Norwegian University of Science and Technology

# Secure data aggregation for wireless sensor network

**Trang Tran-Thi-Thuy**

Master in Security and Mobile Computing
Submission date: June 2010
Supervisor: Danilo Gligoroski, ITEM

Norwegian University of Science and Technology
Department of Telematics

# Problem Description

Smartdust the company produces perimeter guarding systems, consisting of wireless sensors. Currently, the sensors have an 8-bit processor (with no specific functionality for cryptography), 8kB RAM and 4kB ROM. Any node may possibly want to talk to any other node. Currently the traffic between sensors is not protected.

Smartdust, however, would like to ensure the _integrity_ of data reported by sensors. The task of the proposed MSc project is to basically design and implement a solution for authenticating the data produced by sensors. The thesis work should result in:

* an overview of possible solutions. Consider solutions for securing point-to-point communications, as well as solutions based on group key agreement. Keep in mind the restricted computational capabilities.
* Selection / design of a particular solution and its implementation.
* Testing and analysis of the implemented solution.
* If possible: Smartdust's next steps in developing perimeter guarding systems.

Assignment given: 15. January 2010
Supervisor: Danilo Gligoroski, ITEM

# Abstract

Like conventional networks, security is also a big concern in wireless sensor networks. However, security in this type of networks faces not only typical but also new challenges. Constrained devices, changing topology or susceptibility to unprecedented security threats such as node capture and node compromise has refrained developers from applying conventional security solutions into wireless sensor networks. Hence, developing security solutions for wireless sensor networks not only requires well security analysis but also offers a low power and processing consuming.

In this thesis, we implemented security solution targeting IRIS sensor motes. In our implementation, a public key-based key exchange is used to establish shared secret keys between sensor nodes. These secret keys are used to provide authenticity, integrity and freshness for transmission data. Our implementation ensures the flexibility in integrating our solution with available TinyOS operating system. Additionally, the thesis work also focuses on evaluating the performance in wireless sensor networks both in memory and energy consuming.

# Acknowledgements

First, I would like to express my gratitude and heartiest thanks toward my home and host supervisors: Professor Danilo Gligoroski from Norwegian University of Science and Technology and Professor Antti Ylä-Jääski from Aalto University School of Science and Technology for giving me opportunity to work on this thesis. Especially, I would like to thank wholeheartedly to my instructor : Professor Peeter Laud from University of Tartu for his collaboration, inspiring discussion and realistic suggestion at all stages of my thesis work. Additionally, I owe my gratitude to Raido Pahtma and Jurgo Preden who kindly offers helps in my thesis work. Without their support, this work would not have been completed.

Secondly, I would like to thank all my friends in TKK for keeping me up during my depressed thesis work.

Last but not least, I am indebted with my parents who have been there with me all the time. Their never ending love and support beyond boundaries of field and geography have encouraged me to pursue my interest. Especially, I would like to thank my brother Huy Tran Quoc has been a patient and valuable advice giver.

Espoo June 29th, 2010

Trang Tran Thi Thuy

# Abbreviations and Acronyms

| | |
|---|---|
| AES | Advanced Ecryption Standard |
| CA | Certificate Authority |
| ECC | Elliptic Curve cryptogrpahy |
| ECDH | Elliptic Curve Diffie Hellman |
| ECDSA | Elliptic Curve Digital Signature Algorithm |
| FIFO | First In First Out |
| MAC | Message Authentication Code |
| MTU | maximum transmission unit |
| nesC | network embedded system C |
| PKI | Public Key Infrastructure |
| RSA | Rivest, Shamir and Adleman |
| STS | Station-to-Station |
| TripleDES | Triple Data Encryption Standard |
| WSN | Wireless Sensor Network |

# Contents

**Abstract** i

**Acknowledgements** i

**Abbreviations and Acronyms** iii

**1 Introduction** 1

1.1 Motivation . . . 1

1.2 Two-tier security problem in Wireless Sensor Network . . . 2

1.2.1 Message confidentiality, authenticity and integrity . . . 3

1.2.2 Key management problem . . . 3

1.3 Thesis outline . . . 5

1.3.1 Thesis goal . . . 5

1.3.2 Thesis structure . . . 5

**2 Background** 6

2.1 Cryptography primitives . . . 6

2.1.1 Symmetric cryptography . . . 6

2.1.2 Public key cryptography . . . 8

2.2 TinyOS and nesC programming language . . . 11

2.2.1 TinyOS - sensor node operating system . . . 11

2.2.2 nesC - programming language for sensor node . . . 13

2.3 Related work . . . 13

2.3.1 Key management proposals . . . 13

2.4 Authenticity, integrity and confidentiality solution . . . . . . . 16

**3 Design 18**

3.1 Platform . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.2 Design Goals . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.2.1 Security Goal . . . . . . . . . . . . . . . . . . . . . . . . 20
3.2.2 Performance . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.2.3 Ease of use . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.3 Protocol Design . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.3.1 Our security model . . . . . . . . . . . . . . . . . . . . . 22
3.3.2 Key management module . . . . . . . . . . . . . . . . . . 23
3.3.3 Packet handling module . . . . . . . . . . . . . . . . . . . 28

**4 Implementation 32**

4.1 Key exchange module . . . . . . . . . . . . . . . . . . . . . . . 32
4.1.1 Architecture description . . . . . . . . . . . . . . . . . . 33
4.2 Packet handling module . . . . . . . . . . . . . . . . . . . . . . 36
4.2.1 Architecture description . . . . . . . . . . . . . . . . . . 37
4.2.2 Transparency solution . . . . . . . . . . . . . . . . . . . . 38

**5 Evaluation 40**

5.1 Experiment platform . . . . . . . . . . . . . . . . . . . . . . . . 40
5.2 Key Exchange module . . . . . . . . . . . . . . . . . . . . . . . 40
5.2.1 TinyECC optimization . . . . . . . . . . . . . . . . . . . . 40
5.2.2 Memory usage . . . . . . . . . . . . . . . . . . . . . . . . 43
5.2.3 Energy consumption . . . . . . . . . . . . . . . . . . . . . 45
5.3 Packet handling module . . . . . . . . . . . . . . . . . . . . . . 49
5.3.1 Execution time . . . . . . . . . . . . . . . . . . . . . . . . 49
5.3.2 Memory consumption . . . . . . . . . . . . . . . . . . . . . 50

**6 Conclusion and Future work 51**

6.1 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

6.2 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

# List of Tables

2.1 Rule A and Rule B . . . . . . . . . . . . . . . . . . . . . . . . . 8

3.1 Computational overhead between ECDH and EC-based STS . 26

3.2 Security strength comparison between ECDH and EC-based STS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

5.1 Memory overheads. All figures are in bytes . . . . . . . . . . . 45

5.2 Memory usage between *Key exchange module* and the whole system. All the figures are in bytes. . . . . . . . . . . . . . . . . 45

5.3 Energy consumption for Full message key exchange . . . . . . 48

5.4 Energy consumption for Fragmented message key exchange . . 49

5.5 Memory consumption in Packet handling module. . . . . . . . 50

# List of Figures


2.1 CBC-MAC operation . . . . . . . . . . . . . . . . . . . . . . . . 8

3.1 IRIS sensor mote from Crossbow . . . . . . . . . . . . . . . . . 19
3.2 IEEE 802.15.4 packet format . . . . . . . . . . . . . . . . . . . . 19
3.3 Security architecture . . . . . . . . . . . . . . . . . . . . . . . . 22
3.4 Certificate format . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.5 Key exchange protocol . . . . . . . . . . . . . . . . . . . . . . . 28

4.1 Key Exchange module decomposition . . . . . . . . . . . . . . 33
4.2 Key Exchange module decomposition - Fragmented exchange message version . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.3 Packet fragmentation . . . . . . . . . . . . . . . . . . . . . . . . 36
4.4 Packet Handling architecture overview . . . . . . . . . . . . . 37
4.5 Encapsulation of packet through Packet handling layer . . . . 38

5.1 Experiment platform . . . . . . . . . . . . . . . . . . . . . . . . 41
5.2 Comparing TinyECC optimization . . . . . . . . . . . . . . . . 42
5.3 Energy experiment setup . . . . . . . . . . . . . . . . . . . . . . 46
5.4 Power consumption *Full message key exchange* with *execution time optimization* . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.5 Power consumption for *Full message key exchange* with *memory optimization.* . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.6 Power consumption for *Fragmented message key exchange* with *execution time optimization.* . . . . . . . . . . . . . . . . . . . . 47

5.7 Power consumption for *Fragmented message key exchange* with *memory optimization*. . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.8 Time spent for sending one packet . . . . . . . . . . . . . . . . . . 50

# Chapter 1

# Introduction

## 1.1 Motivation

Wireless sensor network — an emergent application of computer systems — has gradually attracted a lot of attention due to its broad application potential. As a heterogeneous network, wireless sensor network consists of low-cost constrained devices, namely *sensor motes*, responsible for sensing environmental events such as temperature, pressure or lighting. This data later is transmitted to a back end server where it is processed and information about state of monitored environment is deduced. This transmission happens through hop-by-hop wireless communication where collaboration between sensor nodes is required in order to finally forward sensor readings to the destination. Wireless communication alleviates the workload of sensor node deployment since sensor mote can be easily placed at any places without worrying about wiring of the network. Such easy deployment together with low cost in implementing sensor network, thank to inexpensive devices, has broadened the applicability of wireless sensor networks. Wireless sensor network's applications range from civil applications such as habitat monitoring, environmental observation to critical military applications such as a battlefield surveillance or intrusion detection application.

Due to the increasing usage of wireless sensor networks, security also becomes a growing concern in this type of network. The properties of a wireless sensor network make it more susceptible to certain types of attackers, compared to wired network . Wireless communication channel is one of those properties. As a broadcast communication media, wireless channel is easily intercepted. The adversary can inject packets into communication stream without difficulties or direct all traffic into *bogus node* and thus control the

data stream of the whole network. As the adversary can filter traffic at his will, a seized network can provide false information about the environment state. Additionally, bogus sensor node can intentionally tear down routing protocol, partition network or increase end-to-end latency. Another property that makes wireless sensor networks is more vulnerable than wired networks is non-attendance. As wireless sensor network is usually remotely controlled and operates under no observation, the adversary can easily occupy any sensor node and read data from sensor node. This so-called node capture attack can lead to reveal of security information which probably nullifies the whole security system. For example, if the whole sensor network shares only one secret key, capturing one sensor node can compromise all other sensor nodes. Furthermore, constrained devices create their own problems for wireless sensor network. As powered by batteries, sensor node needs to save energy to gain longer life which refrains sensor nodes from using complex computations. Security solution thus has to be redefined to match sensor node limitations. That's the reason why many security mechanisms in conventional networks are inapplicable or require modification and serious testing in order to be implemented in wireless sensor networks.

## 1.2 Two-tier security problem in Wireless Sensor Network

Like conventional networks, message confidentiality, authenticity and integrity are also major concerns in wireless sensor networks. In order to address those concerns, conventional network , such as Internet, uses symmetric cryptographies. A disadvantage in symmetric cryptography is that two communicating parties have to share the same secret before they can actually securely talk to each other. This requirement is not a problem for Internet network where communicating parties can transfer shared secrecy out-of-channel or more popular solution is using asymmetric cryptography. Asymmetric cryptography as a solution to problem of exchanging shared secret keys is widely deployed in conventional networks with many security protocols such as SSH, SSL or IPSec. However, applying such computationally complex cryptographic suite for wireless sensor network seems not a straight forward solution since sensor nodes are famous for not only their limitation in processing ability but their memory and energy constraints as well. Hence, solving key management problem is as important as ensuring message confidentiality, authenticity and integrity.

### 1.2.1 Message confidentiality, authenticity and integrity

Wireless sensor networks share the same problem with normal wireless networks as message transmission is carried over-the-air. The problem is that this type of communication is subject to eavesdropping and interference. Nevertheless, wireless sensor networks are mainly used for sensing environmental conditions, so the adversaries can easily deduce that transferred data in wireless sensor networks are mainly environmental measurements. This explains why preventing eavesdropping is not the main security concern in wireless sensor networks. In fact, wireless sensor network security is more against interference. Inserting malicious information, into broadcast channel can cause big impact on accuracy of the whole network. Moreover, the attacker can even use incorrect sensor reading to cause severe consequences. For example, injecting false information in a perimeter intruder detection system can direct security guards away from the correct location of intrusion.

Protecting message transmission from eavesdropping or altering has been a well-researched subject among security researchers.Together with the arrival of higher processing power computers, security solutions become stronger and stronger. More and more complex computation need cryptographic protocols and primitives that are stronger and stronger. However, the case is not the same for wireless sensor networks. While the usage of the same complex cryptographic solution is impossible, other selected security solutions for wireless sensor network should not be easily broken by using common high processing power computers.

Due to the efficiency of symmetric cryptography which provides the same security level as asymmetric cryptography but requires less computational power, it becomes the key security solution for wireless sensor networks. Nevertheless, in the context of these networks, symmetric cryptography brings out another problem, namely *key management.*

### 1.2.2 Key management problem

In order to prevent malicious nodes from sending false information, only pairs of legitimate nodes which share secret keys with each other can exchange information. Secret keys are the only way to differentiate legitimate nodes from malicious nodes. Thus, *key management* involves generating and distributing secret keys among sensor network and also taking care of revoking key and re-keying. Solutions to *key management* in wireless sensor network are divided into two categories: key pre-distribution and online-key generation.

In the context of wireless sensor network, each of these key management has it own advantages and disadvantages.

Key pre-distribution enables sensor nodes by storing shared secret key before being deployed in to the field. Compared to online-key generation, this method alleviates computation overhead for sensor nodes which in turn saves energy consumption. However, this method shows some disadvantages for wireless sensor networks. Firstly, in order to be able to communicate with each and every sensor node in network, a sensor node has to store all shared keys with each and every node. This means the number of total shared secret keys for network increases quadratically with the number of nodes in network. Storing such a large number of keys causes memory problems in sensor nodes. Knowing topology of network in advance can help reducing number of needed key since only two nodes that can speak to each other should save their shared key. However, this case only plays a little role in wireless sensor network scenarios since most topology of wireless sensor network is constructed during deploying phase or network topology is also subject to change. E.g one sensor node happens not to work as normal any more or some nodes are added to the network. Second disadvantage and also the biggest problem with key pre-distribution is re-keying. Because keying phase happens before deploying phase, the only way to re-keying all nodes is to transfer secret keys into sensor nodes again. If one nodes is exposed, all nodes sharing secret with this node need to be re-keyed.

