data) {
    //rxbufptr->strength = data;
    atomic {
        usRSSIVal = data;
        bRSSIValid = TRUE;
    }
    return SUCCESS;
}

// Default events for radio send/receive coordinators do nothing.
// Be very careful using these, you'll break the stack.
default async event void RadioSendCoordinator.startSymbol() { }
default async event void RadioSendCoordinator.byte(TOS_MsgPtr msg, uint8_t byteCount) { }
default async event void RadioReceiveCoordinator.startSymbol() { }
default async event void RadioReceiveCoordinator.byte(TOS_MsgPtr msg, uint8_t byteCount) { }
}

```

File SecRoseM.nc

```
module SecRoseM {
    provides interface SecRose;
    uses interface Leds as l;
}

implementation {

    uint32_t iKey[] = { 0x00A3D709, 0x0083F848, 0x00F6F4B3, 0x00211578, };
    uint32_t fKey[4] = { 0 };
    ROSE_counter counter[SECROSE_MAX_NODES+1]; // counters table
    // vars for acks
    ROSE_acks acks[SECROSE_MAX_ACK+1];
    uint8_t num_acks = 0;
    enum { ADD_VALUE, REMOVE_VALUE };
    // arrays for calcMAC() and calcACK() and btea. By allocating them here we use less memory
    uint8_t kalpha[8] = { 0 }; // key K1
    uint8_t kbeta[8] = { 0 }; // key K2
    uint8_t kbuffer[8] = { 0 }; // buffer for keys
    uint8_t v[56] = { 0 }; // temp v[] array to pass to btea.cipher(). Size value is not optimized
    uint8_t ack[2] = { 0 }; // temp array to store the last prepared ack for use by the ACK packet in a while

/* FUNCTIONS */
    // function to mix iKey+counter and produce fKey
    async command void SecRose.mixKey(uint32_t dest) {
        if ( dest == TOS_BCAST_ADDR ) {
            dest = SECROSE_MAX_NODES;
        }
        atomic fKey[0] = iKey[0] + counter[dest].active >> 24;
        atomic fKey[1] = iKey[0] + counter[dest].active >> 16;
        atomic fKey[2] = iKey[0] + counter[dest].active >> 8;
        atomic fKey[3] = iKey[0] + counter[dest].active >> 0;
    }

/* FUNCTION SEPARATOR */
    async command void SecRose.counterHandle(TOS_MsgPtr data, uint8_t action) {
        uint8_t flag = call SecRose.readFlag(data->length);
        uint8_t cur_counter = data->count_value;
        uint16_t node = 10000;
        uint8_t pos = 0;

        atomic {
            if ( action == PKT_SENT ) {
                if ( flag == 0 ) {
                    node = 0;
                }
            }
            else if ( flag == 1 ) {

```



```

        if ( num_acks >= SECROSE_MAX_ACK ) { num_acks--; }
        pos2 = num_acks;
        for (pos=(num_acks-1);pos<SECROSE_MAX_ACK;pos--) { // shift whole array 1 place to the right
            acks[pos2].ack[0] = acks[pos].ack[0];
            acks[pos2].ack[1] = acks[pos].ack[1];
            acks[pos2].addr = acks[pos].addr;
            pos2--;
        }
        // now set the first location
        acks[0].ack[0] = data->ack_value[0];
        acks[0].ack[1] = data->ack_value[1];
        acks[0].addr = node;
    }
    if ( action == REMOVE_VALUE ) {
        num_acks--; // by decreasing num_acks we cause the rest of the code to remove the oldest entry
        // shift table one place to the left after num_acks
        pos2 = node+1;
        for (pos=node;pos<=(num_acks+1);pos++) {
            acks[pos].ack[0] = acks[pos2].ack[0];
            acks[pos].ack[1] = acks[pos2].ack[1];
            acks[pos].addr = acks[pos2].addr;
            pos2++;
        }
    }
    return 0;
}
/* FUNCTION SEPARATOR */
// The top function in the encryption suite. Provides interfacing with the other components
async command result_t SecRose.packetEncDec(TOS_MsgPtr data, bool action) {
    uint16_t tmp = 0;
    uint8_t end, pos;

    data->pad_size = 0;    // make sure this is reset by default

    // 1. mix the key according to conditions (only if we got some data!)
    if ( data->length != 0 ) {
        if ( data->flag == 1 ) { tmp = TOS_BCAST_ADDR; }
        else {
            if ( action == ENCRYPT ) { tmp = data->addr; }
            if ( action == DECRYPT ) { tmp = data->source; }
        }
        call SecRose.mixKey(tmp);
    }
    else { return 1; } // when there is no data, do not encrypt anything

    // call steal + encrypt function
    call SecRose.stealEncDec(data, action);

    return SUCCESS;
}

```

```

/* FUNCTION SEPARATOR */
// Function to apply encryption and stealing.
async command result_t SecRose.stealEncDec(TOS_MsgPtr data, bool action) {
    uint8_t pos, pos2, stolen_mac_bytes, stolen_header_bytes;
    pos = pos2 = stolen_mac_bytes = stolen_header_bytes = 0;
    // add all data to v[]
    for (pos=0;pos<data->length;pos++) {
        atomic v[pos] = data->data[pos];
    }
    // complement v[] with data from mac[] untill the last 4-byte block is complete
    pos2 = 0;

