UNIVERSITÀ DI PISA
FACOLTÀ DI INGEGNERIA

Corso di Laurea Specialistica in Ingegneria Informatica
Curriculum Networking e Multimedia

An IEEE 802.15.4 security sublayer implementation for CC2420

SUPERVISORS:

Prof. Gianluca Dini
Ing. Alessio Bechini

CANDIDATE:

Roberta Daidone

Anno Accademico 2009-2010

To my family

Abstract

During the last ten years, the presence of sensor networks in common life has become pervasive and sensor nodes are currently used in many areas of interest. One of the most common radio communication protocol designed for Personal Area Networks (PAN) is described by the IEEE 802.15.4 standard, according to which data communication among devices can also be protected on a per frame basis, so making it possible to assure data authenticity and confidentiality, and security mechanisms can be configured in a flexible and effective way.

In this thesis work, the IEEE 802.15.4 security sublayer has been implemented. In particular, the TinyOS implementation for the *tmote sky* mote and the CC2420 chipset have been considered. The main goal of this work is to extend the above mentioned MAC layer implementation in order to make the IEEE 802.15.4 security mechanisms available, that is sending and receiving both ciphered and authenticated frames, by means of the security features provided by the CC2420 chipset.

All security data structures and procedures have been implemented, so making it possible to deal with different cryptographic keys usage and retrieval modes. During the development phase, some problems strictly related to computational and memory capacity shortage have been faced and properly addressed. Finally, the implemented security sublayer has been tested and evaluated by means of a simple application, which sends secured packets whose payload changes both in content and size.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the last few years, the concept of sensors network has come to be the new horizon of networking, looking forward to the idea of pervasive computing. These systems are meant to support a large number of applications, many of which are security sensitive, such as medical or environmental monitoring instruments.

A sensor node (or mote) is a small battery supplied device endowed with a sensing system able to collect various kind of data, a processing system which orates information and a communication system which sends and shares data with other motes. So, the big challenges are power consumption and complexity, as well as costs.

In order to avoid retransmissions and saving battery power as well, communications should be reliable. Besides, it could be necessary or desirable to guarantee the origin of messages (authenticity), protect data from unauthorized accesses (confidentiality) and prevent unwanted replays of received messages (anti-replay). By doing so, it is more difficult for an adversary to modify or to inject messages in order to alter data communications.

A technology suitable for sensors networks is described by the IEEE 802.15.4 standard: it describes wireless Medium Access Control (MAC) and Physical (PHY) layers specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs). IEEE 802.15.4 offers a wide range of security options: counter mode encryption (CTR), authentication only mode (CBC-MAC) and a combination of them (CCM). Cryptography is based on AES (Advanced Encryption

Standard) 128 bits symmetric-key cryptography. Thanks to the variety offered by the standard, it is possible to tune the security level appropriate for a certain application. The security level is recognized by exploring the Auxiliary Security Header and it is handled by means of specific data structures. These contain security parameters and permit to validate and decipher messages in a flexible and dynamic manner.

In this thesis work all the data structures and the security procedures described by the standard, the four different Key Identifier Modes and their management code, the routines support for device identificaton as well as keys retrieval and management have been implemented.

During implementation work Tmote sky motes have been used since they rely on an open source platform designed for experimentation within the research community and they also include the CC2420 chipset which provides extensive hardware support for packet handling and security mechanisms. The nesC developing language and the TKN branch of the TinyOS environment has been used [8].

In order to test the correctness of communications between a network coordinator and some Reduced Function Devices (RFDs), it has been realized an application which sends secured and unsecured packets and manages different Security Levels according to a certain Key Identifier Mode. It is worth noting that the device architecture affected all the project phases, especially during testing and debugging. In fact, these phases turned out to be quite troublesome because of the lack of powerful development environments as well as debugging instruments.