On the other hand, on-line key generation solves the problem of key pre-distribution. This key management solution allows sensor nodes to establish mutual secret key on-the-fly when needed. This solution doesn't require sensor nodes to store secret key in advance and even works well in unknown topologies, as sensor nodes only need to store their own information in order to generate secret keys. Since this information is private to each sensor node, exposing one node only has local impact on that specific node. The other advantage of this method is ability to re-keying on-line. Key are generated on-the-fly, it's easy to generate another key if old keys are exposed. In order to achieve this, public key cryptography is the most notable solution. However, public key cryptography is famous for its complex computation. This means a drawback of this method is heavy energy consumption. Thus, searching for an efficient way to choose and implement public key cryptography is a problem that need to be solved when deploying security for wireless sensor networks.

## 1.3 Thesis outline

### 1.3.1 Thesis goal

The goal of this thesis is to design and implement a complete security solution for wireless sensor network. This security solution has two aspects, that respectively solve aforementioned tow-tier security problem in wireless sensor networks. First, design and implement a security solution which provides authenticity, integrity and freshness. Second, design and implement a suitable key exchange protocol which establishes secret keys between every two sensor nodes. This security solution is targeted towards the sensor network platform of Defendec [2] that uses Iris motes [4] and TinyOS operating system [30]. So security solution needs to work on Iris mote and can be integrated into TinyOS operating system. Ability to run on other motes is optional. The implementation needs to be lightweight and stable.

### 1.3.2 Thesis structure

The thesis describes an overall security solution for wireless sensor networks including sending message in an authenticated manner (integrity and authenticity checking) and solution to securely establish keys between two arbitrary nodes.

**Chapter 2**Description about the thesis background is presented. First, we describe cryptographic primitives used to perform secure communication and secure key exchange. Additionally, this chapter also give an overview about TinyOS operating system and nesC programming language. The end of chapter 2 discusses other security proposals for wireless sensor node.

**Chapter 3**Description about design goal, a security design for secure communication and a security design for key exchange are presented in this chapter. Moreover, this chapter also discusses why these security designs are chosen to implement.

**Chapter 4**Implementation issues are covered in this chapter.

**Chapter 5**Performance of our implementation are quantified. Both energy and memory consumptions are addressed in this chapter.

**Chapter 6**Conclusion of the thesis is provided and so are some future work suggestions.

# Chapter 2

# Background

This chapter presents some fundamentals related to our thesis work. As the background to our security solution, we start this section with some cryptographic primitives. The symmetric and asymmetric cryptography we aim to use in our security solution is discussed in general. Description about selected algorithms are given in more detail. Next, we describe an overview of the operating system TinyOS and programming language nesC which are used in our sensor motes. Finally, related work introduces some prominent security proposals and available solutions for wireless sensor networks.

## 2.1 Cryptography primitives

Originated from the ancient time, cryptography is always a strong choice everytime human want secret communication. As developing together with human's need, modern cryptography is classified into two categories: *symmetric cryptography* and *asymmetric cryptography* or also called *public key cryptography*. Possessing exclusively different properties, these two categories are usually used together in many security solution in order to achieve the most reasonable and effective result.

### 2.1.1 Symmetric cryptography

As the oldest cryptographic method, symmetric cryptography has come into use since the birth of cryptography. The basic concept of symmetric cryptography is that if two communicating parties want to talk in private, they needs to share the same secret key and use that key to perform modification

on conversational data. Symmetric cryptography is used for many purposes including providing data secrecy using encryption and decryption or ensuring data integrity and authenticity by constructing *Message authentication code.*

### Encryption algorithm

Encryption algorithms are used to transform transmitted message from understandable format , namely *plaintexts* , into *ciphertexts* which are unreadable to any uninvolved parties. In symmetric encryption, senders encrypt plaintexts using secret keys which later also is used by receivers to decrypt ciphertexts. Symmetric encryption algorithms can be classified into two different types : *stream cipher* and *block cipher.* These two types of cipher differ in the way that plaintexts are encrypted. While stream ciphers process input messages bit-by-bit, block ciphers operate on invariable large block of input messages. Some examples of stream ciphers are Linear feedback shift register [14], RC4 [36] and some well-known block ciphers are DES [18], AES [19] or Blowfish [38].

***SkipJack*** [16] is a NSA-developed 32-round block cipher using 80-bit key on 64-bit data blocks. As a NSA invention, the block cipher was classified by NSA until 1998. In SkipJack, every 64-bit block first is divided into four 16-bit words $w_1$, $w_2$, $w_3$ and $w_4$ and then go through 32 rounds of either Rule A or Rule B. The first eight rounds use Rule A, then followed by 8 rounds of Rule B , then followed by 8 rounds of Rule A and finalized by 8 rounds of Rule B. Rule A and Rule B can be described as in Table 2.1.1. G function is a Feistel function which permutes 16-bit words using 80-bit key. Used to be a NSA application, SkipJack has been well examined and considered secure by a group of independent cryptographers in 1993 [17]. The most noticeable weakness of SkipJack is short key size – 80-bit key size compared with AES 128, 192 or 256 bit; otherwise SkipJack is secure [16], has a good performance and is wireless sensor network friendly [29].

### Message authentication code

Message authentication code(MAC) is a type of data used to provide authenticity and integrity of its corresponding message. There are two types of algorithms used to calculate MAC : *hash-based MAC* such as HMAC [27] or *Cipher-based MAC* as CBC-MAC [15].

CBC-MAC is an MAC-constructing algorithm using a block cipher $E$ in CBC mode with *zero initialization vector* $\boldsymbol{0}$ . Suppose that we need to calculate

| Rule A | Rule B |
|---|---|
| $w_1^{k+1} = G^k(w_1^k) \oplus w_4^k \oplus counter^k$<br>$w_2^{k+1} = G^k(w_1^k)$<br>$w_3^{k+1} = w_2^k$<br>$w_4^{k+1} = w_3^k$ | $w^{k+1} = w_4^k$<br>$w_2^{k+1} = G^k(w_1^k)$<br>$w_3^{k+1} = w_1^k \oplus w_2^k \oplus counter^k$<br>$w_4^{k+1} = w_3^k$ |
| $w_i^k$ output word from round $k$, $i = 1, 2, 3, 4$<br>$G^k()$ keyed permutation function<br>*counter* starts at 32 and decrements by one after every round | |

Table 2.1: Rule A and Rule B

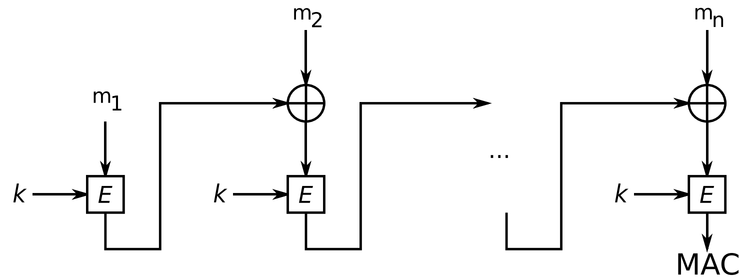


Figure 2.1: CBC-MAC operation

a message $m$ with secret key $k$. The message $m$ is split into block size of cipher $E$ $m = m_1||m_2||...||m_x$. Then these blocks are used as input into cipher $E$ in CBC mode as in Figure 2.1. CBC-MAC is widely known to to be insecure toward variable length message. If the adversary happens to know two plaintexts and their corresponding MACs, he can forge a legitimate MAC for a new message. However, prepending the message length to the message is suggested to solve this weakness [15].

### 2.1.2 Public key cryptography

Contrary to symmetric cryptography, public key cryptography does not require the sender and the receiver to share any secret information beforehand. In fact, each end possesses a pair of keys : *private key* and *public key*. While private key is kept secretly to the owner, public key is publicly known to every parties. The sender uses private key to encrypt or sign message while the receiver uses public key to decrypt or verify received message.

### Elliptic curve cryptography

Elliptic curve cryptopraphy (ECC) is a domain for defining public-key cryptosystems. The domain was first suggested by Neal Koblitz and Victor S.Miller in 1985 [23, Chap. 3] . An elliptic curve is a collection of points $(x, y) \in \mathbf{K}^2$, where $\mathbf{K}$ is a field, satisfying the equation:

$$y^2 = x^3 + ax + b$$

together with an extra point $O$ that could be thought of as the point in infinity. We say that elliptic curve $E$ over field $\mathbf{K}$, denote $E/\mathbf{K}$.

Let two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ in $E/\mathbf{K}$ be given, $P$ and $Q$ are two distinct and not negative of each other. The sum of two points $P$ and $Q$ is $R = P + Q = (x_3, y_3)$ where

$$x_3 = (\frac{y_2-y_1}{x_2-x_1})^2 \text{ and } y_3 = (\frac{y_2-y_1}{x_2-x_1})(x_1 - x_3) - y_1$$

Together with addition operation just defined, all of points on the Elliptic curve with $O$ as its identity form an abelian group. The proof can be found in [23, Sec. 3.1.2]. The scalar multiplication $KP$ can be calculated as $kP = P + P + ... + P$ ($k$ times).

Elliptic curve cryptosystems make use of elliptic curves defined either over $GF(p)$ or $GF(2^m)$. Prime domain has parameters $(p, a, b, G, n, h)$ while binary domain has parameter $(m, f(x), a, b, G, n, h)$ (from now on, $(E, a, b, G, n, h)$ is used as presentation of one of these two domains) where

- $p$, $a$, $b$ in prime domain defines elliptic curve $y^2 = x^3 + ax + b$ over field $GF(p)$.

- $m$ ,$f(x)$, $a$ and $b$ in binary domain defines elliptic curve $y^2 = x^3 + ax + b$ over field $GF(2^m) = F_2(x)/f(x)$ with $f(x)$ is an irreducible polynomial.
- $G$ base point or generator. The set of $\{0, G, 2G, ..\}$ is a cyclic group and can be used analogously to subgroup of $Z_{p^*}$ in cryptographic construction.
- $n$ order of G so that $nG = O$.
- $h$ the cofactor $h = \#E(F_p)/n$ .

### Signature scheme

Signature is a new functionality which only exists in public key cryptography, not in symmetric cryptography. Signature is a piece of information ensuring authenticity of messages between two parties without any shared secret information in advance. The sender creates the signature by using his private key while the receiver verifies a signature by using the sender's public key. One of the most well-known signature schemes is Elliptic curve digital signature algorithm ECDSA [23, Sec. 4.4.1] as follows :

- **Set up**
  Let an Elliptic curve domain be given, $(E, a, b, G, n, h)$ and a hash function $H$. If sensor mote A wants to send an authenticated message to sensor mote B. Sensor mote A randomly generates private key $d_A \in [1, n-1]$, public key $Q_A = d_A G$ and mesage $m$. From now on, something (e.g. taking inverses) happens *modulo* $n$, is denoted $\pmod n$.

- **Signing**
  1. Compute $e = H(m)$.
  2. Select $k \in [1, n-1]$.
  3. Compute $r = x_1 \bmod n$ with $(x_1, y_1) = kG$. If $r = 0$, return to previous step.
  4. Compute $s = k^{-1}(e + rd_A) \pmod n$. if $s = 0$, go back to step 2.

  Signature of message $m$ is $(r, s)$.

- **Verification**
  1. Check whether $r$ and $s$ are integers within $[1, n-1]$. If not, return fail. Otherwise continue.
  2. Compute $e = H(m)$.
  3. Compute $w = s^{-1} \pmod n$
  4. Compute $u_1 = ew \pmod n$ and $u_2 = rw \pmod n$
  5. Compute $(x_1', y_1') = u_1 G + u_2 Q_A$.
  6. Check whether $r = x_1' \pmod n$. If yes, signature is valid; Otherwise invalid

- **Proof** If $(r, s)$ is legitimate on message $m$, we have $s = k^{-1}(e + d_A r) \pmod n$. Rearrange this equation, we have :

  $$k \equiv s^{-1}(e + d_A r) \equiv s^{-1}e + s^{-1}rd_A \equiv we + wrd_A \equiv u_1 + u_2 d_A \pmod n$$

  Thus $(x'_1, y'_1) = u_1 G + u_2 Q_A = (u_1 + u_2 d_A)G = kG$, and so $r = x'_1 \pmod n$

#### Key exchange scheme

Key exchange scheme in public key cryptography allows two entities to mutually agree upon the same secret key without any shared prior knowledge. As one of the most well-known key exchange scheme, Elliptic Curve Diffie Hellman (ECDH) is the Elliptic curve version of discrete logarithm-based Diffie Hellman key exchange. It can be described as follows :

Let an Elliptic curve domain be given, $(E, a, b, G, n, h)$. Two sensor motes A and B want to mutually establish shared secret key. Each sensor mote, A and B, picks a private key $d_A$ and $d_B$ ( randomly generated numbers between $[1, n-1]$). From $d_A$ and $d_B$ , they can deduce their corresponding public key $Q_A = d_A G$ and $Q_B = d_B G$. So in order to generate shared key, mote A has to send its own public key $Q_A$ to B and B, in return, has to reply with its own public key $Q_B$. After both sides receive each other's public key, while mote A can compute $k = d_A Q_B$, and mote B can compute $k = d_B Q_A$. Shared secret is derived from the $x$ coordinate of point $k$.

Among key exchange schemes, ECDH is the simplest one which performs mutual key establishment in the fastest way. Nevertheless, this scheme does contains several security vulnerabilities such as : no *key freshness* (only one key is established during entities' lives) or no entity authentication ( no mechanism to clearly verify whether either of two sensor motes really participated in key exchange).