    while ( (pos % 4) != 0 ) { // check last block size
        atomic v[pos] = data->mac[pos2];
        pos++; pos2++; stolen_mac_bytes++;
    }
    // for very small packets, add more data to v[] from mac, group, type or pad
    if ( (data->length+stolen_mac_bytes) < 8 ) {
        // not all 4 bytes of MAC are always used 10 lines above. Continue using them here. This code needs improvement/debug
        while ( (pos % 8) != 0 ) {
            atomic v[pos] = data->mac[pos2];
            pos++; pos2++; stolen_mac_bytes++;
            if ( stolen_mac_bytes == 4 ) break; // max MAC size is 4 bytes
        }
        // is the block complete?
        if ( (data->length+stolen_header_bytes+stolen_mac_bytes) < 8 ) {
            atomic v[pos++] = data->type; // no ? add type
            stolen_header_bytes++; // keep track of count to use later
        }
        // is the block complete?
        if ( (data->length+stolen_header_bytes+stolen_mac_bytes) < 8 ) {
            atomic v[pos++] = data->group; // not yet? then add group
            stolen_header_bytes++;
        }
        // still not complete? add a pad byte
        if ( (data->length+stolen_header_bytes+stolen_mac_bytes) < 8 ) {
            if ( action == ENCRYPT ) { // do not destroy pad byte when a packet is received
                atomic v[pos++] = 0; // when sending, pad byte is simply a 0
            }
            else { // on reception (decryption), the pad byte includes information
                atomic v[pos++] = data->data[data->length]; // so we use it. put it on v[]
            }
            stolen_header_bytes++;
            data->pad_size = 1;
        }
    }
}
// call btea section
if ( action == ENCRYPT ) { // encrypt or ...
    call SecRose.btea(v, pos, fKey);
}
if ( action == DECRYPT ) {

```

```

        call SecRose.btea(v, -pos, fKey);    } // ... decrypt
// now place the contents of v[] to appropriate fields of packet // put start of v[] to data[]
pos2 = 0;
for (pos=0;pos<data->length;pos++) {
    atomic data->data[pos] = v[pos2++];
}
for (pos=0;pos<stolen_mac_bytes;pos++) { // put middle of v[] to mac[]
    atomic data->mac[pos] = v[pos2++];
}
if ( stolen_header_bytes > 0 ) { // if there were bytes stolen from header
    atomic data->type = v[pos2++]; // replace type
    stolen_header_bytes--;
    if ( stolen_header_bytes > 0 ) { // replace group if more
        atomic data->group = v[pos2++];
        stolen_header_bytes--;
    }
    if ( stolen_header_bytes > 0 ) { // replace pad if even more
        atomic data->data[data->length] = v[pos2];
    }
}
return SUCCESS;
}
/* FUNCTION SEPARATOR */
async command void SecRose.btea(uint8_t* vs, int32_t n, uint32_t* ks) {
    int pos;
    call SecRose.bteaCipher(((uint32_t*)vs), n/4, ks);
}
/* FUNCTION SEPARATOR */
async command void SecRose.bteaCipher(uint32_t* vl, int32_t n, uint32_t* k) {
    uint32_t z, y=vl[0], sum=0, e, DELTA=0x9e3779b9;
    uint8_t n_minus_one; // optimisation
    uint32_t p, q;

    if (n > 1) { // Coding Part
        n_minus_one = n-1;
        q = SECROSE_TEA_CYCLES;
        z = vl[n_minus_one];
        while (q-- > 0) {
            sum += DELTA;
            e = (sum >> 2) & 3;
            for (p=0; p<n_minus_one; p++) {
                y = vl[p+1];
                z = vl[p] += (z>>5^y<<2) + (y>>3^z<<4)^(sum^y) + (k[p&3^e]^z);
            }
            y = vl[0];
            z = vl[n_minus_one] += (z>>5^y<<2) + (y>>3^z<<4)^(sum^y) + (k[p&3^e]^z);
        }
    }

    else if (n < -1) { // Decoding Part

```

```

n = -n;
n_minus_one = n-1;
q = SECROSE_TEA_CYCLES;
sum = q*DELTA;
z=v1[n_minus_one];
while (sum != 0) {
    e = (sum >> 2) & 3;
    for (p=n_minus_one; p>0; p--) {
        z = v1[p-1];
        y = v1[p] -= (z>>5^y<<2) + (y>>3^z<<4)^(sum^y) + (k[p&3^e]^z);
    }
    z = v1[n_minus_one];
    y = v1[0] -= (z>>5^y<<2) + (y>>3^z<<4)^(sum^y) + (k[p&3^e]^z);
    sum -= DELTA;
}

return;
}

/* FUNCTION SEPARATOR */
// Get the ack value from ack[] and place them on mac[] because MHSR reads from mac[] only
async command void SecRose.calcACK(TOS_MsgPtr data, uint16_t node) {
    uint8_t tmp;
    atomic data->mac[0] = ack[0];
    atomic data->mac[1] = ack[1];
}

/* FUNCTION SEPARATOR */
// Function to calculate the MAC of a packet.
// Implements: http://csrc.nist.gov/publications/nistpubs/800-38B/SP\_800-38B.pdf
async command result_t SecRose.calcMAC(TOS_MsgPtr data, uint16_t node, bool action) {
    uint8_t pos, pos2, end, flag; // temp and control variables
    bool condition;

    // mix the key for "node"
    if ( data->flag == 1 ) { call SecRose.mixKey(TOS_BCAST_ADDR); }
    else { call SecRose.mixKey(node); }
    flag = call SecRose.findFlag(data); // we need to know the flag, so that we know what to hash
    // reset v[]
    for(pos=0;pos<14;pos++) {
        ((uint32_t*)v)[pos] = 0;
    }
    // reset kbuffer[]
    for (pos=0; pos<2; pos++) ((uint32_t*)kbuffer)[pos] = 0;

    // fill v[] with what we want to hash
    atomic {
        // all packets, add data
        end = data->length;
        memcpy(v, data->data, data->length);
    }
}

```

```

// add .source.
if ( data->source > 255 ) { v[end] = ((uint8_t *)data)[3]; end++; }
v[end] = ((uint8_t *)data)[4]; end++;

// all add .type
v[end] = data->type; end++;

v[end] = data->group; end++;
} // end atomic

// Key set-up
// step 1: kbuffer = encrypt a block of 64 0's with key fKey
call SecRose.bteaCipher(((uint32_t*)kbuffer), 2, ((uint32_t*)fKey));