The rest of the thesis is organized as follows: next chapter (Chapter 2) provides a brief description of the IEEE 802.15.4 standard and its security features, while Chapter 3 describes the main features of TelosB motes and CC2420 chipset. Chapter 4 discusses the implementation of the security mechanisms. Chapter 5 reports perfomances evaluation and, finally, Chapter 6 draws some conclusive remarks.

Chapter 2

IEEE 802.15.4

2.1 Overview

IEEE 802.15.4 describes a protocol for communication among low-power devices in Wireless Personal Area Networks (WPANs).

Two different device types can participate in an IEEE 802.15.4 network: Full-Function Device (FFD) and Reduced-Function Device (RFD). A network comprises at least one coordinator, that is a FFD capable of relaying messages from other devices. Plus, one coordinator is elected as the Personal Area Network (PAN) Coordinator. An RFD, on the other hand, is intended to do extremely simple applications, consequently, it can be implemented using minimal resources and memory capacity. An RFD is associated to a single PAN coordinator at a time.

According to the application requirements, a network may be organized in either two topologies: star topology and peer-to-peer topology (Figure 2.1). The star topology mirrors the classic host-client network paradigm: all the messages from devices must pass through the single central controller, called the PAN coordinator; every device has to associate with the PAN coordinator to be part of the network. Also in peer-to-peer topologies there is a PAN coordinator, but any device is allowed to communicate with any other device (as long as they are in range of one another) creating a mesh network if they are supposed to do so.

The physical medium is accessed through a Carrier Sense Multiple Access

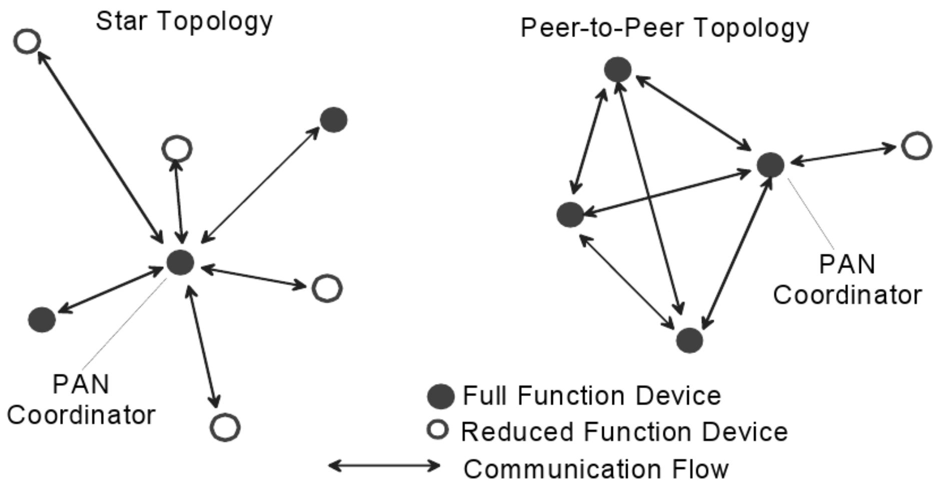


Figure 2.1: Star and peer-to-peer topology examples

protocol, with Collision Avoidance (CSMA/CA). It can be used for channel access either in an un-slotted version or in a time-slotted version with beacon frames to keep motes synchronized. In time-slotted version, between every two beacons, each device competes with others during the Contention Access Period (CAP), while guaranteed time slots can be assigned during the Contention Free Period (CFP).

Common data transmissions use unallocated slots when beaconing is in use; confirmations do not follow the same process. Acknowledgement messages may be optional under certain circumstances, in this case a success assumption is made. Whatever the case, if a device is unable to process a frame at a given time, it simply does not confirm its reception: timeout-based retransmissions can be performed a number of times, following after that a decision of whether to abort or to keep trying.

IEEE 802.15.4 specifies both PHY and MAC layer. The PHY layer activates and deactivates the radio transceiver, monitors energy detection and link quality indicator for received packets, controls the Clear Channel Assessment (CCA) for CSMA/CA and selects channel frequency and data transmission and reception.