## 2.2 TinyOS and nesC programming language

### 2.2.1 TinyOS - sensor node operating system

Being constrained devices, sensor motes require lightweight operating systems which are designed to focus on reliable low-power operations. TinyOS [30], a collaboration between University of California, Berkeley and Intel Research and Crossbow Technology, is the most popular operating system for

wireless sensor motes. TinyOS design goals are to provide a robust system which can operate with limited resources, support various platforms and wide range of applications.

In order to achieve these goals, TinyOS operating system is designed as a set of *components* with an objective to increase the flexibility of the whole system. As separate modules, components are certainly easier to develop, use and replace. The effortless component replacing is attained by defining the common *interface* between interacting components. Interfaces are the set of functions which specify the communication rules between two different components. Developers can replace or switch two components at ease if these two follows exactly the same interface. Furthermore, interfaces and components design model also makes module replacing or interchanging transparent to developers.

Besides the ability of switching between dissimilar software implementations, TinyOS also provides ability to switch between software implementations and hardware devices. For example, software implementations and hardware devices of a cryptography service should be interchangeable. As hardware is mainly developed with *split-phase* operation model other than *blocking* in other operating system. In blocking operations, a function calling is returned only when its whole operation is completed. In split-phase operation, a function calling, on the other hand, is returned immediately. The actual function operation still runs to completion and then a separate callback function is signalled. For example, an application call a function to request radio chip to send a message. This function is returned immediately while the radio chip continues sending message. When sending is finished, radio chip issues an interrupt to signal the application about the completed job. Hence, in order to provide compatibility with hardware components, many TinyOS components are developed with split-phase model.

Another issue in hardware and software components interchanging is the internal concurrency inside hardware components. For example our radio chip above, the radio chip with internal processor is capable of signalling the interrupt while the main processor is busy performing other operations. However, all software implementations use the same core processor which is allowed to perform only one operation at a time. To solve this problem, TinyOS introduce *task model*. Every time an application wants to perform an operation, it posts a task into a FIFO *task queue*. By splitting a whole operation of a software component into several short tasks and posting into task queue, TinyOS allows interleaving different software components' tasks inside task queue and more importantly it allows different components to

virtually run in concurrency. To programmer, a task is no more than a deferred procedure which is ensured by TinyOS operating system to run in the near future but not immediately.

### 2.2.2 nesC - programming language for sensor node

nesC (netword embedded system C) [22] is mainly used for developing application for TinyOS operating system. As mainly used for network sensor motes, nesC is extended from C programming language to be able to provide codes targeting a specific platform and low-level functionalities interacting with underlying hardwares. Unlike C programs, in nesC language, separate modules are not allowed to compile individually. Instead, the whole nesC programs are compiled altogether so that it can get overall analysis and optimization. Moreover, aim to work on constrained devices, nesC program is a "static language" as it provides no dynamic memory allocation at runtime and fixed call-graphs at compile time. These constraints not only make TinyOS easier in managing memory but also reduce difficulties and uncertainties in program analysing. As designed for TinyOS operating system, nesC language provides features that helps programming in TinyOS such as supporting components concept, task implementation or concurrency model.

## 2.3 Related work

This section describes some prominent researches in security for wireless sensor network. Researches in this area mainly focuses on solving either key management problem or authenticity, integrity and confidentiality problem.

### 2.3.1 Key management proposals

As one of the main problem in wireless sensor network security, key management schemes have been widely studied for the last several years. Due to various constraints both in sensor motes and the whole network, there hasn't been an efficient key management scheme for all type of WSNs. In fact, every scheme is designed based on several assumptions about structures, goals or even deployment environment of WSNs. However, key management protocols for WSNs can be divided into two categories : key pre-distribution and online key generation.

### Key pre-distribution

A key management scheme is categorized as key pre-distribution if secret keys are already loaded into sensor motes before deployment. Hence, this scheme causes no communication overhead to WSNs as the shared keys are already somehow established and stored inside each sensor motes. As network size prevents sensor motes from storing all per-link keys, researchers has proposed several scheme in order to reduce the size of key collection.

The very first key pre-distribution was suggested by Eschenauer and Gligor [21], namely *Random key Pre-distribution Scheme.* In this scheme, instead of storing all per-link keys, each sensor note is only loaded with a *key ring* — a subset which is randomly drawn from a common large key pool $S$. After deployment, sensor motes go through *shared key discovery stage* to discover other peers sharing common keys. Two sensor motes sharing the same key can use that key to communicate securely. If two sensor motes having no common key want to exchange messages, they have to rely on other parties who shares common keys with both of them. These trusted third parties generate new keys and distributes them to two sensor motes. This procedure is called *path key establishment stage.* The advantage of this algorithm is that it clearly helps sensor motes by reducing key storage size while still maintains a reasonable theoretical connectivity. Eschenauer and Gligor have proved that [21] with a key pool $S$ of 10000 keys, each sensor mote is required to store only 75 keys wit probability that every two sensor mote share common key is 50%. However, the connectivity of WSNs is physically reduced by many other practical reasons such as deployment area or weather condition. Moreover, this proposal provides a weak resilience against *node capture.* If a sensor node is compromised, all other sensor nodes sharing common keys with that node are exposed also.

In order to improve resilience of previous scheme, Chan and Perrig [11] proposed a modified version called *The q-composite Random Key Pre-distribution Scheme.* The difference between this scheme and the previous one is that instead of only 1 key, q keys are required between two sensor motes to be able to communicate securely. Certainly, this scheme enhances the network resilience against node capture since attackers need to capture more sensor motes in order to possibly break a certain link. However, in return, the key ring size gets bigger to ensure the same connectivity probability.

Chan and Perrig [11] also proposed *Random pairwise scheme.* In this scheme , a certain number of pairwise keys are generated and loaded to corresponding sensor motes. Hence, only two sensor motes that are stored with a pairwise key can communicate. As not all per-link keys are pre-distributed, only a

fraction of sensor motes can communicate and unconnected pair of sensor motes can communicate through trusted third sensor motes. The advantage of this scheme is that pairwise keys are only loaded into their corresponding sensor motes, thus if one sensor mote is captured, communication among the rest sensor motes are still safe. This scheme however provides a weak scalability and causes difficulties in adding new sensor motes into already deployed network.

In general, key pre-distribution schemes promise to provide a little processing overhead since key already stored in every sensor motes. Nevertheless, they are usually susceptible to node capture attacks. Additionally, it is difficulty to add new sensors into network or re-key compromised sensor nodes because every sensor node has to be loaded with new keys if new sensor nodes join network.

Another branch of key pre-distribution scheme is that rather than storing directly secret keys, sensor motes can be loaded with a secret information which is used to generate shared secret key. One simple example of this type is *Master key based pre-distribution scheme* proposed by Lai and Kim [28]. In this scheme, a master key is stored to every sensor node. This master key later is used together with random number generated by sensor motes to established shared secret key.

Alternatively, another proposal by Liu and Ning [32], namely *Polynomial-based key pre-distribution scheme*, uses polynomials in order to generate shared secret key. These polynomials are in form of a bivariate t-degree function $f(x, y) = \sum_{i,j=0}^{t} a_{ij} x^i y^j$ with $a_{ij} = a_{ji}$ over a finite field $G(q)$ where q is a large enough prime number. Since $a_{ij} = a_{ji}$, $f(x)$ satisfies $f(x, y) = f(y, x)$. Subsequently, a sensor node ID $A$ can produce a shared key with a sensor node ID $B$ by substitute $A$, $B$ into the polynomial, that is, $f(A, B)$. On sensor node ID B side, the same operation is repeated to generate the shared key $f(B, A)$. As special property of polynomial $f(x)$, we have $f(A, B) = f(B, A)$, hence sensor node A and B are now carrying the same shared key. This scheme is proved to be resilient against $t/2$ compromised key. As t-degree function $f(x, y)$ with $a_{ij} = a_{ji}$, only $t/2+1$ different coefficients are unknown. By knowing $t/2 + 1$ compromised keys, the attacker can build up a system of $t/2+1$ linear equations with $t/2+1$ unknown variables. This set of linear equations is solvable which leads to expose the whole polynomial.

Generally, *Master key based pre-distribution scheme* and *Polynomial-based key pre-distribution scheme* provide an excellent scalability and little memory requirement. However, as one sensor node is compromised and the common secret information — master keys, polynomials or matrices — is extracted,

the whole communication inside WSNs are exposed. Thus, these schemes provide no resilience at all. Furthermore, like previous key exchanges, adding new sensor nodes or re-keying already deployed sensor nodes in this type of key pre-distribution schemes are not straightforward and require human interference.

### Online key-generation

In online key-generation, pairwise secret keys are not loaded into sensor motes before deployment. Contrarily, every pairwise key is created on-the-fly and on demand. Online key-generation normally involves cryptographic operation in order to mutually establish shared key in a secret way. Cryptographic key management scheme for WSNs are mostly based on available solutions for conventional network, especially public key cryptography solution. Key exchange between two sensor motes in public key cryptography involves using of one sensor mote's private key and other sensor mote's public key. In fact, every sensor mote only has to store it own key pair and it receives the other sensor mote's public key during exchanging key. Hence, this key exchange provides an infinite scalability and an excellent resilience against node capture. That is because if one sensor mote is captured, the attackers can extract no more information than mere information about the compromised node. However, using public key cryptography in constrained devices such as sensor motes is very processor, memory and time consuming. This is the reason why originally developers hesitated using public key cryptography solution in WSNs. However, recently there are several public key cryptography libraries aiming to run on constrained sensor motes such as RSA-based TinyPK [40], ECC-based EccM [33] and TinyECC [31], Pairing-based TinyPBC [35] and NanoPBC [12].

## 2.4 Authenticity, integrity and confidentiality solution

The first proposal published by Perrig et al is SNEP (*Secure Network encryption protocol*) [37]. SNEP promises to provide all confidentiality, authenticity, integrity and freshness during transmission. Following is the message structure that a sensor node A sends to a sensor node B :

$$A \rightarrow B : E_{\langle K_{encr},C\rangle}(D), MAC_{K_{mac}}(C|E_{\langle K_{encr},C\rangle}(D))$$

with $E_{\langle K,IV\rangle}(D)$ means encryption of data $D$ using a block cipher under any cipher-block chaining with encryption key $K$ and initialization vector $IV$. And $MAC_{K_{mac}}(D)$ means calculating MAC of data $D$ with $K_{mac}$ key. $K_{encr}$ and $K_{mac}$ are encryption and MAC key. $C$ is freshness counter. $M_1|M_2$ denotes concatenation of messages $M_1$ and $M_2$. This protocol provides semantic security, data authentication, replay protection, weak freshness and low communication overhead.

Another solution was proposed by Karlof, Sastry and Wagner, namely TinySec [25] has become default link layer security solution in TinyOS 1.x. This security solution offers two operation modes: *TinySec-AE* (authentication and encryption) and *TinySec-Auth* (authentication). TinySec-AE allows sensor mote sending messages as follows:

$$A \rightarrow B : IV, E_{\langle K,IV\rangle}(D), MAC_K(D)$$

and TinySec-Auth :

$$A \rightarrow B : D, MAC_K(D)$$

Comparing TinySec's TinySec-AE with SNEP, we can see that SNEP provides a better overhead as it does not send initialization vector IV together with the packet. Additionally, SNEP also provides freshness while TinySec does not. However, maintaining freshness counter at sensor motes causes memory increases.

# Chapter 3

# Design

This chapter discusses the design of our complete security solution. Due to our constrained wireless sensor nodes, our security solution mainly focuses on checking integrity, freshness and authenticity of sent packets. First, in order to understand the limitation of our platforms, the chapter starts with the information about the sensor nodes that we are going to apply our security solution on. Then, from the limitation, since it's unlikely to have an "all-in-one" solution, we need to restrict the requirements of our security solution. Since the importance of preventing packet injection or alteration is higher than preventing from eavesdropping, our security implementation mainly concentrates on checking integrity, freshness and authenticity of the packets. In order to fulfil these requirements, we implement a secure key exchange protocol which generates shared key between any two arbitrary sensor nodes. Later on, these shared keys are used to produce Message Authenticity Code (MAC) in order to ensure integrity and authenticity of every sent packet. Freshness of sent packet is ensured by a counter.

## 3.1 Platform

The hardware platform that we are going to deploy our solution are IRIS from Crossbow [4], shown in Figure 3.1. IRIS sensor mote is equipped with an 8-bit processor Atmel ATmega1281, 8KB RAM, 128KB programming memory and an additional 512KB serial flash memory. TinyOS serves as sensor mote's operating system. Sensor node uses radio transceiver which follows IEEE 802.15.4 standard [7] and supports sending data at maximum rate 250 kbps.

Figure 3.1: IRIS sensor mote from Crossbow

| 1 byte | 2 bytes | 1 byte | 0/2/4/10 bytes | 0/2/4/10 bytes | < 102 bytes | 2 bytes |
|---|---|---|---|---|---|---|
| Length | Flags | Sequence Number | Destination Address | Source Address | Data payload | CRC |

Figure 3.2: IEEE 802.15.4 packet format

-

IEEE 802.15.4 specification is designed for low rate wireless networks. IEEE 802.15.4 standard also has security specification which uses AES-CTR for encryption, AES-CBCMAC for calculating MAC and AES-CCM for both encryption and calculating MAC. Other Crossbow sensor motes such as Micaz motes, are equipped with CC2420 Radio Frequency transceiver where these security solutions are implemented. Our Iris sensor motes, unfortunately, uses RF2320 Radio Frequency transceiver which doesn't support IEEE 802.15.4 security solution. On the other hand, IEEE 802.15.4 security specification doesn't include specification about key management.

IEEE 802.15.4 uses two types of packets for data exchange: Data packet and Acknowledgement packet. As their names, data packets are mainly used to exchange data between two nodes while acknowledgement packets are intended to ensure that packets have truly arrived at their destinations. Between these two types of packets, data packets, due to its function, are the only type that is used for key management solution. An IEEE 802.15.4 data packet consists of several fields as seen in Figure 3.2.

The most important field that we need to pay attention is Data payload. This is the field that carries the actual payload from the upper layer. IEEE 802.14.5 only allows this field to be less than 102 bytes. This number defines the longest transmittable packet which in turn limits the possible length of key management messages. We see that, the longer the key management messages are, the less overhead our security solution makes toward sensor node. However, according to [26], increasing the packet length also leads to

increase packet loss rate which in turn, affects the efficiency of our security solution. The trade-off between overhead and packet loss needs considering while designing security solution.