// step 2: (a) if the MSB of kbuffer is 0 then kalpha = (kbuffer << 1)
//          (b) else kalpha XOR (kbuffer << 1) with 0x1b
atomic {
    condition = (kbuffer[0] & 0x80); // keep a copy of the result on the check if MSB of kbuffer == 0

    // The code below does kalpha = (kbuffer << 1) (shitss the whole kbuffer one bit left)
    // This is needed in every case.
    // Also the code below fulfills step 1a
    pos = kbuffer[0] << 1;
    kalpha[0] = pos;
    for (pos=1; pos<8; pos++) {
        kalpha[pos-1] |= kbuffer[pos] >> 7;
        kalpha[pos] = kbuffer[pos] << 1;
    }

    // to fullfill step 2a we need to check the original MSB, we use our pre-obtained condition result
    // note: if condition is met then the contents of alpha assigned above are OVERWRITTEN here
    // this means that this might not be the most efficient way to do this
    if ( !condition ) {
        for (pos=0; pos<6; pos++) {
            kbuffer[pos] ^= 0x0; // bitwise XOR with 0 on first 7 bytes
            kalpha[pos] = kbuffer[pos]; // assign to kalpha
        }
        kbuffer[7] ^= 0x1b; // for last byte we XOR with 0x1b
        kalpha[7] = kbuffer[7]; // and assign
    }

    // step 3: (a) if the MSB of kalpha is 0 then kbeta = (kalpha << 1)
    //          (b) else kbeta XOR (kalpha << 1) with 0x1b
    //          In essence it's exactly as step 2 with "kalpha" replacing "kbuffer" and
    //          "kbeta" replacing "kalpha". The rest of code remains the same. See above for comments
    condition = (kalpha[0] & 0x80);

    pos = kalpha[0] << 1;

```



```

kbeta[0] = pos;
for (pos=1; pos<8; pos++) {
    kbeta[pos-1] |= kalpha[pos] >> 7;
    kbeta[pos] = kalpha[pos] << 1;
}

if ( !condition ) {
    for (pos=0; pos+1<7; pos++) kalpha[pos] ^= 0x0; kbeta[pos] = kalpha[pos];
    kalpha[7] ^= 0x1b; kbeta[7] = kalpha[7];
}
} // end atomic

// use
// Step 1: if the last block is a full block then XOR it with kalpha
atomic {
    if ( (end % 8) == 0 ) { // check last block's size
        pos2 = 0;
        for (pos=(end-8); pos<end; pos++) {
            v[pos] ^= kalpha[pos2]; // do the XOR
        }
    }

    // (b) else append 1 set bit after the last byte and 7 unset bits.
    else {
        v[end] = 0x80; end++; // append 10000000 to v[end]
        while ((end % 8) != 0 ) { // while v[]'s last block is not a complete block,
            v[end] = 0; end++; // append 0's to it untill it is
        }
        // and XOR the last block of v[] with kbeta
        pos2 = 0;
        for (pos=(end-8); pos<end; pos++) {
            v[pos] ^= kbeta[pos2];
            pos2++;
        }
    }
} //end atomic

// Step 4: XOR and encrypt the whole v[] (in blocks) using:
// block of v[] for plaintext
// kbuffer to store the result of encryption
// fKey as key
// use 0's for the first time (when kbuffer is not filled)

pos2 = 0; // pos on kbuffer[]
for (pos=0; pos<end; pos++) {
    if ( pos < 8 ) {
        atomic kbuffer[pos2] = 0 ^ v[pos]; // initialisation, use 0's instead of kbuffer[]'s value
    }
    else {

```

```

        atomic kbuffer[pos2] ^= v[pos];          // step 4a
    }
    pos2++;
    if ( pos2 == 8 ) {
        pos2 = 0;
        call SecRose.bteaCipher(((uint32_t*)v), 2, ((uint32_t*)fKey));
    }
}

// Step 5: return the calculated MAC
// assign first 4 bytes of kbuffer in the mac position of the packet
for (pos=0;pos<SECROSE_MAC_LENGTH;pos++) atomic data->mac[pos] = kbuffer[pos];

// assign last byte of kbuffer in the count_value position of the packet
atomic data->count_value = kbuffer[SECROSE_MAC_LENGTH];
if ( data->count_value == 0 ) { data->count_value++; }

// if this is a packet that is going to be sent, store the ACK in data->ack_value[].
// this will then be used by counterHandle()
if ( action == SEND ) {
    atomic data->ack_value[0] = kbuffer[5];
    atomic data->ack_value[1] = kbuffer[6];
}
else { // if it is a received packet store it to ack[] to be used by calcACK() in a while, when the ACK packet will be prepared
    atomic ack[0] = kbuffer[5];
    atomic ack[1] = kbuffer[6];
}

return 1;
}
/* FUNCTION SEPARATOR */
async command bool SecRose.validateMAC(TOS_MsgPtr data) {
    uint8_t mac_pos;          // a temp counter var
    uint8_t mac_tmp[SECROSE_MAC_LENGTH]; // for storing the mac

    // First make a copy of the received MAC value
    for (mac_pos=0;mac_pos<SECROSE_MAC_LENGTH;mac_pos++) mac_tmp[mac_pos] = data->mac[mac_pos];

    call SecRose.calcMAC(data, data->source, RECV);          // calculate what the MAC should be based on the received data

    // compare MAC's
    if ( (mac_tmp[0]==data->mac[0]) && (mac_tmp[1]==data->mac[1]) && (mac_tmp[2]==data->mac[2]) && (mac_tmp[3]==data->mac[3]) ) {
        return TRUE;
    }
    else {
        dbg(DBG_USR1, "%i. MAC Mismatch\n", TOS_LOCAL_ADDRESS);
        dbg(DBG_USR1, ".");
        return FALSE;
    }
}

```