The MAC layer allows the transmission of the MAC frames through the physical channel. It also offers beacon management, channel access, guaranteed time slot management, frame validation, acknowledged frame delivery, association and disassociation. In addition, the MAC sublayer provides hooks for implementing application-appropriated security mechanisms. [6]

2.2 MAC frame structure

According to the IEEE 802.15.4 standard [9], trasmissions are organized into frames, which have been designed trying to keep complexity at a minimum. The standard, while still assuring robustness for transmissions on a noisy channel, provides four different frame structures that have specific functions:

- Data;
- Acknowledgment;
- Beacon;
- MAC Command.

Beacon frames are transmitted by a coordinator to implement significant power saving modes or when attempting to establish a network, Data frames and Acknowledgment frames are used for data transfers and to confirm successful frame reception respectively. Finally, MAC command frames are used to handle all MAC peer entity control transfers, sending low-level commands from one node to another. Each further protocol layer is added to the structure with layer-specific headers and footers.

2.2.1 Data frame structure

The data frame structure is similar to the other three frame types, its content is originated by the upper layers. The MAC payload is prefixed with the MAC Header (MHR) and appended with the MAC Footer (MFR). The MHR contains a two octets Frame Control Field, a single octet Data Sequence Number (DSN), Addressing Fields whose size changes according to the addressing mode, and, optionally, a variable-length Auxiliary Security Header (ASH).

The MFR is composed of a 16-bit Frame Check Sequence (FCS). The MHR, the MAC payload, and the MFR together form the MAC data frame (Figure

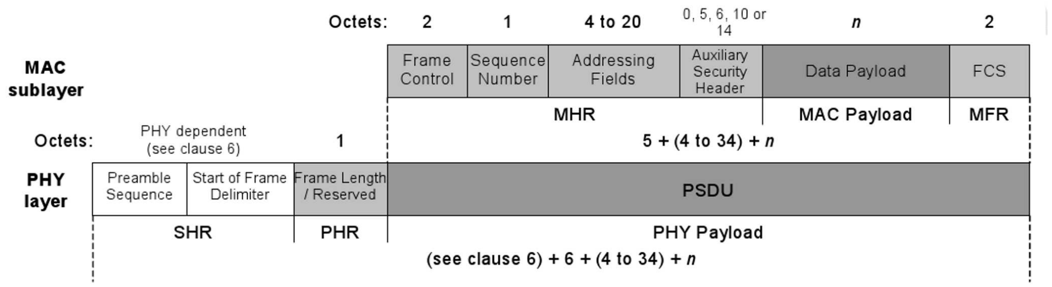
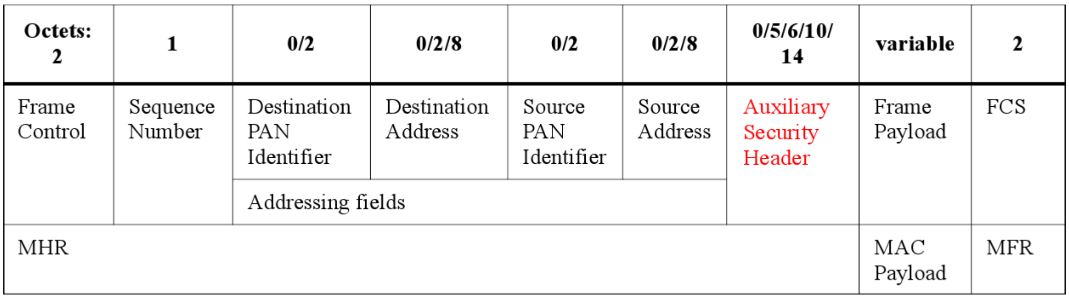


Figure 2.2: Data Frame and PHY Packet

2.2). These fields compose the PHY payload. The PHY packet is completed by the Synchronization Header (SHR) and the PHY header (PHR). The SHR contains a preamble sequence to allow the receiver to acquire and synchronize the incoming signal and a start of frame delimiter that signals the end of the preamble. Besides, the PHR carries the frame length byte, which indicates the length of the PHY payload.