## 3.2 Design Goals

As sensor platforms are constrained, our security solution needs to limit our security goals and performance is, of course, a must to consider. Besides, the variety of sensor platforms requires security solution to be flexible, which means, it is easy to integrate with current operating systems.

### 3.2.1 Security Goal

As mentioned previously, in order to maintain high performance in sensor nodes, a security solution is implemented to serve an essential limited set of security goals : integrity, authenticity and freshness. We decided to eliminate secrecy because of the special nature of wireless sensor networks. The purpose of wireless sensor networks in general is to monitor physical or environmental conditions which is inherently public information, everyone can easily get access to these information. For example, consider a temperature sensor network. An attacker can easily guess that transferred data are temperature figures so encryption is unnecessary. In the meanwhile, that attacker can cause a bigger trouble, for example false fire alarm, if he can intentionally insert a large temperature number into sensor network. Hence, eliminating encryption doesn't cause much difference in security properties but actually saves latency and power consumption compared to if encryption is implemented.

#### Integrity and Authenticity

Our security solution is designed to mainly focus on ensuring integrity and authenticity. A security solution promising to provide integrity means that every message is ensured to be received exactly the same as it is sent. If the adversary tries to modify a message such as inserting new information or altering information, the receiver should be able to recognize those changes in the message. While integrity is to prevent altering message, authenticity is to prevent packet injection. A security solution offering authenticity means that packets from illegitimate sources can be detected and discarded.

#### Freshness

To ensure freshness for every packet means to make sure every packet legitimately received is recent. More specifically, freshness is about how to prevent malicious nodes from using replay-attacks. We think that offering integrity and authenticity but no freshness still can give the adversary chances to cause consequences as severe as if integrity and authenticity were not provided. Consider our example — the temperature sensor network — again. In this case, every packet sent through this network is checked whether it has been modified or whether it is sent from where it claims to be sent. The adversary now is incapable of inserting new packets or modifying packets at his will. Yet he is capable of re-sending legitimate packets. This ability allows him to cause the same attack as mentioned above, false fire alarm. He can perform this attack whenever a high temperature message has been legitimately sent through this sensor network. Hence, freshness is another important goal in our security solution.

### 3.2.2 Performance

As security solution is going to work on constrained sensor nodes, performance is a must to pay attention to. Performance involves power consumption, RAM consumption or even time delay. Time delay here could be a delay at each sensor node where security solution is applied. A more important delay is latency in total when packets are transferred from sender to receiver. There are many factors that greatly affect sensor node's performance. One of those is *the cryptographic algorithm*. While cryptographic computation happens at constrained sensor node, the attacker, unfortunately, can use computers with high processing power to break cryptographic algorithm. Yet, to avoid such problem, increasing security level by using computational complex cryptographic algorithm consumes more power, processor or even RAM. Hence, trade-off between security level and performance largely lies on choosing an appropriate cryptographic algorithm. Another factor affecting performance is *packet design*. Since in order to provide security, additional information, such as Message Authenticity Code (MAC), is added into normal packet, the actual message length is shortened which, in turn, decreases the throughput of network. Because time-sensitiveness can be a requirement in some sensor networks, such as intrusion detection system or fire detection system, latency in message transmission should be carefully taken into account.

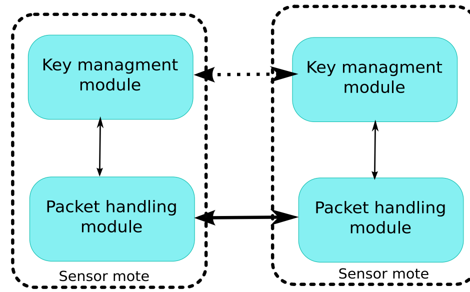


Figure 3.3: Security architecture

### 3.2.3 Ease of use

As our security application acts as a service to other applications on top, in order to alleviate programming work, our aim is to design a module which provides APIs similar with the original interface as much as possible. This helps our module to easily integrate with current TinyOS model. Additionally, if in the future, other modules are added into TinyOS such as routing module, our module still can be easily placed into TinyOS module stacks.

## 3.3 Protocol Design

### 3.3.1 Our security model

As presented as in Figure 3.3, our security model for wireless sensor networks consists of two modules: Key management module and packet handling module. These two modules target to solve each tier in two-tier problem in wireless sensor network as in Section 1.2. Packet handling module is the lower layer module which is responsible for providing integrity, authenticity and freshness. In order to do so, packet handling module requires a shared secret key between every two nodes. This shared secret key is offered by upper module, key management module. In order to generate shared secret, two key management modules can mutually agree on secret keys with or without communicating with each other depending on the types of key management solution. In our security solution, we decided to use cryptographic key exchange algorithm as key management solution, while using MAC and freshness counter in order to provide integrity, authenticity and freshness.

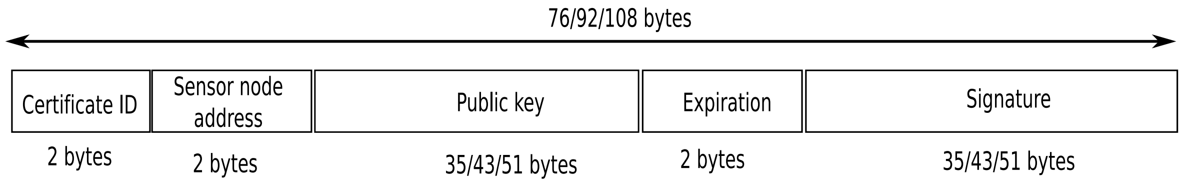


Figure 3.4: Certificate format

### 3.3.2 Key management module

As we aim to implement security solutions for various purpose WSNs, public key cryptographic key exchange is the most versatile solutions. We decided to use Elliptic Curve cryptosystem as the key exchange solution. Elliptic curve cryptography is observed as suitable solution for constraint devices. The advantage of elliptic curve cryptosystems (ECC) lies in keylength. It is widely believed that ECC uses shorter keys but provides the same security level as other public-key cryptosystem. According to RSA Lab [1] in 2000, breaking a 160-bit key length ECC cryptosystem requires 600 months with 4300 machines consuming 4GB memory. The same resources are needed for breaking a 760 bit key length RSA cryptosystem [1]. And some organizations such as ECRYPTII [8] consider 160-bit key length ECC equivalent to 1024-bit key length from other asymmetric cryptosystem based on discrete algorithm problem. Shorter secret not only helps sensor motes saving memory and allows storing more secret keys but managing smaller keys promisingly saves processor and RAM. Additionally, since public key needs to be exchanged between sensor motes in order to generate shared secret key, the shorter public key is, the shorter exchange message which in turn reduces the number of exchanged packets.

#### Public key Infrastructure(PKI)

In practical implementation, the issue of using public key cryptography is that public keys have no connection with their owner, sensor nodes. Thus, we designed a PKI in order to bind public keys with their corresponding sensor nodes and also maintain this binding. In our PKI, Certificate is used as a mean to testify the sensor node's ownership towards public key. Certificates contains certificate ID, sensor node address, public key, expiration date and signature as in Figure 3.4.

*Certificate ID* is unique among all certificates issued by CA. Certificate ID

is required in certificate so that sensor nodes don't have to store the whole lengthy certificate; instead they only have to store Certificate ID and essential information such as expiration date. Because only one secret key is generated, other sensor motes' public keys and signatures - the longest parts in certificate - are trivial.

*Sensor address* is the unique address in the whole network which is assigned to each sensor node in the beginning. In TinyOS operating system, it's 2-byte value TOS_NODE_ID.

*Public key* is the sensor node's public key, an Elliptic curve point. The size of public keys depends on ECC key size. The corresponding public key sizes of systems using a 128-bit, 160-bit and 192-bit ECC key are 35 bytes, 43 bytes and 51 bytes.

*Expiration date* is the date from that onward the certificate is invalid.

*Signature* is issued by Certificate Authority (CA) who signed certificate ID, sensor address, public key and expiration date all together with public key using ECDSA signature algorithm. In order to differentiate trusted CA from malicious one, sensor nodes are initially loaded with a set of trusted public keys. These public keys belong to trusted CA and are used to verify certificates that sensor nodes receive during their operating lives. Similar to *public key*, signature lengths also vary with ECC key sizes. Systems using 128-bit, 160-bit and 192-bit ECC key respectively results in 35-byte, 43-bytes and 51-byte signature.

However, beside CA's public key, each sensor node has to be loaded with its own unique private key and certificates. And every time, when a sensor node makes an key exchange with other sensor node, it has to store certificate information which are Certificate ID and expiration date. It is essential to keep information about certificate and its expiration date since certificate can become invalid later due to compromised certificate or expired key.

The big problem of our security solution is secret key storage. Static information such as CA's public key, sensor node's certificate and private key can be easily stored into flash memory from where sensor nodes can read when necessary. However, per-link secret key is established on demand and thus the required memory increases together with the number of communicating nodes. In order to solve this problem, a memory swapping [41] between RAM and flash memory is recommended, however this mechanism is not covered in this thesis.

Node capture attacks is a common security threat to wireless sensor networks which may happen at any time. Attackers can purposely extract information

from sensor node's memory which leads to private key leakage. Compromised keys then results in "false node" attack. In order to prevent attackers from using compromised private key to set up "false node", *revocation list* — a list of all compromised certificates —- is broadcast to every node in network to inform about "broken certificate". Revocation list should be signed by CA to make it unforgeable.

### Design alternatives

As cryptographic solutions involves CPU-intensive calculation, we examined two simplest key exchange solutions ECDH and EC-based Station-to-Station (STS) protocol. 160-bit key length was used to examine both schemes, hence certificates are 92 bytes. These two schemes can be described briefly as follows:

Let an Elliptic curve domain be given, $(E, a, b, G, n, h)$. Two sensor motes A and B want to establish shared key secret key, A is loaded with private key $d_A$ ( a number between $[1, n-1]$ and certificate $cert_A$ which contains A's public key $Q_A$ signed by CA. Similarly, B has private key $d_B$, certificate $cert_B$ with public key $Q_B$.

- ECDH
  - $A$ sends to $B$ : $cert_A$ ( 92 byte message).
  - $B$ verifies $cert_A$. if correct, B replies with $cert_B$ (92 byte message) and then computes shared key $k = d_B Q_A$.
  - On receiving $cert_B$, $A$ verifies it. If correct, A computes shared key $k = d_A Q_B$ which equals to $d_B Q_A$.
- EC-based STS
  - $A$ generates a random number $a \in [1, n-1]$ and sends $aG$ to B (43 byte message).
  - On receiving, $B$ also generates randomly $b \in [1, n-1]$, computes $bG$ and replies with a message $(bG, cert_B, Enc_K(Sig_{d_B}(aG, bG)))$ (177 byte message ) with $K = b(aG)$, $Enc_K()$ any symmetric encryption scheme using key $K$, $Sig_{d_B}()$ ECDSA signature scheme using key $d_B$.
  - On receiving, $A$ verifies $cert_B$. If correct, $A$ continues to compute $K = a(bG)$ and use $K$ to decrypt $Enc_K(Sig_{d_B}(aG, bG))$.

| | Key exchange scheme | |
|---|---|---|
| | ECDH | EC-based STS |
| Hash | 1 | 2 |
| Modulo calculation | 3 | 6 |
| ECC multiplication | 3 | 4 |
| ECC addition | 1 | 1 |
| Symmetric encryption | 0 | 1 |
| estimated total runtime on MicaZ sensor mote | 9.8s | 20.1s |

Table 3.1: Computational overhead between ECDH and EC-based STS

Then, $A$ verifies $Sig_{d_B}(aG, bG)$. If correct, $A$ replies with message $cert_A, Enc_K(Sig_{d_A}(aG, bG))$ (134 byte message).

We compare these two schemes based on the following criteria :

- Computational overhead
- Communication overhead
- Security strength

**Computational overhead** Table 3.1 shows the computational overhead between these two schemes. EC-based STS certainly costs more computations than ECDH. In order to quantify the computational difference, we uses the performance of TinyECC [31] library running on MicaZ platform to see the real runtime between two schemes. As we can see, EC-based STS scheme does take double runtime compared with ECDH.

**Communication overhead** With ECC 160-byte key length, the ECDH scheme requires each sensor mote to send a 92 byte message, while in EC-based STS scheme, the key exchange initiator has to send 2 messages ( 43 bytes and 134 bytes) and the responder only sends 177 byte message. So EC-based STS scheme requires a higher communication overhead. Importantly, the impact of heavy communication in EC-based STS scheme probably increases together with the distance between two sensor motes. Since the further two sensor motes are, the more intermediate sensor motes are used to forward messages.

| | ECDH | EC-based STS |
|---|---|---|
| number of key can be established | one | many |
| Key freshness | No | Yes |
| Key confirmation | No | Yes |
| Forward secrecy | Not defined | Yes |
| Entity authentication | No | Yes |
| Non-repudiation | No | Yes |

Table 3.2: Security strength comparison between ECDH and EC-based STS

**Security strength** Even though EC-based STS is more processor and energy consuming, it does offer a better security solution. This can be seen as in Table 3.2. As in wireless sensor network scenario, *entity authentication* and *non-repudiation* are not necessary since sensor motes can be easily captured and forged. A forged sensor node can easily perform illegitimate key exchanges with other sensor motes while such key exchanges are still considered *entity authenticated* and *non-repudiated*. This hardly happens in conventional network as taking control over a base station or a network entity costs more effort than in WSNs.

Another drawback of ECDH is one key over sensor node's lifetime. This causes danger for the network where key disclosure can happen easily. Because if a shared key between two sensor nodes is compromised, even though two sensor nodes run the key exchange scheme again, the same already-exposed key is generated. However, unlike conventional network, wireless sensor networks, due to their limited battery, can only operate for several years. As the experiment conducted by RSA lab [1], the requirement for breaking ECC 160 bit key length is 600 months - longer than lifetime of a sensor node - proving that brute-force attack is not a security concern in our security solution. In other cases, such as node capture, the whole sensor node is exposed, using EC-based STS also could not differentiate between compromised and uncompromised sensor motes.