```

}
/* FUNCTION SEPARATOR */
// Determines and returns the flag of a packet based on the .addr field of it
// if it is a control packet pending to be sent it will determine the flag by the .flag value
// Also, this function writes the flag down in the data->flag location of the packet
async command uint8_t SecRose.findFlag(TOS_MsgPtr data) {
    uint8_t flag = 5;

    if ( data->flag == 3 ) { flag = 3; }
        // control
    else {
        if ( data->addr == 0 ) { flag = 0; }
            // normal
        if (data->addr == TOS_BCAST_ADDR) { flag = 1; } // broadcast
        if ( (data->addr < TOS_BCAST_ADDR) && (data->addr > 0) ) { flag = 2; } // long
        data->flag = flag;
    }
    return flag;
}
/* FUNCTION SEPARATOR */
// determines the flag and overloads the .length value to include the flag
// it also updates the .flag field
async command uint8_t SecRose.writeFlag(TOS_MsgPtr data) {
    if ( data->flag == 3 ) { // control packet
        data->length |= 0xC0; // flip both the first and second bits of data->length by OR-ing with 0xC0
    }
    else { // all other packets
        if ( data->addr == 0 ) { data->flag = 0; } // normal packet means the first two bits will remain unchanged,
        // so, just set the data->flag field and nothing else

        else if (data->addr == TOS_BCAST_ADDR) { // Broadcast packet
            data->length |= 0x40; // flip second bit of data->length by OR-ing with 0x40 (= 01000000)
            data->flag = 1;
        }
        else { // Long packet
            data->length |= 0x80; // flip the first bit of data->length by OR-ing with 0x80 (= 10000000)
            data->flag = 2;
        }
    }
    return data->flag;
}
/* FUNCTION SEPARATOR */
// determines the flag from the overloaded .length value, returns the flag
async command uint8_t SecRose.readFlag(uint8_t data) {
    /* normal packet defaults to do nothing */
    if (data < 64) { return 0; }
    /* Broadcast packet for lengths of the range 64 - 127 */
    if ((data >= 64) && (data <= 127)) { return 1; }
    /* Long packet for lengths in the range 128 - 191 */
}

```