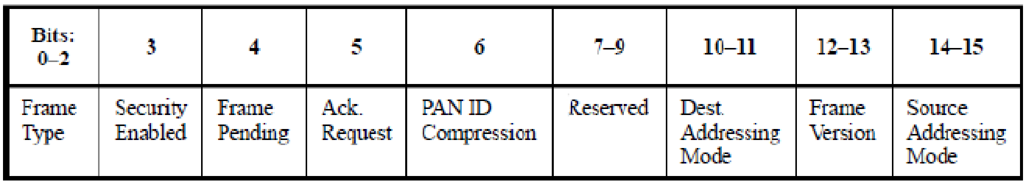
2.3 MAC header and auxiliary security header



Octets: 2	1	0/2	0/2/8	0/2	0/2/8	0/5/6/10/14	variable	2
Frame Control	Sequence Number	Destination PAN Identifier	Destination Address	Source PAN Identifier	Source Address	Auxiliary Security Header	Frame Payload	FCS
		Addressing fields						
MHR							MAC Payload	MFR

Figure 2.3: MAC Frame

The MAC frame is composed of the MAC header, the MAC payload, and the MAC Footer. However, some fields like the addressing fields or the security header might not be included in all frames, so it has a variable length, as shown in Figure 2.3.



Bits: 0–2	3	4	5	6	7–9	10–11	12–13	14–15
Frame Type	Security Enabled	Frame Pending	Ack. Request	PAN ID Compression	Reserved	Dest. Addressing Mode	Frame Version	Source Addressing Mode

Figure 2.4: Frame Control Field

2.3.1 Frame control field

The Frame Control Field is a two octets field. It contains information defining the frame type, addressing fields type, and other control flags. It is formatted as illustrated in Figure 2.4.

2.3.1.1 Frame type subfield

The Frame Type subfield contains the 3-bit encoding of the current frame type (command, beacon, acknowledgment or data).

2.3.1.2 Security enabled subfield

The Security Enabled subfield is set to one if the frame is protected by the MAC security sublayer and must be set to zero otherwise. The Auxiliary Security Header field of the MHR is present only if this subfield is set to one.

2.3.1.3 Frame pending subfield

The Frame Pending subfield is set to one if the device sending the frame has more data for the recipient. It is used only in beacon frames or frames transmitted either during the CAP by devices operating on a beacon-enabled PAN or at any time by devices operating on a nonbeacon-enabled PAN. Otherwise, it shall be set to zero on transmission and ignored on reception.

2.3.1.4 Acknowledgment subfield

The Acknowledgment Request subfield is one bit in length and specifies whether an acknowledgment is required from the recipient device when receiving a data or MAC command frame. If this subfield is set to one, the recipient device sends an acknowledgment frame only if, on reception, the frame passes the third level of filtering. If this subfield is set to zero, the recipient device will never send acknowledgment frames.

2.3.1.5 PAN ID compression subfield

The PAN ID Compression subfield specifies whether the MAC frame to be sent contains only one of the PAN identifier fields when both source and destination addresses are present. If this subfield is set to one and both the source and destination addresses are present, the frame has to contain only the Destination PAN Identifier field, and the Source PAN Identifier field is assumed to be equal to the destination's.

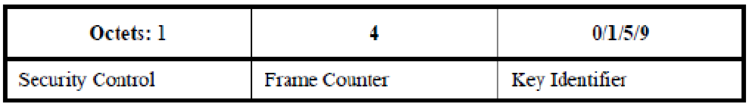
2.3.1.6 Frame version subfield

The Frame Version subfield specifies the 2-bit encoding of the frame version.

2.3.1.7 Addressing mode subfields

Finally, the Destination Addressing Mode and the Source Addressing Mode subfields indicate if address fields contain 16-bit short addresses or 64-bit extended addresses.