**Key exchange protocol** In conclusion, after considering between two design alternatives, we decided to use ECDH to achieve a better performance with an acceptable security strength. The result from ECDH is later hashed using SHA1 in order to produce final shared secret key. Because of simple ECDH key exchange protocol, our key management happens as in Figure 3.5. Both *key exchange request* and *key exchange response* only contain sensor node's certificate which is used to legitimately bind public key with sensor

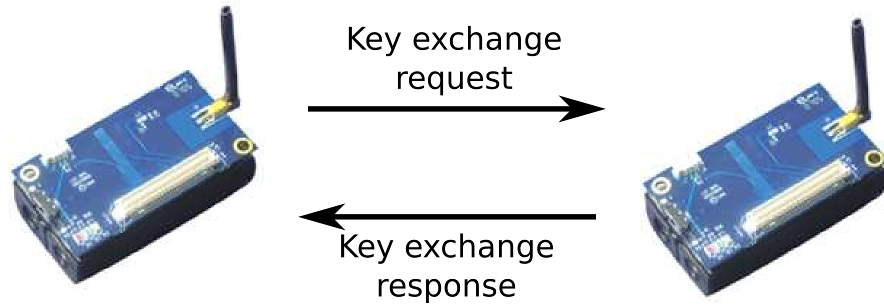


Figure 3.5: Key exchange protocol

node address.

### 3.3.3 Packet handling module

Packet handling module acts as a middleware layer that calculates the MAC for every packet received from the upper layer. This MAC is inserted into the packet later on, and the whole new packet is transferred to the lower layer. In conventional network, this type of security module is usually placed at the IP network layer with the intention to secure packet transmission between two arbitrary network entities. This end-to-end security solution however seems not receiving much support in wireless sensor networks. Many security solutions such as TinySec [25] in wireless sensor networks are placed at the link layer of network protocol stack which offers hop-by-hop secure packet transmission. We think that these two types of secure transmission, hop-by-hop and end-to-end, are all important in wireless sensor network.

*Hop-by-hop security mechanism* such as TinySec is a security solution which is suitable for environment monitoring sensor networks. In this type of network, sensor readings are collected from many sensor nodes and forwarded back to a base station. Nevertheless, due to gradually changing or broadly happened environment events, neighbouring sensor nodes normally report the same or similar data readings. In order to save energy and traffic, data aggregation or selection can happen at some intermediate nodes in network that are better at processing power and energy, before data finally reaches base station. Since this type of data transmission requires intermediate nodes to access and modify packet content, combining security processing with data processing is a good solution. Additionally, hop-by-hop security mechanism can protect network from packet injection at a lower cost of power

consumption. Unlike end-to-end security mechanism where packet injection can only be found at the destination, hop-by-hop solution is able to discard false packets at the edge of the network. This early packet discard certainly saves intermediate sensor nodes from saving precious energy and processor in forwarding packets to the destination. The other advantage of hop-by-hop security solution lies in saving precious memory by storing less shared secret keys. In hop-by-hop security mechanism, only neighbouring sensor nodes securely communicate with each other, thus a sensor node only have to store shared keys with its surrounding nodes. This reduces greatly key storage requirement which is a problem in end-to-end network.

*End-to-end security mechanism*, on the other hand, is suitable for wireless sensor network where end-to-end reliable transmission is a requirement. Such sensor networks are battlefield surveillance network, intrusion detection applications, etc ... In this type of network, unlike aforementioned type of wireless sensor network - environment monitoring sensor network, every sensor reading is critical and required to reliably transferred back to base station. A shortage of hop-by-hop security mechanism in providing end-to-end reliable transmission is failure to prevent tamper attack. While node captures in sensor network can happen easily and compromised nodes are difficult to discover, in hop-by-hop security mechanism attackers can easily control compromise and control sensor nodes in order to intercept data forwarded through these nodes. Unlike end-to-end security mechanism where attackers can only drop packets forwarded through compromised nodes, hop-by-hop security mechanism gives attackers chances to modify packet contents which leads to the failure of packet-critical network. Thus, end-to-end security mechanism gives a better resilience in network where undetected compromised nodes exist. The other advantage of end-to-end security which makes it surpass hop-by-hop sensor network in mission-critical network is shorter total end-to-end latency together with less energy and processor consumption. While hop-by-hop saves power and process consumption during packet injection attack, end-to-end security solution certainly requires less power and processor due to absence of in-network data processing which results in less total end-to-end transmission delay.

As the different role of security solution in various types of wireless sensor network, it's difficult to have an "all-in-one" security mechanism. Thank to the flexibility of TinyOS operating system, we decided to design our packet handling module so that our module can be easily integrated into TinyOS protocol stack at any layer. This helps users adjust our security solution to suit their needs. As mentioned earlier, our security solution provides freshness together with integrity and authenticity.

### Freshness

There are two types of freshness : *weak freshness* and *strong freshness*. The former ensures freshness by checking the arrival order of received packets. This is achieved by maintaining counters between two communicating parties. Packets are considered valid if packets received later contains higher value counter than packets received earlier. This mechanism produces a low communication overhead but instead cost sensor nodes memory. As the number of communicating nodes increases, the required RAM increases. The other freshness solution, *strong freshness*, uses request-response method to provide total packet order. For example : the receiver sends a randomly generated *nonce* together with a *request* to the sender, then the sender response with the same *nonce* or a modified version of *nonce* to ensure freshness. This mechanism causes high communication overhead ( extra packet, extra computation) but in return, valuable memory is saved.

Between these two types of freshness, we decided to use *weak freshness* as our freshness mechanism. The first reason is *weak freshness* can easily provide total transparency to upper layer. Secondly, *weak freshness* is enough for wireless sensor networks which only expect transmitting sensor readings back to base station. Thirdly, as we are going to implement per-link shared secret key, solving the problem of placing key storages can cover the problem of storing freshness counter.

In *weak freshness* solution, the longer counter is, the higher security the solution offers. However, longer counter means more RAM and higher communication overhead. So the question is how to choose a suitable long counter in order to balance security and memory trade-off ? Our IRIS mote is able to send up to 250 kbps [4] and the default packet length in TinyOS is 54 bytes which means if a sensor node sending at maximum speed use 2 byte counter in nearly 2 minutes. However, sensor node rarely uses maximum data rate due to energy saving. For example, in the experimental wireless sensor network deployed at Great Duck Island [39] , sensor motes only read once every 70 seconds. Additionally, since each counter is kept specific between every two sensor node, the frequency of increasing one specific counter is even lower. So in order to save memory and provide reasonable security, 2-byte counter is default option in our design. With this 2-byte counter, if one packet is sent every minute between the same two sensor nodes, counter is used up during 45 days.

### Integrity and authenticity

Integrity and authenticity is ensured by calculating MAC for both data and aforementioned freshness counter. We used CBCMAC with SkipJack as block cipher to generate message authentication code. According to [20], CBC-MAC with AES does outperform HMAC with SHA1. A survey for block cipher in wireless sensor network has been conducted in *TinySec*, as AES, and TripleDES, two commonly known block ciphers, are implemented and tested their speed, the author has suggested SkipJack as block cipher for sensor motes. Hence, we decided to use CBCMAC with SkipJack to provide a better runtime.

The output from Skipjack is 64 bits which means the longest possible MAC is 8 bytes. With 8 byte MAC, the adversary needs to generate blindly about $2^{63}$ packets in order to forge a valid packet. To test whether the packet is valid or not, the adversary needs to probingly send to a sensor node. With 250 kbps, $2^{8}$ packets are sent within one minute, thus $2^{63}$ packets requires $2^{55}$ minutes. The attack probably couldn't complete due to sensor node's power depletion.

# Chapter 4

# Implementation

This chapter presents how security design described in the previous chapter is implemented. Since our security solution comprises of two modules: Key exchange module and Packet handling module, this chapter is organized into two sections. Section 4.1 describes our key exchange module implementation. Section 4.2 describes our integrity and freshness checking module with an overview of transparent implementation of this module toward other modules in TinyOS. All of the implementation was developed in nesC language. Additionally, thesis worker also put effort in porting TinyECC and TinySec libraries from TinyOS 1.x into TinyOS 2.x and modified these two libraries in order to work on IRIS motes.

## 4.1 Key exchange module

Key exchange module is implemented as a single-threaded module in TinyOS which provides key exchange functionality to other modules. Besides, according to the design in previous chapter, we are going to use *Certificate* as a proof of sensor node address, thus key exchange module also contains other components which are responsible for managing sensor node's *Certificate* and trusted public keys. As key exchange module serves as a underlying daemon which might receive many requests from other modules above or even handle requests in form of *Request key exchange packet* sent from other sensor nodes, key exchange module requires having *Request queue* to avoid loosing any incoming request. As long *key exchange message* containing *Certificate* and *Signature* results in long packet size which increases packet loss rate, we decided to implement two versions of key exchange module: *the full mes-*

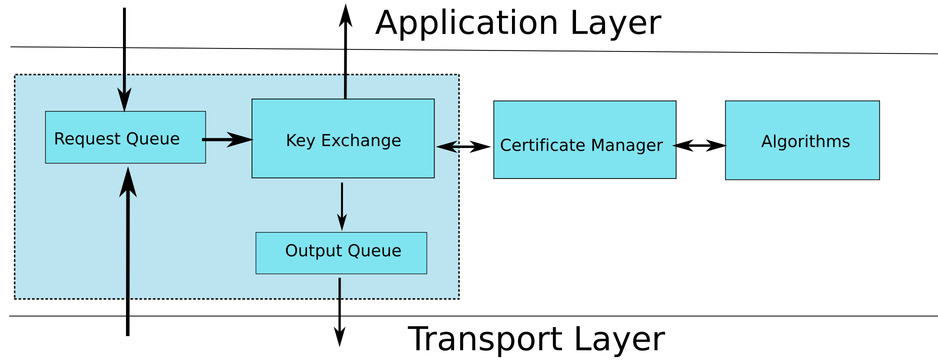


Figure 4.1: Key Exchange module decomposition

*sage key exchange* where *request* and *response messages* are sent in only one packet and *the fragmented message key exchange* where *request* and *response messages* are split into a number of smaller packets according to the possible longest packet that a sensor node is able to send.

### 4.1.1 Architecture description

#### Full message key exchange

Architecture overview from key exchange module is presented as in Figure 4.1. The whole module is divided into several smaller components: *Key Exchange*, *Certificate Manager* and *Algorithms*.

The core component is *Key Exchange* whose functionality is receiving *requests* either from upper layer or from other sensor nodes. *Request* from upper layer consists of *sensor node address* and *certificate type* indicating, while *request* from other sensor nodes come in form of *key exchange request* message (as in Section 3.3.2). Since *Key Exchange* module involves time-consuming activities such as signature verification or key generation, newly arrived request can easily get lost while *Key Exchange* module is still in busy processing old request. In order to avoid such loss, a *Request Queue* is placed at input of Key Exchange module to store arriving requests before they are getting processed. *Key exchange request* from application layer is queued so that our module is able to serve many applications reducing probability that application layer modules access key exchange service while our module is in use. *Request Queue* also can buffer *key exchange request* from other sensor node which is delivered through radio transceiver. As Radio transceiver is a

frequently used component in sensor node, the longer that one *Key exchange* packet occupies radio buffer due to our busy module, the higher possibility that it could prevent other applications from receiving their specific packets. Hence, *Request Queue* also improves the whole system response. Additionally, as the lower layer such as Link Layer probably serves only one task at a time, *Key Exchange* module needs an *Output Queue* to store packets waiting until the lower layer is ready to handle new packet. Optimizing queue size is a reliability-memory trade-off, as increasing queue size helps security module buffer more packets which avoids loosing request but in return, increases memory usage. Our IRIS sensor mote uses RF230 radio transceiver for which TinyOS operating system implements a buffer of 3 received packets capacity [6]. As our *Key exchange* module relies on top of this radio layer, we applied the same capacity for *Request Queue* and *Output Queue*. Since our *Key exchange* module requires bidirectional communication, even if we offers a higher capacity queue, *Key exchange response* or *Key exchange request* sent from other sensor nodes still get lost if radio layer is not able to handle more than 3 packets.

*Certificate Manager* module manages *Certificate*, *private keys*, *trusted public keys*. *Certificate Manager* also provides shared key calculation functionality to Key Exchange module.

*Algorithms* module provides essential algorithms for Certificate Manager such as signature verification or shared key calculation. Algorithm module is based on TinyECC library [31] . As TinyECC library is only working on TinyOS 1.x, we ported it into TinyOS 2.x. However, TinyECC is originally designed as *blocking model* which means whenever an operation from one module takes control of the processor, other modules have to wait until that operation is totally completed. Unfortuanately, as TinyECC library contains mainly long-running operations such as signature verification or certificate signing, this blocking model prevents other operation from running leading to unresponsive system in such constrained processor as sensor mote. In order to fix this problem, we modified TinyECC from *blocking model* into *split-phase model*. In *split-phase* system , on the other hand, when a time consuming operation is called, the call returns immediately which allows the caller can be scheduled for other operations. As soon as that time consuming operation completes, it signals a callback function provided by the caller. However, even when the library is changed into new running model, many heavy functions in TinyECC still cause unresponsiveness in IRIS mote. Hence, in this new model, by restructuring long running operation into short TinyOS *tasks*, we also modified time and processor consuming operations allowing TinyECC library to be able to run in constrained IRIS mote.