```

        if ((data >= 128) && (data <= 191)) { return 2; }
        /* Control packet for lengths in the range 192 - 255 */
        if ( data >= 192 ) { return 3; }
        return 5;
    }
/* FUNCTION SEPARATOR */
// determines the flag from the overloaded .length value, fixes the overloading and returns the flag
async command uint8_t SecRose.fixFlag(TOS_MsgPtr data) {
    /* normal packet defaults to do nothing */
    if (data->length < 64) {
        data->flag = 0;
        return data->length;
    }
    /* Broadcast packet for lengths of the range 64 - 127 */
    if ((data->length >= 64) && (data->length <= 127)) {
        data->flag = 1;
        return (data->length)-64;
    }
    /* Long packet for lengths in the range 128 - 255 */
    if ((data->length >= 128) && (data->length <= 191)) {
        data->flag = 2;
        return (data->length)-128;
    }
    /* Control packet for lengths in the range 192 - 255 */
    if ( data->length >= 192 ) {
        data->flag = 3;
        return (data->length)-192;
    }
    return 5;
}
} // end implementation

```

REFERENCES

1. Akyildiz, I.F., et al., *Wireless sensor networks: a survey*. Computer Networks, 2002. **38**.
2. Karlof, C., N. Sastry, and D. Wagner. *TinySec: A Link Layer Security Architecture for Wireless Sensor Networks*. in *Second ACM Conference on Embedded Networked Sensor Systems (SensSys 2004)*. 2004.
3. Menezes, A.J., P.C.V. Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*. 1996: CRC Press LLC. ISBN 0-8493-8523-7.
4. Schneier, B., *Applied Cryptography Second Edition*. 1996 ISBN 0-471-12845-7 John Wiley & Sons.
5. Schneier, B., *Schneier on Security*. 2008: Wiley
6. Schneier, B., *Secrets & Lies: Digital Security in a Networked World*. 2000: John Wiley & Sons, Inc. 304.
7. Dierks, T. and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. 2008; Available from: IETF Network Working Group. RFC5246. <http://tools.ietf.org/html/rfc5246>.
8. Postel, J. *SPECIFICATION OF INTERNET TRANSMISSION CONTROL PROGRAM*. 1981; Available from: IETF Network Working Group. RFC791. <http://tools.ietf.org/html/rfc791>.
9. Postel, J., *RFC 793: Transmission control protocol*. 1981, September.
10. IEEE. *802.15.4-2006*. 2006; Available from: <http://standards.ieee.org/getieee802/802.15.html>.
11. *Crossbow Technology Inc*. <http://www.xbow.com>
12. Mainwaring, A., et al. *Wireless sensor networks for habitat monitoring*. in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*. 2002. Atlanta, Georgia, USA: ACM.
13. Li, Y., Z. Wang, and Y.Q. Song. *Wireless sensor network design for wildfire monitoring*. in *6th World Congress on Intelligent Control and Automation (WCICA 2006)*. 2006. Dalian (China).
14. Martinez, K., J.K. Hart, and R. Ong, *Environmental sensor networks*. IEEE Computer, 2004. **37**(8): p. 50-56.
15. Wark, T., et al., *Transforming Agriculture through Pervasive Wireless Sensor Networks*. IEEE Pervasive Computing, 2007. **6**(2): p. 50-57.
16. Gamage, C., et al. *Security for the Mythical Air-Dropped Sensor Network*. in *Proceedings of the 11th IEEE Symposium on Computers and Communications*. 2006.
17. Karp, B. and H.T. Kung. *GPSR: greedy perimeter stateless routing for wireless networks*. in *International Conference on Mobile Computing and Networking 2000*. Boston, Massachusetts, United States ACM Press.
18. Intanagonwiwat, C., R. Govindan, and D. Estrin. *Directed diffusion: a scalable and robust communication paradigm for sensor networks*. in *International Conference on Mobile Computing and Networking 2000*: ACM Press.
19. Heinzelman, W.R., A. Chandrakasan, and H. Balakrishnan. *Energy-Efficient Communication Protocol for Wireless Microsensor Networks*. in *Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8 - Volume 8*. 2000: IEEE Computer Society.
20. Ren, K., W. Lou, and Y. Zhang, *LEDS: Providing Location-Aware End-to-End Data Security in Wireless Sensor Networks*. IEEE TRANSACTIONS ON MOBILE COMPUTING, 2008.
21. Luk, M., et al. *MiniSec: a secure sensor network communication architecture*. in *Proceedings of the 6th international conference on Information processing in sensor networks 2007*. Cambridge, Massachusetts, USA: ACM.
22. Yoon, S., C. Veerarittiphan, and M.L. Sichitiu, *Tiny-sync: Tight time synchronization for wireless sensor networks*. ACM Transactions on Sensor Networks 2007. **3**(2).
23. Du, W., et al. *A pairwise key pre-distribution scheme for wireless sensor networks*. in *Conference on Computer and Communications Security*. 2003. Washington D.C., USA: ACM Press.
24. Ganesan, D., et al. *Highly-resilient, energy-efficient multipath routing in wireless sensor networks*. in *International Symposium on Mobile Ad Hoc Networking & Computing 2001*. Long Beach, CA, USA
25. Chen, B., et al., *Span: an energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks*. Wireless Networks, 2002. **8**(5): p. 481 - 494.
26. Çam, H., et al. *Energy-efficient security protocol for wireless sensor networks*. in *IEEE VTC Fall 2003*. 2003. Orlando, USA.
27. Jolly, G., et al. *A Low-Energy Key Management Protocol for Wireless Sensor Networks*. in *Proceedings of the 8th IEEE International Symposium on Computers and Communication*. 2003: IEEE.

28. Walters, J.P., et al., *Wireless sensor network security: A survey.*, in *Security in Distributed, Grid, Mobile, and Pervasive Computing*. 2006, Auerbach Publications, CRC Press. Available from: <http://www.cs.wayne.edu/~weisong/papers/walters05-wsn-security-survey.pdf>.
29. Perrig, A., et al., *SPINS: security protocols for sensor networks*. *Wireless Networks*, 2002. **8**(5): p. 521 - 534.
30. Li, T., et al., *SenSec Design. Tech. Rep.* 2005, Institute for Infocomm Research: Singapore. p. 15.
31. Levis, P., et al., *TinyOS: An Operating System for Sensor Networks*, in *Ambient Intelligence*. 2005. p. 115-148. Available from: http://dx.doi.org/10.1007/3-540-27139-2_7.
32. Kahn, J.M., R.H. Katz, and K.S.J. Pister. *Next century challenges: mobile networking for "Smart Dust"*. in *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. 1999. Seattle, Washington, United States: ACM.