2.3.2 Auxiliary security header



Octets: 1	4	0/1/5/9
Security Control	Frame Counter	Key Identifier

Figure 2.5: Auxiliary Security Header

The Auxiliary Security Header has a variable length and, as shown in Figure 2.5, it contains information required for security processing, including the Security Control Field, the Frame Counter Field, and the Key Identifier Field. It is present only if the Security Enabled subfield of the Frame Control field is set to one. It is formatted as illustrated in Figure 2.6.

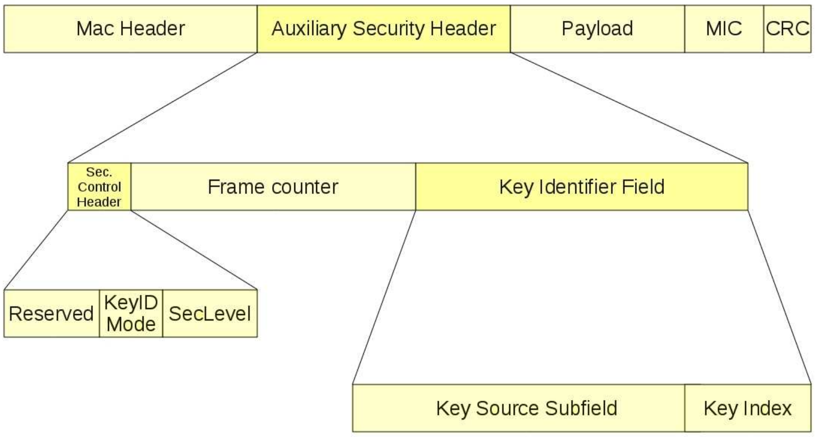
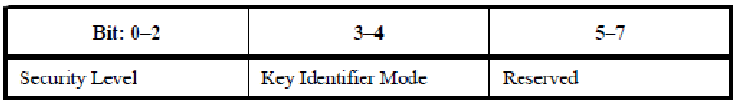


Figure 2.6: Auxiliary Security Header Structure

2.3.2.1 Security control subfield



Bit: 0–2	3–4	5–7
Security Level	Key Identifier Mode	Reserved

Figure 2.7: Security Control Subfield

The 8-bit Security Control field is used to provide information about what kind of protection is applied to the frame. The Security Control Field has to be formatted as shown in Figure 2.7.

The Security Level subfield is three bit in length and indicates the actual frame protection provided. This value can be adapted on a frame-by-frame basis and allows for varying levels of data authenticity (to allow minimization of security overhead in transmitted frames where required) and for optional data confidentiality. Table 2.1 summarizes all security levels available.

The Key Identifier Mode subfield is two bit in length and indicates whether the key used to protect the frame can be derived implicitly or explicitly. Furthermore, it is used to indicate the particular representation of the Key Identifier field, if the key is derived explicitly. The Key Identifier Mode subfield is set according to Table 2.2. The Key Identifier field of the Auxiliary Security Header is present only if this subfield has a value not equal to 0x00.

Security Level Identifier	**Security Control Field**	**Security Attributes**	**Data confidentiality**	**Data authenticity**
0x00	’000’	None	OFF	NO (M = 0)
0x01	’001’	MIC-32	OFF	YES (M = 4)
0x02	’010’	MIC-64	OFF	YES (M = 8)
0x03	’011’	MIC-128	OFF	YES (M = 16)
0x04	’100’	ENC	ON	NO (M =0)
0x05	’011’	ENC-MIC-32	ON	YES (M = 4)
0x06	’110’	ENC-MIC-64	ON	YES (M = 8)
0x07	’111’	ENC-MIC-128	ON	YES (M = 16)