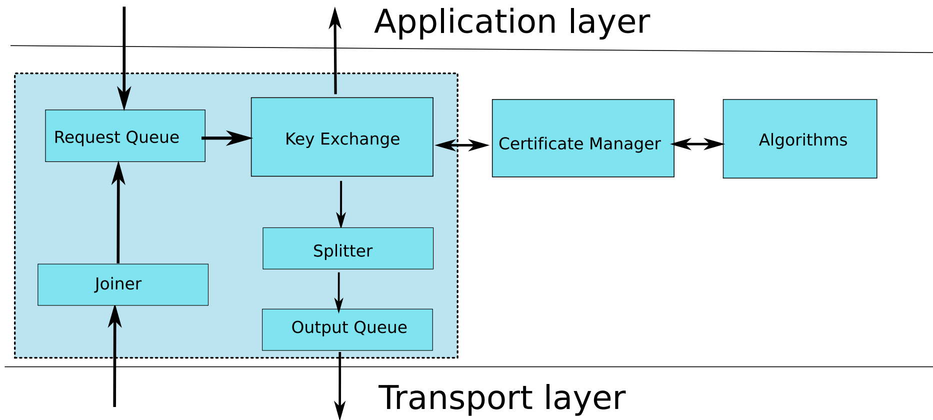


Figure 4.2: Key Exchange module decomposition - Fragmented exchange message version

### The fragmented message key exchange

In Internet network, Internet layer provides functionality of IP fragmentation which helps passing large packets through links with smaller maximum transmission unit (MTU). Since wireless sensor network is expected to convey small packet such as sensor readings, such functionality has not been supported in TinyOS operating system. The maximum transmission unit in IEEE 802.14.5 can carry 102 bytes pay load, yet TinyOS allows developers to re-define this number into a smaller one in order to save memory. In fact, the default payload in TinyOS for IEEE 802.14.5 link layer is 29 bytes which means every time sensor node buffers a packet in preparation for sending, only 29 bytes of memory are required instead of 102 bytes. Nevertheless, our *key exchange request* and *key exchange response* are certainly long packets. If an ECC 160 bit key is used, a *key exchange request* or a *key exchange response* cost a 92 byte packet. Especially, in case of an ECC 192 bit key, a 108 byte packet definitely needs packet fragmentation. Additionally, since currently TinyOS only provides a simple link layer transmission, every packet is sent without reliability and susceptible to packet loss. As wireless transmission media is unstable and easily causes corruption, large packet size results in higher packet loss rate [26]. Thus, minimizing packet size improves reliability and stability of the whole sensor network. As security module probably is the only one that requires sending such long packets, we decided to split our *key exchange messages* into smaller transmission unit. The architecture overview for this solution is presented as in Figure 4.2

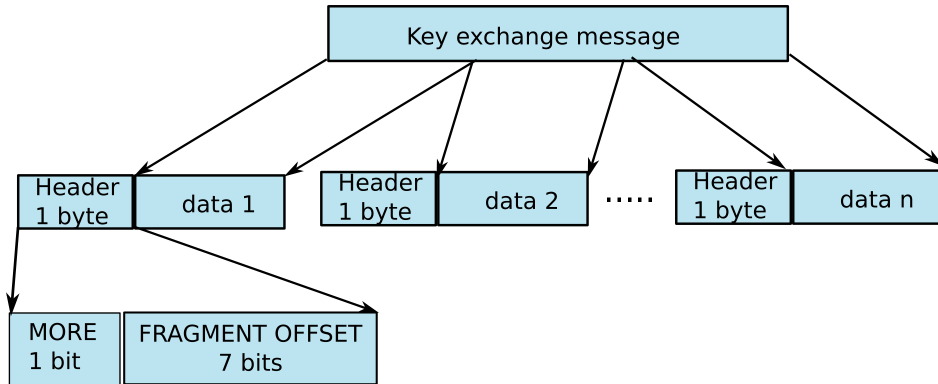


Figure 4.3: Packet fragmentation

Two new components, *Splitter* and *Joiner*, are integrated into Key Exchange module. As their self-explanatory names, *Splitter* is responsible for fragmenting packets before sending into the *Output Queue* and *Joiner* has to buffer arriving packets and push a complete *key exchange message* to *Request Queue* whenever all fragmented packets of a single message have arrived. The fragmented packet is constructed as in Figure 4.3. The number of fragmentation depends on the MTU of underlying layer. In order to reduce communication overhead, *header* only costs one byte. Inside *header* byte, MORE flag is turned on in all except the last fragment and FRAGMENT OFFSET indicates the position of the fragment in *key exchange message*. As our possible maximum packet size is 108 bytes (when 192 bit key is in use), 7 bit FRAGMENT OFFSET with maximum value 128 can cover all possibilities.

## 4.2 Packet handling module

Packet handling module provides integrity checking for every application above it. The most important requirement in implementing packet handling module is low running time and transparency toward upper and lower layer. As this module is frequently used by many applications lying on top, we expect short running time of this module in order to cause not much latency in end-to-end transmission. Transparency in implementation not only relieves developers work but also increases flexibility in module integration. As packet handling module requires handling shared secret key (MAC key) between sensor nodes, a separate component for managing MAC key is im-

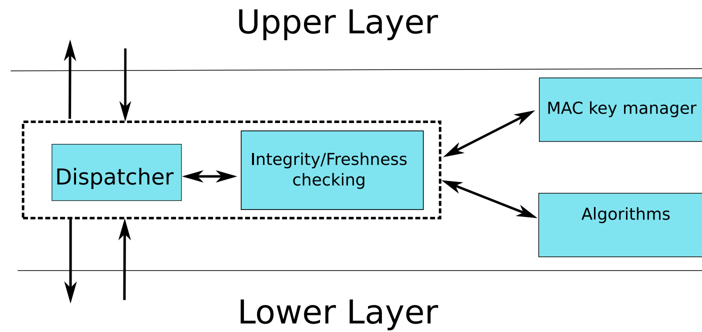


Figure 4.4: Packet Handling architecture overview

plemented.

### 4.2.1 Architecture description

Figure 4.4 show the architecture overview of Packet handling module.

As mentioned earlier, we design our security module with a purpose that it can be used as either hop-by-hop or end-to-end security solution. In hop-by-hop security architecture, our module can be placed right on top of link layer to modify each packet before being sent to neighbouring node. In this scenario, upper layer could be network layer and lower layer is link layer. In end-to-end security architecture, lying just on top of transport layer security module can transparently serve application layer. The packet handling consists of three components: *Integrity/Fresness checking*, *MACKey storage* and *Dispatcher*.

*Integrity/Fresness checking* module is the core component modifying packet received from upper layer and transferring to lower payer. As a packet from upper layer arrives this module, the whole packet and *2 byte freshness counter* are input to MAC calculation function. The result and counter is placed at the beginning of the whole new packet. And this packet as in Figure 4.5 is passed down to lower layer. Placing *MAC* and *counter* value at the beginning of new packet allows a fixed *MAC* and *counter* value with a variable-sized *data*.

Because there are many applications at upper layer such as reading temperature, pressure,.. using security solution service as well as many types of service at lower layer such as broadcast sending, end-to-end sending or different routing protocol that applications want to use, *Distpatcher* examines

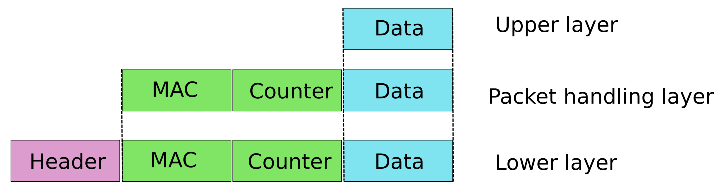


Figure 4.5: Encapsulation of packet through Packet handling layer

message header in order to direct packet into correct service at lower layer or return packet into correct application at upper layer.

*MAC key manager* stores and manages all MAC keys shared with other sensor nodes in the network. A MAC key manager also contains other MAC key-related information such as *address* of sensor node that this key is shared with, information about Certificate including *Certificate ID* and *expiration date* in order to drop this MAC key when it is expired and *freshness counter*. Such information together with key itself costs maximum 18 bytes of memory for each established MAC key. As the cost of memory increases when the number of entities that a sensor has to communicate with grows, we expect that a memory swap mechanism between RAM and flash memory needs to be implemented to solve memory problem. However, in hop-to-hop security model, the number of entities that a sensor node can talk with should not be too large since sensor node applies security mechanism only with neighbouring node. In end-to-end security model, since there is no restriction in type of entities that a sensor node could communicate with, capacity of reserved memory for MAC key could partly decide the maximum number of possible communicating entities which in turn leads to the decision of wireless sensor network size.

*Algorithms* provides encryption and CBC MAC algorithms required by Packet handling module. This component is based on TinySec library. Like TinyECC library, TinySec library is also designed for TinyOS 1.x and follows *blocking model*. Thus, we also ported TinyECC into TinyOS 2.x, converted TinyECC into *split-phase model* and split long running functions ( ECDSA verification and ECDH shared key computation function).

### 4.2.2 Transparency solution

As we mentioned earlier, transparency is our goal in implementing *Packet handling module* since it relieves developers' workload in addition to gain flexible and successful integration into TinyOS operating system. In order to

achieve transparency, several other additional modules were implemented in order to provide a consistent architecture from lower layer up to upper layer. In TinyOS, every layer communicates with upper layer through interface *AMSend* and with lower layer through interface *Receive*. Additionally, each layer needs to provide following services *AMPacket*, *Packet* and *Acks* [30, Sec. 9.2.3]. In order to provide transparency and flexible integration, we implemented our module according to two aforementioned interfaces and provides above services also.

# Chapter 5

# Evaluation

Due to constrained resources, each application in sensor mote needs analysing in many aspects such as memory usage, running time or energy consumption. In this chapter, we analyse and discuss the performance of our implementation by presenting several experiments that we have conducted on IRIS sensor mote. The chapters starts with the section about *Key exchange module* and then the second discusses about *Packet handling module*.

## 5.1 Experiment platform

The test environment consists of two IRIS sensor motes (as described in 3.1) running our applications. Since communications involve only two sensor motes, we deployed *Key exchange module* and *Packet handling module* right on top of IEEE 802.14.5 link layer. Test platform can be visualized as in Figure 5.1 . These two sensor nodes use TinyOS 2.1.0 operating system. For *Key exchange module*, we used *secp160r1* elliptic curve recommended by SECG [10] which uses 160 bit key parameters.

## 5.2 Key Exchange module

### 5.2.1 TinyECC optimization

Memory usage in *Key exchange module* involves programming memory ROM and , more important, working memory RAM. The TinyECC library provides several optimizations to save either running time or working memory RAM.

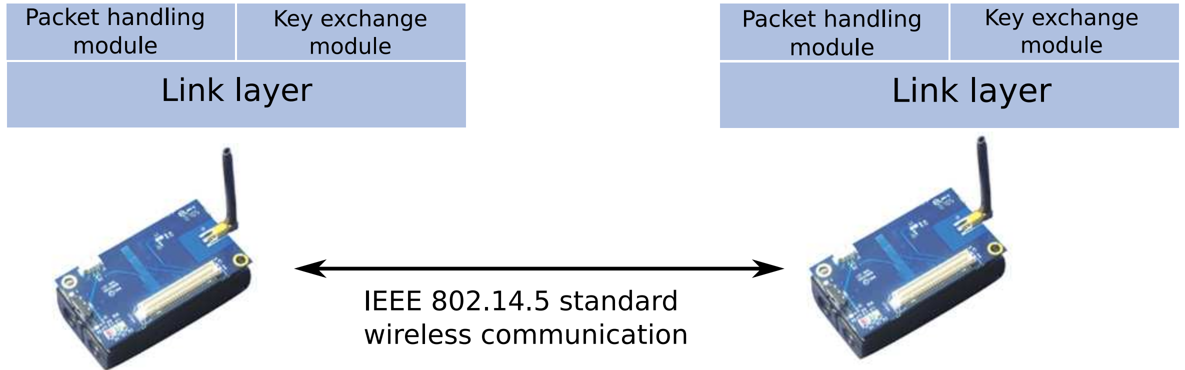


Figure 5.1: Experiment platform

Those optimizations are :

* *Barrett reduction* [13]: a method to quickly compute modular reduction $y = x(mod n)$ using *fast division algorithm*.

* *Curve optimization* is applied for Elliptic curves recommended by NIST [9] and SECG [10] which uses pseudo-Mersenne prime. This optimization helps improving module reduction on pseudo-Mersenne prime.

* *Projective coordinate system* [23, Sec. 3.2.1] . The library will use projective coordinate to present and perform Elliptic curve operations rather than using affine coordinate

* *Sliding window for scalar multiplication* [23, Sec. 3.3.3]. This optimization is meant to boost multiplication of an integer and an ECC point.

* *Shamir trick* optimization [23, Sec. 3.3.3] is used to improve signature verification.

All of these optimizations aim to reduce processing time by compensating working memory and programming memory. However, not all of these optimizations are essential to our application. In order to select suitable appropriate optimization for our application, we measured ROM and RAM when each of these optimization is turned on. Additionally, execution time for exchanging key was also measured. This execution time was recorded since a sensor mote starts from verifying certificate until calculating shared key is completed. We only conducted test using *Full message key exchange*

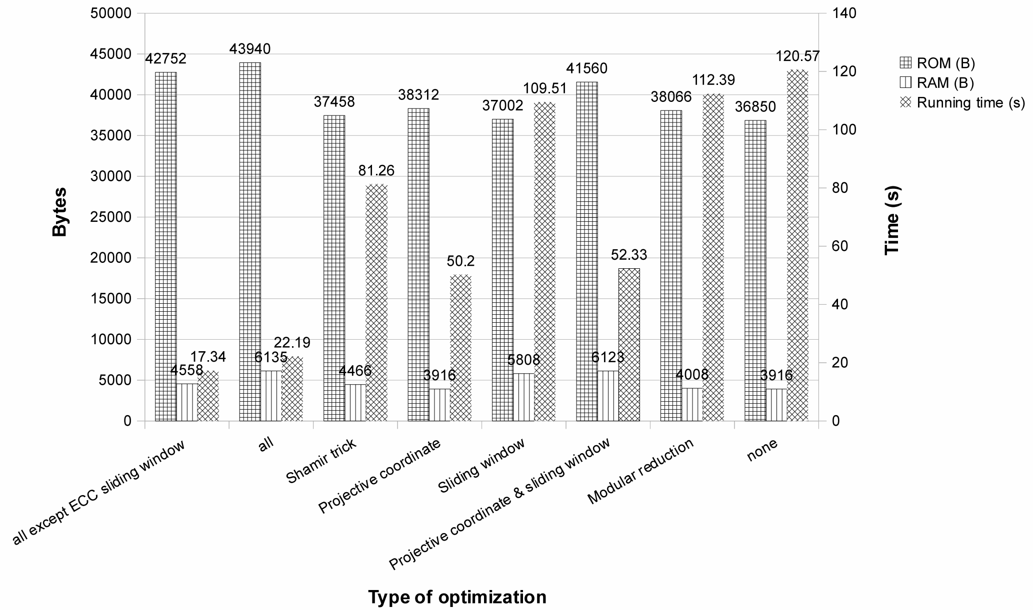


Figure 5.2: Comparing TinyECC optimization

version. As using *fragmented message key exchange* doesn't affect the comparison among TinyECC optimization, we believed that conducting with *Full message key exchange* gives a similar result.