33. Perrig, A., J. Stankovic, and D. Wagner, *Security in wireless sensor networks*. *Communications of the ACM*, 2004. **47**(6): p. 53 - 57.
34. Wartena, F., J. Muskens, and L. Schmitt. *Continua: The Impact of a Personal Telehealth Ecosystem*. in *International Conference on eHealth, Telemedicine, and Social Medicine, 2009. eTELEMED '09*. 2009.
35. BP. *Condition Monitoring*. 2009 5/2009; Available from: http://www.xbow.com/General_info/Info_pdf_files/BP_Intel_Sensor_Networks.wmv.
36. Shamir, A., *How to share a secret*. *Commun. ACM*, 1979. **22**(11): p. 612-613.
37. Eschenauer, L. and V.D. Gligor. *A key-management scheme for distributed sensor networks*. in *Conference on Computer and Communications Security*. 2002. Washington, DC, USA ACM Press.
38. Huang, Q., et al. *Fast Authenticated Key Establishment Protocols for Self-Organizing Sensor Networks*. in *2nd ACM international conference on Wireless sensor networks*. 2003: ACM.
39. Liu, D. and P. Ning. *Establishing pairwise keys in distributed sensor networks*. in *Conference on Computer and Communications Security*. 2003. Washington D.C., USA: ACM Press.
40. Intel. *Sensor Networks Research 2009* [cited 2009; Available from: <http://techresearch.intel.com/articles/Exploratory/1501.htm>.
41. Huang, S.-C., et al. *Comfort of air conditioning using sensor networks with game theory*. 2007; Available from: <http://dspace.lib.fcu.edu.tw/bitstream/2377/3589/1/ce07ics002006000135.pdf>.
42. Rabaey, J.M., et al., *PicoRadio Supports Ad Hoc Ultra-Low Power Wireless Networking*. *Computer*, 2000. **33**(7): p. 42-48.
43. Teumim, D.J., *Industrial Network Security*. 2nd ed. 2004: ISA.
44. Milenkovic, A., C. Otto, and E. Jovanov, *Wireless sensor networks for personal health monitoring: Issues and an implementation*. *Computer Communications*, 2006. **29**(13-14): p. 2521-2533.
45. *Data Protection Act*. 1998: UK. http://www.opsi.gov.uk/acts/acts1998/ukpga_19980029_en_2#pt1-11g2
46. Atmel, *ATmega 128 & ATmega 128L. 8-bit AVR microcontroller with 128K Bytes in-system programmable flash*. 2004.
47. Crossbow, *Imote2 Datasheet*, in *High-performance Wireless Sensor Network Node*. 2010, Crossbow Technology. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/Imote2_Datasheet.pdf
48. Crossbow, *MICA2 Datasheet*, in *Wireless Measurement System*. 2010, Crossbow Technology. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf
49. Krishnamachari, B. and S. Iyengar, *Distributed Bayesian Algorithms for Fault-Tolerant Event Region Detection in Wireless Sensor Networks*. *IEEE Trans. Comput.*, 2004. **53**(3): p. 241-250.
50. Estrin, D., et al., *Next century challenges: scalable coordination in sensor networks*, in *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. 1999, ACM: Seattle, Washington, United States. p. 263-270.
51. Sandhu, J.S. *Wireless Sensor Networks for Commercial Lighting Control: Decision Making with Multi-agent Systems*. in *AAAI Workshop on Sensor Networks*. 2004.
52. Pottie, G.J. and W.J. Kaiser, *Wireless integrated network sensors*. *Commun. ACM*, 2000. **43**(5): p. 51-58.
53. David Gay, P.L., David Culler, Eric Brewer. *nesC 1.1 Language Reference Manual*. 2003; Available from: <http://nesc.sourceforge.net/papers/nesc-ref.pdf>.
54. Diffie, W. and M. Hellman, *New Directions In Cryptography*. *IEEE Transactions on Information Theory*, 1976. **22**(6): p. 644 - 654.
55. Lai, B.C., et al. *Reducing Radio Energy Consumption of Key Management Protocols for Wireless Sensor Networks*. in *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*. 2004.
56. Bhatti, S., et al., *MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms*. *Mobile Networks and Applications*, 2005. **10**(4): p. 563-579.
57. Dunkels, A., B. Gronvall, and T. Voigt, *Contiki - a lightweight and flexible operating system for tiny networked sensors*. 29th Annual IEEE International Conference on Local Computer Networks, 2004, 2004: p. 455- 462.

58. Beutel, J., et al., *Prototyping Wireless Sensor Network Applications with BTnodes*, in *Wireless Sensor Networks*. 2004. p. 323-338. Available from: <http://www.springerlink.com/content/51anjx6utqy7e8q7>.
59. Crossbow Technology Inc; Available from: <http://www.xbow.com>.
60. Khemapech, I., A. Miller, and I. Duncan, *Simulating Wireless Sensor Networks*
61. Ekonomou, E. and K. Booth. *Simulating Sensor Networks*. in *6th Annual Conference on the Convergence of Telecommunications, Networking & Broadcasting (PGNet)*. 2005. Liverpool / UK: Liverpool John Moores University.
62. Levis, P., et al. *TOSSIM: accurate and scalable simulation of entire tinyOS applications*. in *Conference On Embedded Networked Sensor Systems* 2003. Los Angeles, California, USA
63. Titzer, B.L., D.K. Lee, and J. Palsberg. *Avrora: Scalable Sensor Network Simulation with Precise Timing*. in *4th international symposium on Information processing in sensor networks*. 2005: IEEE Press Piscataway, NJ, USA.
64. Berkeley, U. *TinyOS*. 2004; Available from: <http://www.tinyos.net>.
65. Kulkarni, S., A. Iyer, and C. Rosenberg, *An address-light, integrated MAC and routing protocol for wireless sensor networks*. *IEEE/ACM Transactions on Networking*, 2006. **14**(4): p. 793-806.
66. ITU. *Open Systems Interconnection - Basic Reference Model: The basic model*. 1994; Available from: <http://www.itu.int/rec/T-REC-X.200-199407-I/en>.
67. IEEE. *LAN/MAN Wireless LANS*. 2009; Available from: <http://standards.ieee.org/getieee802/802.11.html>.
68. Juang, P., H. Oki, and Y. Wang. *Energy Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet*. in *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2002. San Jose, CA.
69. Hill, J., et al., *System Architecture Directions for Networked Sensors*. *ACM SIGPLAN Notices*, 2000. **35**(11): p. 93 - 104.
70. ARM. *ARM7TDMI*. Available from: <http://www.arm.com/products/CPUs/ARM7TDMI.html>.
71. NIST. *Recommendation for Key Management – Part 1: General (Revised)*. 2007; Available from: http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf.
72. Lenstra, A.K. and E.R. Verheul, *Selecting Cryptographic Key Sizes*, in *Proceedings of the Third International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography*. 2000, Springer-Verlag. p. 446-465.