Table 2.1: Security Level values and options

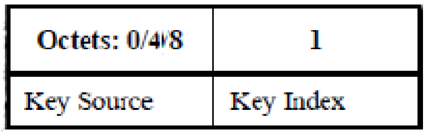
Key Identifier mode	**Security Mode subfield**	**Description**	**Key Identifier field length** (octets)
0x00	’00’	Key is determined implicitly from the originator and receipient(s) of the frame as indicated in the frame header.	0
0x01	’01’	Key is determined from the 1-octet Key Index subfield of the Key Identifier Field of the auxiliary security header in conjunction with macDefaultKeySource.	1
0x02	’10’	Key is determined explicitly from the 4-octet Key Source subfield and the 1-octet Key Index subfield of the Key Identifier Field of the auxiliary security header	5
0x03	’11’	Key is determined explicitly from the 8-octet Key Source subfield and the 1-octet Key Index subfield of the Key Identifier Field of the auxiliary security header.	9

Table 2.2: Key Identifier values and options

2.3.2.2 Frame counter field

The Frame Counter field is a 4-octets field representing the macFrameCounter attribute of the originator of a protected frame. It is used to assure replay protection.

2.3.2.3 Key identifier field



Octets: 0/4/8	1
Key Source	Key Index

Figure 2.8: Key Identifier Field

The Key Identifier field has variable length and is used for cryptographic protection of outgoing frames, either explicitly or in conjunction with implicitly defined side information. The Key Identifier field is present only if the Key Identifier Mode subfield of the Security Control field of the Auxiliary Security Header is set to a value different from 0x00. The Key Identifier field is formatted as illustrated in Figure 2.8.

The Key Source subfield, if present, is either four octets or sixteen octets in length, according to the value specified by the Key Identifier Mode subfield of the Security Control field, and indicates the originator of a group key. The Key Index subfield is one octet in length and allows unique identification of different keys having the same originator.

2.4 Security structures

The MAC sublayer is responsible for providing security services on specified incoming and outgoing frames, when requested by the higher layers. The information according to which is determined how to provide security is located in the security-related PIB (PAN Information Base) [9]. This security-related PIB is divided in seven structures:

- Key Table;

- Device Table;
- Minimum security level table;
- Frame counter;
- Automatic request attributes;
- Default key source;
- PAN coordinator address.

The Key table contains key-descriptors, which are keys with related key-specific information required for security processing.

The device table holds device-descriptors, containing device-specific addressing and security-related information which, combined with key-specific information from the key table, provide all the keying material needed to secure/unsecure frames.

The minimum security level table holds information regarding the minimum security level the device expects having applied by the originator of a frame, depending on frame type and, if it concerns a MAC command frame, the command frame identifier.

The four octets frame counter is used to provide replay protection and semantic security of the cryptographic building block used for securing outgoing frames. Such counter is an integer which is incremented every time an outgoing frame is secured. When the frame counter reaches its maximum value of 0xffffffff, the associated keying material can no longer be used, thus requiring all keys to be updated.

The Automatic Request table holds all the information needed to secure outgoing frames generated automatically and not as a result of a higher layer primitive, as is the case with automatic data requests.

The default key source is commonly shared between originator and recipient(s) of a secured frame, so that, when combined with additional information explicitly contained in the requesting primitive or in the received frame, it allows an originator or a recipient to determine the key required for securing or unsecuring the frame, respectively. The address of the PAN coordinator is an information commonly shared between all devices in a PAN. The code of the implementation can be seen in Appendix A.

2.5 Security modes

The 802.15.4 security layer is handled at the MAC layer, below application control. The application specifies its security requirements by setting the appropriate control parameters into the radio stack. If an application does not set any parameters, then security is not enabled by default. An application must explicitly enable security.

The specification does not support security for acknowledgement packets; other packet types can optionally support integrity protection and confidentiality protection. An application has a choice of security suites that control the type of security protection that is provided for the transmitted data. Each security suite offers a different set of security properties and guarantees, and ultimately different packet formats.

The 802.15.4 specification defines eight different security suites. We can broadly classify them by the properties they offer: no security (NO_SEC), encryption only (CTR), authentication only (CBC_MAC), and both encryption and authentication (CCM). Each category that supports authentication comes in three variants depending on the size of the MIC it offers. Each variant is considered a different security suite and has its own name. In fact, the Message Integrity Code (MIC) can be either four, eight, or sixteen bytes long. The longer the MIC is, the lower is the chance an adversary has to blind forgery by guessing an appropriate code.