**Experiment result** The result is presented in Figure 5.2. In this experiment, as *Barrett reduction* and *Curve optimization* together improves module reduction, both these optimizations are turned on at the same time in one experiment (Modular reduction). One notice about our experiment is that that ROM and RAM are listed including memory used by TinyOS operating system and other necessary modules also. Since TinyOS operating system and other modules are kept untouched during switching between optimizations, memory usage still can be correctly compared in this experiment.

As seen from the result, the most effective optimization is *project coordinate* which largely reduces the running time but only causes increment in programming memory ROM and no change in working memory RAM. *Modular reduction* and *Shamir trick* improves a bit in processing time but also leads to increase in both ROM and RAM. *Sliding window* is a special optimization since it only reduce execution time a bit but occupies so much RAM (from 3916 bytes to 5808 bytes). Additionally, if *Sliding window* and *projective coordinate* are combined, runtime is even larger than *projective window* alone.

The reason is that *sliding window* costs a lot of time in the initialization phase due to heavy pre-computations and those precomputed values then are expensively stored into RAM. However, these pre-computations are related to sensor node's public keys which are different in every key exchange. (Because each key exchange session is carried out with different communicating sensor node). Hence, those stored value are required to be recomputed for every single key exchange session causing wasting memory usage and even processing time. So, we decided to run our application with all optimization except *sliding window.* The execution from this combination of optimization gave the best performance, lowest execution time but quite acceptable memory usage. From all above experiments, we select two optimization settings :

* *Memory optimization* This optimization is achieved by only switching *projective coordinate* on.
* *Execution time optimization* This optimization is accomplished by using all optimizations except *sliding window.*

Actually, *memory optimization* results in a bit larger ROM usage but still maintains the same RAM usage. Since our sensor mote IRIS contains 128KB ROM but only 8KB RAM, compensating a bit ROM usage (from 36KB to 37 KB) to maintain a reasonable runtime ( from 120.57s to 50.2 s) is worthy. Of these two optimizations, *memory optimization* certainly offers minimal RAM usage (3916 bytes) but causes longer execution time ( 50.2s). On the other hand, *execution optimization* compensates RAM usage (4558 bytes) for shorter execution time (17.34s).

### 5.2.2 Memory usage

This section describes the impact of *Key exchange module* on memory usage ( both working memory and programming memory) in IRIS mote.

In TinyOS operating system, an nesC application has similar *memory model* like in Unix operating system. The whole working memory RAM can be divided into : *stack segment* and *data segment. Stack segment* is placed in higher memory address which stores data related to calling a function or handling an interrupt. *Data segment*, which starts from the end of memory space and grows upward, consists of *Heap*, *Data* and *BSS. Heap* is used for dynamic allocation memory. In TinyOS programming, dynamic memory allocation is advised not to use in order to maintain a stable working memory

that is essential for constrained memory devices like sensor motes. *Data* — might be also called *program constant* — is the storage of global and static variables which are initialized. *Bss* contains global uninitialized variables and zero-initialized static variables.

When an application is developed for a sensor mote running TinyOS, the compiler translates that application into CPU-readable instructions and static data. Then, CPU instructions are linked with other required components such as libraries or the most important component — TinyOS operating system. Next, the whole linked files together with static data are aggregated into different segments of an executable file — *TinyOS image* which can be deployed directly into sensor mote. In *TinyOS image*, CPU instructions are placed at *.text segment*, initialized global static data go into *.data segment* and uninitialized global data or zero-initialized static data are stored at *.bss segment*. In our evaluation, in order to measure working memory and programming memory, we examined this *TinyOS image* to extract the size of *.text*, *.data* and *.bss* segment. Hence, from this we can calculate the working memory and the programming memory as follows :

$$\textit{Working memory (RAM)} \simeq \textit{.bss size} + \textit{.data size}$$
$$\textit{Programming memory (ROM)} = \textit{.text size} + \textit{.data size}$$

Since RAM usage also consists of *Stack segment* which is not reflected in *TinyOS image*, our RAM estimation does not provide an overall estimation. However, a relative comparison between our module implementation can be achieved through this estimation and more importantly the memory overhead of our implementation toward the whole sensor mote.

To examine the *TOS image*, we used *objdump* [5] tool to extract the size of each module and the result is presented in Table 5.1. As we can see from the table, *Fragmented message key exchange* certainly consumes more ROM and RAM compared with *Full message key exchange*. This is certainly due to additional packet processing and data buffer to split key exchange message.

Next, we compare memory usage between our key exchange module and the whole system as in Table 5.2. As shown in the table, in our specific key exchange module, there is higher memory requirements in *Fragmented message key exchange* than *Full message key exchange*. But in the whole system, the result is the other way around. The reason lies in reduction in maximum link layer packet. As *Fragmented message key exchange* is used, maximum link layer packet is set to a lower value which results in smaller sending buffer. The sending buffer decrease happens not only in link layer but in other components which rely on on link layer service also. Compared

| | Full message key exchange | | Fragmented message key exchange | |
|---|---|---|---|---|
| | Execution opt. | Memory opt. | Execution opt. | Memory opt. |
| .bss | 1940 | 1362 | 2011 | 1366 |
| .data | 1 | 1 | 1 | 1 |
| .text | 12286 | 8160 | 12740 | 8552 |
| ROM | 12287 | 8161 | 12741 | 8553 |
| RAM | 1941 | 1363 | 2012 | 1367 |

Table 5.1: Memory overheads. All figures are in bytes

| | | Key exchange | | Whole system | |
|---|---|---|---|---|---|
| | | ROM | RAM | ROM | RAM |
| Full message key exchange | Execution opt. | 12287 | 1941 | 42752 | 4558 |
| | Memory opt. | 8161 | 1363 | 38312 | 3916 |
| Fragmented message key exchange | Execution opt. | 12741 | 2012 | 43528 | 4278 |
| | Memory opt. | 8553 | 1367 | 39276 | 3636 |

Table 5.2: Memory usage between *Key exchange module* and the whole system. All the figures are in bytes.

to the whole system, our key exchange component is in charge of around 45% RAM consumption in *execution time optimization* version and about 27% in *memory optimization.*

### 5.2.3 Energy consumption

Key exchange module is the most energy-consuming in our implementation since it contains long running and processor consuming operations. Hence, by knowing exact consumed energy helping developers to estimate their sensor nodes' survival time.

As sensor mote consumes a little energy, in order to accurately measure energy consumption, we wired our sensor mote to a 0.15 Ω resistor in serial connection and both of them are connected to power source which provides a stable 3.9V DC. Our set up can be visualized as in Figure 5.3. By recording the resistor's voltage $U$, using the following formula we can estimate energy consumption $E$ in one time unit

$$E = (3.9 * U - U * U)/0.15.$$

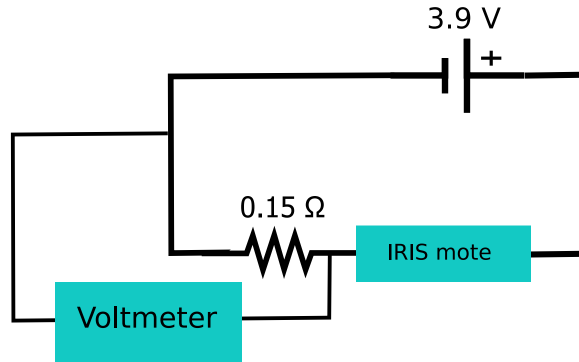


Figure 5.3: Energy experiment setup

We used a voltmeter to sample the resistor's voltage every one second. The voltmeter records every 50$ms$ and report the average voltage for each second. From the average voltage and with the above formula, the total power consumption during execution time can be computed.

**Experiment result** We conducted a serie of experiments with two versions of *Key exchange module* and two aforementioned optimizations. Figure 5.4 and Figure 5.5 present the voltage measured on the resistor in case of *the full message key exchange* was used with *execution time optimization* and *memory optimization* respectively. As seen in the graph, we can clearly see the impact of our module in sensor mote. The voltage increase which results in higher power consumption started when the sensor mote starts exchange key. High voltage remains when sensor mote verifying certificates and calculating shared key. The duration that the voltage remains high indicates the execution time of our module. By measuring the area underneath the curve in the graph, we can calculate the energy consumed for the whole key exchange. We found that the total energy spent for doing key exchange with *Full message key exchange version* with *execution time optimization* is 1.87$J$ and similarly, 5.46$J$ was consumed in order to do key exchange with *memory optimization*.

Similar experiments were carried out with the *Fragmented message key exchange* version. In this experiment, we used a payload of 40 bytes to carry key exchange message. Thus, it is required 3 packets in total to send one key exchange message. Figure 5.6 and Figure 5.7 correspondingly presents the voltage sample of 0.15Ω resistor with *execution time optimization* and *memory optimization*. Using the same aforementioned method, we deduced the energy consumed by this version with *execution time* and *memory optimiza-*

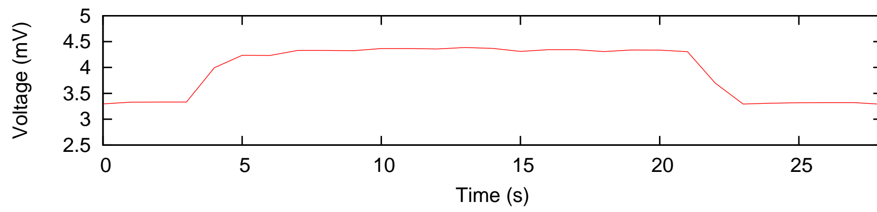


Figure 5.4: Power consumption *Full message key exchange* with *execution time optimization*

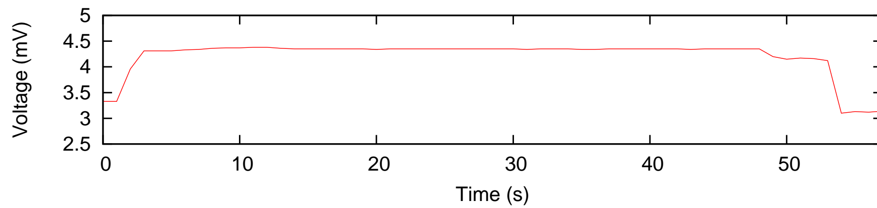


Figure 5.5: Power consumption for *Full message key exchange* with *memory optimization*.

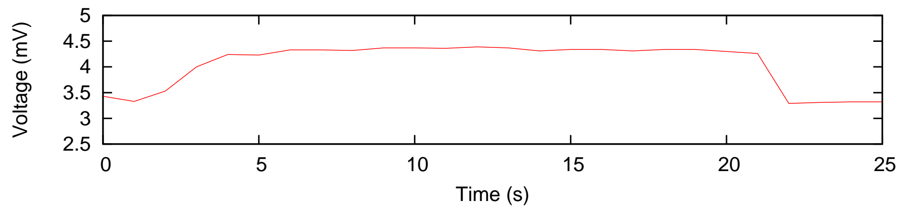


Figure 5.6: Power consumption for *Fragmented message key exchange* with *execution time optimization*.

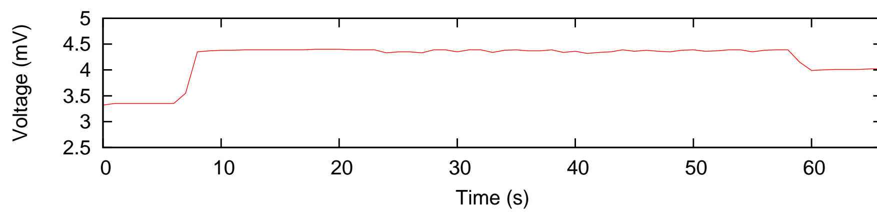


Figure 5.7: Power consumption for *Fragmented message key exchange* with *memory optimization*.

| Optimization | Execution time | Energy consumption | Key exchanges number |
|---|---|---|---|
| Execution time | 17.34 s | 1.87 J | 2076 |
| Memory | 50.2 s | 5.46 J | 717 |

Table 5.3: Energy consumption for Full message key exchange

*tion* are $2.02J$ and $5.69J$ respectively. As we can see, this version costs more energy than *Full message key exchange* due to extra processing on handling fragmented packets (splitting and joining). However, compared with ECC operations, these extra computations only cost a small portion of energy in the whole key exchange operation.