73. CNSS, *CNSS Policy No. 15, Fact Sheet No. 1 - National Policy on the Use of the Advanced Encryption Standard (AES) to Protect National Security Systems and National Security Information*. 2003.
74. Watro, R., et al. *TinyPK: securing sensor networks with public key technology*. in *Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*. 2004. Washington DC, USA: ACM Press.
75. Bellare, M., J. Kilian, and P. Rogaway, *The security of the cipher block chaining message authentication code*. *Journal of Computer and System Sciences*, 2000. **61**(3): p. 362-399.
76. Karlof, C. and D. Wagner. *Secure routing in wireless sensor networks: Attacks and countermeasures*. in *First IEEE International Workshop on Sensor Network Protocols and Applications*. 2003.
77. Douceur, J.R. *The Sybil Attack, Revised*. in *Papers from the First International Workshop on Peer-to-Peer Systems*. 2002.
78. Hu, Y.-C., A. Perrig, and D.B. Johnson, *Wormhole detection in wireless ad hoc networks*. 2002(Tech. Rep. TR01-384).
79. Xu, W., et al., *The feasibility of launching and detecting jamming attacks in wireless networks*, in *Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing*. 2005, ACM: Urbana-Champaign, IL, USA. p. 46-57.
80. Wood, A.D. and J.A. Stankovic, *Denial of service in sensor networks*. *IEEE Computer*, 2002. **35**(10): p. 54–62.
81. Deng, J., R. Han, and S. Mishra, *Enhancing Base Station Security in Wireless Sensor Networks*. *Technical Report CU-CS-951-03*. 2003, Department of Computer Science, University of Colorado.
82. Yang, H., et al. *Toward resilient security in wireless sensor networks*. in *Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing*. 2005. Urbana-Champaign, IL, USA.
83. Zhu, S., S. Setia, and S. Jajodia, *LEAP+: Efficient security mechanisms for large-scale distributed sensor networks*. *ACM Transactions on Sensor Networks* 2006. **2**(4): p. 500-528.
84. Pietro, R.D., L.V. Mancini, and A. Mei, *Energy efficient node-to-node authentication and communication confidentiality in wireless sensor networks* *Wireless Networks*, 2006. **12**(6): p. 709-721.
85. Zhang, Y., et al., *Location-based compromise-tolerant security mechanisms for wireless sensor networks*. *IEEE Journal on Selected Areas in Communications*, 2006. **24**(2): p. 247-260.

86. Yu, Y., R. Govindan, and D. Estrin, *Geographical and Energy Aware Routing: A Recursive Data Dissemination Protocol for Wireless Sensor Networks*. UCLA Computer Science Department Technical Report UCLA/CSD-TR-01-0023, 2001.
87. Xu, Y., J. Heidemann, and D. Estrin. *Geography-informed energy conservation for ad hoc routing*. in *Proceedings of the Seventh Annual ACM/IEEE International Conference on Mobile Computing and Networking*. 2001.
88. Braginsky, D. and D. Estrin. *Rumor routing algorithm for sensor networks*. in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*. 2002. Atlanta, Georgia, USA ACM Press.
89. Manjeshwar, A. and D. Agrawal. *TEEN: A routing protocol for enhanced efficiency in wireless sensor networks*. in *1st International Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing*. 2001.
90. Kömmerling, O. and M. Kuhn. *Design Principles for Tamper-Resistant Smartcard Processors*. in *SENIX Workshop on Smartcard Technology*. 1999. Chicago, Illinois, USA.
91. Hess, E., et al. *Information Leakage Attacks Against Smart Card Implementations of Cryptographic Algorithms and Countermeasures*. in *EUROSMART Security Conference*. 2000.
92. Skorobogatov, S.P., *Semi-invasive attacks - A new approach to hardware security analysis*, in *Computer Laboratory*. 2005, University of Cambridge: Cambridge.
93. Casado, L. and P. Tsigas, *ContikiSec: A Secure Network Layer for Wireless Sensor Networks under the Contiki Operating System*, in *Proceedings of the 14th Nordic Conference on Secure IT Systems: Identity and Privacy in the Internet Age*. 2009, Springer-Verlag: Oslo. p. 133-147.
94. Jinwala, D., D. Patel, and K. Dasgupta, *FlexiSec: A Configurable Link Layer Security Architecture for Wireless Sensor Networks*. *Journal of Information Assurance and Security*, 2009. **4**: p. pp 582-603, ISSN 1554-1010.
95. Zia, T. and A. Zomaya. *A Secure Triple-Key Management Scheme for Wireless Sensor Networks*. in *Proceedings of the 25th IEEE International Conference on Computer Communications*. 2006.
96. Jinwala, D., D. Patel, and K. Dasgupta, *Optimizing the Block Cipher Resource Overhead at the Link Layer Security Framework in the Wireless Sensor Networks*. *Lecture Notes in Computer Science*, 2008. **5352/2008**.
97. Koo, W., et al. *Implementation and analysis of new lightweight cryptographic algorithm suitable for wireless sensor networks*. 2008.
98. National Institute of Standards and Technology. *SKIPJACK and KEA Algorithm Specifications*. 1998.
99. Dworkin, M. *Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*. 2005; Available from: http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf.
100. Rogaway, P., M. Bellare, and J. Black, *OCB: A block-cipher mode of operation for efficient authenticated encryption*. *ACM Trans. Inf. Syst. Secur.*, 2003. **6**(3): p. 365-403.
101. Bloom, B., *Space/time trade-offs in hash coding with allowable errors*. *Communications of the ACM*, 1970.
102. Sun, K., et al. *TinySeRSync: Secure and resilient time synchronization in wireless sensor networks*. in *ACM CCS*.
103. Ganeriwal, S., et al. *Secure time synchronization service for sensor networks*. in *WiSe*. 2005.
104. Kilian, J. and P. Rogaway. *How to protect DES against exhaustive key search*. in *Advances in Cryptology*. 1996: Springer-Verlag.
105. ZigBee Alliance. *Zigbee specification*. 2005.
106. Wood, A.D., et al. *SIGF: A Family of Configurable, Secure Routing Protocols for Wireless Sensor Networks*. in *The Fourth ACM Workshop on Security of Ad Hoc and Sensor Networks*. 2006. Alexandria, VA, USA.
107. Becher, A., Z. Benenson, and M. Dornseif, *Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks* in *Lecture Notes in Computer Science*. 2006, Springer Berlin / Heidelberg. p. 104-118.