In order to examine the impact of energy consumption caused by key exchange module in sensor node's life, we count the number of key exchanges which can be performed during sensor node's survival time. Like other sensor mote, our IRIS sensor mote is powered by two AA batteries. Let us take Energizer EV15 battery [3] as an example. From Figure 5.4 to Figure 5.7 we can see that, when sensor motes perform key exchange operation, the voltage of the $0.15\Omega$ resistor could be approximately to 4.3 mV, which leads to the current of the whole circuit is nearly 28 mA. According to the performance datasheet of Energizer EV15 [3], with this discharge current, a sensor mote can roughly operate in 10 hours. With this 10 hours, a sensor node can perform about 2076 key exchanges with *execution time optimization* and 717 ones with *memory optimization* if the sensor mote uses *full message key exchange*. Similarly, in case *fragmented message key exchange* is in use, sensor mote can approximately execute 1986 and 697 key exchanges corresponding to *execution time* and *memory optimization*. Knowing the amount of key exchanges can help developers in deciding how large the sensor network might be. More precisely, the network size can be limited if a sensor mote can talk with only a certain number of communicating entities in their life. However, as our key exchange is carried only with two sensor motes and there was no packet routing in between, this estimated energy consumption is only a fraction of the whole energy a sensor mote might spend. Moreover, power consumption happens not only at two communicating sensor motes but key exchange might cause energy spending at other intermediate sensor mote if routing between two involved sensor motes is required. In summary, Table 5.3 and Table 5.4 presents execution time, energy and possible number of key exchanges for *Full message key exchange* and *Fragmented message key exchange* respectively.

| Optimization | Execution time | Energy consumption | Key exchanges number |
|---|---|---|---|
| Execution time | 18.12 s | 2.02 J | 1986 |
| Memory | 51.63 s | 5.69 J | 697 |

Table 5.4: Energy consumption for Fragmented message key exchange

# 5.3 Packet handling module

## 5.3.1 Execution time

In order to measure the delay caused by using *Packet handling module*, we measured the time needed for a packet to be completely sent over radio interface. In our implementation, as described in 4.2.1, we used 8 additional bytes for MAC and 2 bytes for *freshness counter*, so originally the maximum payload is 102 bytes but now only 92 bytes. We performed experiments with one sensor mote sending packets with various payload sizes — from minimum 1 byte to maximum 92 bytes. Then we measured the duration that packets are completely sent. In order to compare the latency when packets were passed through *Packet handling packet*, packets were sent in two way: initial plain sending and sending in authenticated way using our implementation. The test was conducted 10 times and we noticed not much difference between each run's result. The results are taken average of and shown as in Figure 5.8.

As seen from chart, runtime for computing MAC and checking freshness is larger than sending just with plain packets. The reason is that the sensor node running with our security implementation actually sends more data (additional MAC and *freshness counter*) than in plain way. Nevertheless, the main reason is of course the cryptographic operation. As carried payload grows, cryptographic operations are performed on large amount of data which in turn results in longer execution time. The running time difference can go up to $0.06s$ if the payload size reaches nearly the maximum payload value — 92 bytes. This latency in sending probably might not cause much trouble if the security solution is meant for end-to-end transmission since security processing only takes place at two ends. But with hop-by-hop security solution, this might cause an unwanted delay in communicating.

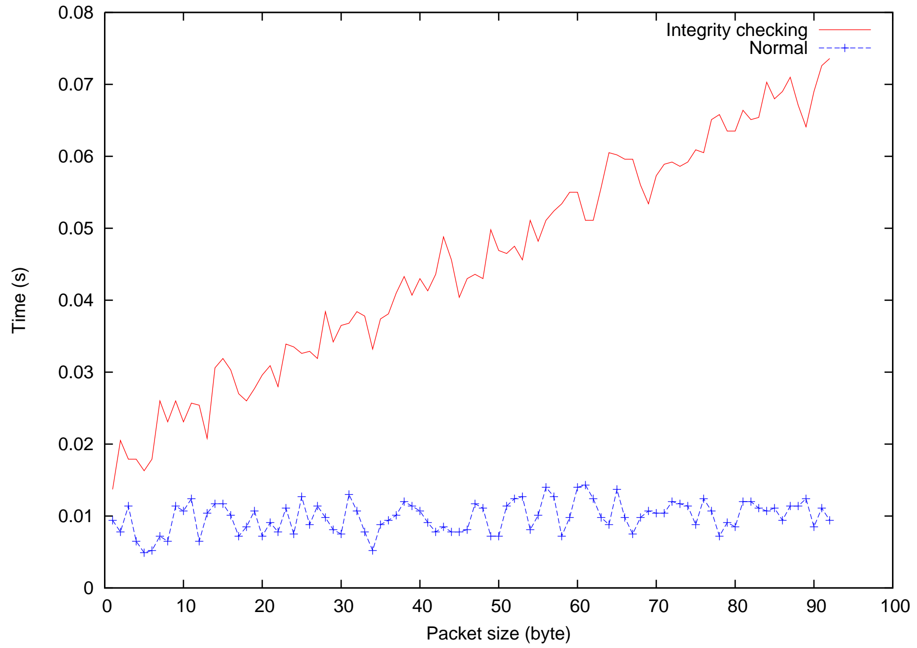


Figure 5.8: Time spent for sending one packet

| .bss | .data | .text | ROM | RAM |
|---|---|---|---|---|
| 453 bytes | 256 bytes | 6720 bytes | 6976 bytes | 709 bytes |

Table 5.5: Memory consumption in Packet handling module.

## 5.3.2 Memory consumption

We performed the same method to measure memory consumption of *Packet handling module*. The result is presented in Table 5.5. Compared with the whole system, *Packet handling module* consumes a small portion of memory. With the least RAM-consuming version — *Fragmented message key exchange* and *memory optimization* , *packet hanndling module* accounts for 19.5% RAM consumption and 17.8% ROM consumption.

# Chapter 6

# Conclusion and Future work

This chapter presents the conclusion of the thesis work and suggests some directions for future work.

## 6.1 Conclusion

The thesis has presented a flexible and TinyOS friendly security solution for IRIS sensor mote. Our security solution has solved two-tier security problem in wireless sensor network: Key management problem and integrity, authenticity and freshness problem.

For key management problem, we aim at offering a solution which allows two arbitrary sensor nodes can communicate to mutually establish shared secret key. We have considered and examined theoretically two designs and select ECDH as our cryptographic solution.

For integrity, authenticity and freshness problem, our objective is to implement flexible module which can be used at an *end-to-end* or *hop-by-hop* security implementation. We did some surveys and selected sensor node friendly cryptographic algorithms — CBC-MAC and SkipJack — to use in our implementation.

Finally, we implemented the designed solution and evaluated its performance. The key management problem is implemented as *Key exchange module*. We tested and selected two versions of this module : memory optimized and runtime optimized. The performance of key management solution requires at least 1363 bytes of RAM or runs at least 17.34s. For authenticity, integrity and freshness solution, our *Packet handling module* occupies a small amount

of memory and performs at an acceptable speed which grows linearly with the message length.

## 6.2 Future work

Several interesting future directions can improve or extend our security implementation to give a better performance. First, as we can recognize from the performance result of *key exchange module*, our *key exchange module* run on IRIS motes much slower than the estimated time on MicaZ motes. As seen in Table 3.1 in Section 3.3.2, we estimated that exchanging key can take at least about 9.8s which is half of the minimum runtime in our current IRIS platform 17.34s (as seen in Table 5.3). The reason is that the library TinyECC that we used is specified for MicaZ motes but not for IRIS motes. Hence, we can customize TinyECC for IRIS platform in order to improve sensor motes' performance. Since both MicaZ and IRIS use 8-bit and similar processors (Atmegal128), optimizing TinyECC for IRIS platform might reduce runtime to approximately the estimated time on MicaZ platform.

Secondly, as our implementation currently provides only authenticity, integrity and freshness for transmission layer, our implementation can be extended with confidentiality functionality. There are cases where confidentiality in WSNs is needed. For example, in battlefield surveillance network, developers want to prevent other illegitimate users from collecting battlefield information by using their sensor network.

Thirdly, even though our implementation has been already measured performance, all of the experiments were conducted with indoor environment and with a small size network. As the environmental events such as changing temperature can greatly affect sensor motes and reduce the performance. In order to gain more precise evaluation, our implementation should be tested with larger network size with outdoor environment.

Fourthly, as wireless sensor networks are unattended, several security threats such as node capture, false node are hardly discovered. Unfortunately, such attacks also cannot be prevented by using our security implementation. In fact, they can invalidate security solution. For example, with legitimate keys from any compromised nodes, false nodes can send "legitimate" message in presence of any security solution. Thus, since detecting compromised and false nodes are as important as securing transmission data, a mechanism [24][34] to detect malicious nodes should be developed to provide a stronger protection for WSNs.

# Bibliography

[1] A Cost-Based Security Analysis of Symmetric and Asymmetric Key Lengths. Tech. rep., RSA Laboratories.

[2] Defendec's Homepage. http://www.defendec.com/ (Last checked June 29th, 2010).

[3] Energizer E15 Product Datasheet. http://data.energizer.com/PDFs/EV15.pdf (Last checked June 29th, 2010).

[4] *Iris sensor mote*. http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=135\%3Airis (Last checked June 29th, 2010).

[5] objdump - Info from object files. Linux manpage.

[6] TinyOS Source Code, May 20010. http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/chips/rf2xx/layers/MessageBufferLayerP.nc?content-type=text\%2Fplain(LastcheckedJune29th,2010).

[7] IEEE 802.15.4-2006 Standard. Tech. rep., IEEE Standard for Information technology, 2006.

[8] ECRYPT2 Yearly Report on Algorithms and Keysizes. Tech. rep., Network of Excellence - Information and Communication Technologies, 2008 - 2009.

[9] Recommended Elliptic Curves for Federal Government Use. Tech. rep., National Institute of Standards and Technology, August 1999.

[10] Standards for efficient cryptography - SEC 2: Recommended elliptic curve domain parameters. Tech. rep., Certicom Research, September 2000.

[11] Adrian, H. C., Perrig, A., and Song, D. Random Key Predistribution Schemes for Sensor Networks. In *In IEEE Symposium on Security and Privacy* (2003), pp. 197–213.

[12] Aranha, D. F., Oliveira, L. B., Lopez, J., and Dahab, R. NanoPBC: Implementing Cryptographic Pairings on an 8-bit Platform. In *Conference on Hyperelliptic curves, discrete Logarithms, Encryption, etc., Frutillar, Chile* (2009).

[13] Barrett, P. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Proceedings on Advances in cryptology—CRYPTO '86* (London, UK, 1987), Springer-Verlag, pp. 311–323.

[14] Beker, H., and Piper, F. *Cipher Systems: The Protection of Communications*. John Wiley & Sons, 1982.

[15] Bellare, M., T, J. K., and Rogaway, P. The Security of the Cipher Block Chaining Message Authentication Code, 2001.

[16] Biham, E., Biryukov, A., and Shamir, A. Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials. *J. Cryptol. 18*, 4 (2005), 291–311.

[17] Brickell, E. F., Denning, D. E., Kent, S. T., Maher, D. P., and Tuchman, W. SKIPJACK review: Interim Report. 119–130.

[18] Coppersmith, D. The Data Encryption Standard (DES) and its strength against attacks. *IBM J. Res. Dev. 38*, 3 (1994), 243–250.

[19] Daemen, J., and Rijmen, V. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer Verlag, Berlin, Heidelberg, New York, 2002.

[20] Deepakumara, J., Heys, H. M., and Venkatesan, R. Performance Comparison of Message Authentication Code (MAC) Algorithms for the Internet Protocol Security (IPSEC).

[21] Eschenauer, L., and Gligor, V. D. A Key-Management Scheme for Distributed Sensor Networks. In *In Proceedings of the 9th ACM Conference on Computer and Communications Security* (2002), ACM Press, pp. 41–47.

[22] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation* (New York, NY, USA, 2003), ACM, pp. 1–11.

[23] Hankerson, D., Menezes, A. J., and Vanstone, S. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[24] Junior, W. R. P., de Paula Figueiredo, T. H., Wong, H. C., and Loureiro, A. A. F. Malicious Node Detection in Wireless Sensor Networks. *Parallel and Distributed Processing Symposium, International 1* (2004), 24b.

[25] Karlof, C., Sastry, N., and Wagner, D. TinySec: a link layer security architecture for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems* (New York, NY, USA, 2004), ACM, pp. 162–175.

[26] Korhonen, J. Wang, Y. Effect of packet size on loss rate and delay in wireless links. *Wireless Communications and Networking Conference, 2005 IEEE 3* (2005).

[27] Krawczyk, H., Bellare, M., and Canetti, R. HMAC: Keyed-Hashing for Message Authentication. *RFC 2104* (1997), 1–12.

[28] Lai, B., Kim, S., and Verbauwhede, I. Scalable Session Key Construction Protocol for Wireless Sensor Networks. In *In IEEE Workshop on Large Scale RealTime and Embedded Systems (LARTES* (2002), p. 7.

[29] Law, Y. W., Doumen, J., and Hartel, P. Survey and Benchmark of Block Ciphers for Wireless Sensor Networks. *ACM Trans. Sen. Netw. 2*, 1 (2006), 65–93.

[30] Levis, P., and Gay, D. *TinyOS Programming*, 1 ed. Cambridge University Press, April 2009.

[31] Liu, A., and Ning, P. TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 245–256.

[32] Liu, D., and Ning, P. Establishing Pairwise Keys in Distributed Sensor Networks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security* (New York, NY, USA, 2003), ACM, pp. 52–61.

[33] Malan, D. J., Welsh, M., and Smith, M. D. A Public-Key Infrastructure for Key Distribution in TinyOS based on Elliptic Curve Cryptography. In *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on* (2004), pp. 71–80.

[34] Mukherjee, P., and Sen, I. Using Learned Data Patterns to Detect Malicious Nodes in Sensor Networks.

[35] Oliveira, L. B., Scott, M., López, J., and Dahab, R. TinyPBC: Pairings for Authenticated Identity-Based Non-Interactive Key Distribution in Sensor Networks. Cryptology ePrint Archive, Report 2007/482, 2007. `http://eprint.iacr.org/` (Last checked June 29th, 2010).

[36] Paul, S., and Preneel, B. A New Weakness in the RC4 Keystream Generator and an Approach to Improve the Security of the Cipher. 2004, pp. 245–259.

[37] Perrig, A., Szewczyk, R., Wen, V., Culler, D., and Tygar, J. D. SPINS: Security Protocols for Sensor Networks. In *Wireless Networks* (2001), pp. 189–199.

[38] Schneier, B. Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop* (London, UK, 1994), Springer-Verlag, pp. 191–204.

[39] Szewczyk, R., Polastre, J., Mainwaring, A., and Culler, D. Lessons From A Sensor Network Expedition. pp. 307–322.

[40] Watro, R., Kong, D., Cuti, S.-f., Gardiner, C., Lynn, C., and Kruus, P. TinyPK: Securing Sensor Networks with Public Key Technology. In *SASN '04: Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks* (New York, NY, USA, 2004), ACM, pp. 59–64.

[41] Zuger, R. Paging in TinyOS. Tech. rep., Swiss Federal Institute of Technology Zurich, August 2006.