108. Suh, G.E., et al. *AEGIS: architecture for tamper-evident and tamper-resistant processing*. in *Proceedings of the 17th annual international conference on Supercomputing*. 2003. San Francisco, CA, USA: ACM.
109. Gu, W., et al. *Defending against search-based physical attacks in sensor networks*. in *IEEE International Conference on Mobile Adhoc and Sensor Systems*. 2005.
110. Agah, A., M. Asadi, and S.K. Das. *Prevention of DoS Attack in Sensor Networks using Repeated Game Theory*. in *Proceedings of the 2006 International Conference on Wireless Networks*. 2006. Las Vegas, Nevada, USA.

111. McCubbin, C.B., A.A. Selcuk, and D. Sidhu. *Initialization vector attacks on the IPsec protocol suite*. in *In 9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2000)*. 2000: IEEE.
112. *U.S. Code collection*. Available from: <http://www.law.cornell.edu/uscode/44/3542.html>.
113. Needham, R.M. and D.J. Wheeler. *TEA Extensions*. in *Technical Report, Computer Laboratory, University of Cambridge, Cambridge, MA*. 1997.
114. Daemen, J., *Cipher and Hash Function Design, Strategies Based on Linear and Differential Cryptanalysis*. 1995, Katholieke Universiteit Leuven. Ph. D. Thesis. Sections 2.5.1 and 2.5.2.
115. GNU, *GNU General Public Licence version 3*, GNU, Editor. 2007. <http://www.gnu.org/licenses/gpl.html>
116. Rivest, R.L., A. Shamir, and L.M. Adleman. *U.S. Patent 4405829 (RSA)*. 1977; Available from: <http://www.google.com/patents?vid=4405829>.
117. Rivest, R.L. *The RC5 Encryption Algorithm*. 1997; Available from: <http://people.csail.mit.edu/rivest/Rivest-rc5rev.pdf>.
118. Rivest, R.L., et al., *The RC6 Block Cipher (Version 1.1; August 20, 1998)*. Available at <http://people.csail.mit.edu/rivest/Rc6.pdf>, 1998.
119. Schneier, B. *Description of a new variable-length key, 64-bit block cipher (Blowfish)*. 1993: Springer.
120. Schneier, B. *Twofish*. 1998; Available from: <http://www.schneier.com/twofish.html>.
121. TripleDES. *NIST Data Encryption Standard (DES)*. 1999; Available from: <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>.
122. Daemen, J. and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*. 2002: Springer-Verlag. ISBN 3-540-42580-2.
123. Wheeler, D. and R. Needham. *TEA, a tiny encryption algorithm*. 1995 [cited 2004 10/11/04]; Available from: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.281>.
124. Guimaraes, G., et al., *Evaluation of Security Mechanisms in Wireless Sensor Networks*, in *Proceedings of the 2005 Systems Communications*. 2005, IEEE Computer Society. p. 428-433.
125. Jinwala, D., D. Patel, and K. Dasgupta, *Investigating and Analyzing the Light-weight ciphers for Wireless Sensor Networks*. 2009(available online: <http://www.dcc.ufla.br/infocomp/artigos/v8.2/art05.pdf>).
126. Rinne, S., T. Eisenbarth, and C. Paar. *Performance Analysis of Contemporary Light-Weight Block Ciphers on 8-bit Microcontrollers*. in *3rd International Symposium on Industrial Embedded Systems — SIES 2008*. 2008.
127. Rinne, S., T. Eisenbarth, and C. Paar. *Performance Analysis of Contemporary Light-Weight Block Ciphers on 8-bit Microcontrollers*. in *SPEED 2007* 2007. Amsterdam.
128. Law, Y., J. Doumen, and P. Hartel, *Survey and benchmark of block ciphers for wireless sensor networks*. *ACM Transactions on Sensor Networks (TOSN)*, 2006. **2**(1): p. 93.
129. Shepherd, S.J., *The Tiny Encryption Algorithm*. *Cryptologia*, 2007. **31**(3): p. 233-245.
130. Kessler, G.C. *An Overview of Cryptography*. 2004 04/11/2004; Available from: <http://www.garykessler.net/library/crypto.html>.
131. Frankel, S. and H. Herbert. *The AES-XCBC-MAC-96 Algorithm and Its Use With IPsec*. 2003 [cited 2008 July]; Available from: <http://www.rfc-archive.org/getrfc.php?rfc=3566>.
132. NIST. *National Institute of Standards and Technology*. 2010; Available from: <http://www.itu.int/rec/T-REC-X.200-199407-I/en>.
133. GNU and Atmel. *AVR32 GNU Toolchain*. Available from: http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4118.
134. Lamport, L., *Time, clocks, and the ordering of events in a distributed system*. *Commun. ACM*, 1978. **21**(7): p. 558-565.
135. Postel, J., *Internet protocol*. 1981, STD 5, RFC 791, September 1981.
136. Wheeler, D. and R. Needham, *Correction to xtea*. Unpublished manuscript, Computer Laboratory, Cambridge University, England, 1998.
137. Yarrkov, E., *Cryptanalysis of XXTEA*. 2010.
138. Douglas, M., *Alkaline single cell batteries and rechargers: results of preliminary tests*.
139. Plummer, D., *An Ethernet address resolution protocol*. 1982, STD 37, RFC 826, MIT.
140. Postel, J., *Internet control message protocol*. 1981, September.
141. Biham, E., A. Biryukov, and A. Shamir, *Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials*. *J. Cryptol.*, 2005. **18**(4): p. 291-311.
142. Ye, F., et al. *A scalable solution to minimum cost forwarding in large sensor networks*. in *Tenth International Conference on Computer Communications and Networks*. 2001.
143. Lindsey, S. and C. Raghavendra. *PEGASIS: Power-efficient gathering in sensor information systems*. in *IEEE Aerospace Conference*. 2002.

144. Xu, Y., J. Heidemann, and D. Estrin. *Energy conservation by adaptive clustering for ad-hoc networks*. in *Poster Session of MobiHoc 2002*. 2002.
145. Xu, Y., J. Heidemann, and D. Estrin, *Adaptive energy-conserving routing for multihop ad hoc networks*. Research Report 527, USC/Information Sciences Institute, 2000.
146. Shnayder, V., et al. *Simulating the power consumption of large-scale sensor network applications*. in *2nd international conference on Embedded networked sensor systems*. 2004. Baltimore, MD, USA: ACM.
147. Han, C., et al., *A dynamic operating system for sensor nodes*.
148. Ganeriwal, S., L. Balzano, and M. Srivastava, *Reputation-based framework for high integrity sensor networks*. ACM Transactions on Sensor Networks (TOSN), 2008. **4**(3): p. 15.
149. Landsiedel, O., K. Wehrle, and S. Gotz. *Accurate Prediction of Power Consumption in Sensor Networks*. in *The Second IEEE Workshop on Embedded Networked Sensors, 2005. EmNetS-II*. 2005: IEEE.