An application indicates the chosen security suite in the MAC frame header. 802.15.4 radio chips control what security suite and keying information to use. The security material is the persistent state necessary to execute the security suite. The application must specify a boolean indicating whether security is enabled. If no security is requested, the packet is sent out as is.

On packet reception, the MAC layer consults the packet flags field in order to determine if any security suite has been applied to that packet. If no security is used, the packet is passed as is to the application. Otherwise, the appropriate security suite, such as key and replay counter, is applied to the incoming packet, presenting the application with an error message if the procedure fails somewhat. In the following sections will be provided more details about the above-mentioned security suites.

2.5.1 NO_SEC

This is the simplest security suite. Its inclusion is mandatory in all radio chips. It does not manage any security material and it does not provide any security guarantees.

2.5.2 CTR

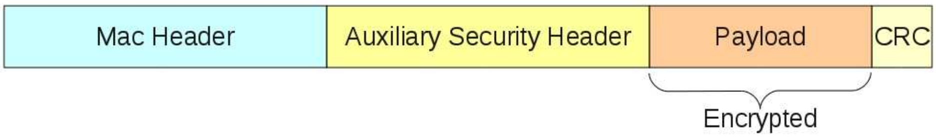


Figure 2.9: CTR Frame Format

This suite provides confidentiality protection using the AES block cipher with counter mode. To encrypt data under counter mode, the sender breaks the cleartext packet into 16-byte blocks and computes $c_i = p_i \oplus E_k(x_i)$. Each 16-byte block uses its own varying counter, which we call x_1. The recipient recovers the original plaintext by computing $p_i = c_i \oplus E_k(x_i)$. Clearly, the recipient needs the counter value x_1 in order to reconstruct p_i .

The x_1 counter, known as a nonce, is composed of a static flags field, the senders address, and three separate counters: a 4-byte frame counter that identifies the packet, a 1-byte key counter field, and a 2-byte block counter that numbers the 16-byte blocks within the packet. The sender increments the frame counter after encrypting each packet. When it reaches its maximum value, the radio returns an error code and the key has to be changed. The requirement is that the nonce must never repeat within the lifetime of any single key, and the role of the frame and key counters is to prevent nonce reuse. The block counter ensures that each block will use a different nonce value; the sender does not need to include it within the packet, since the receiver can infer its value for each block.

As shown in Figure 2.9, the sender includes in the packet three main components: the frame counter, key counter, and encrypted payload into the data payload field of the packet.

2.5.3 CBC_MAC

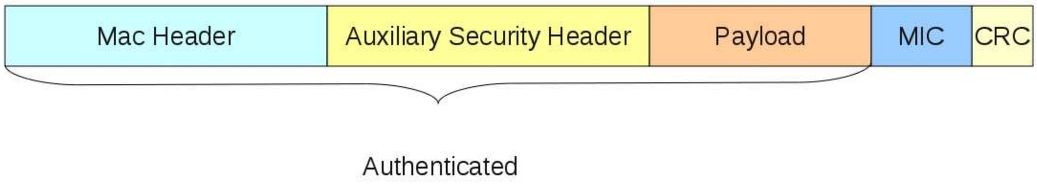


Figure 2.10: CBC_MAC Frame Format

This suite provides integrity protection using CBC_MAC. The sender can compute either a four, eight, or sixteen bytes Message Integrity Code (MIC) using the CBC_MAC algorithm, leading to three different variants. The MIC can only be computed by parties with the symmetric key and can protect packet headers as well as the data payload. The sender appends the plaintext data with the MIC.

The recipient verifies the MAC by computing the MAC and comparing it with the value included in the packet, deciding wether the packet is authenticated or not. Figure 2.10 shows the format of this packet.

2.5.4 CCM

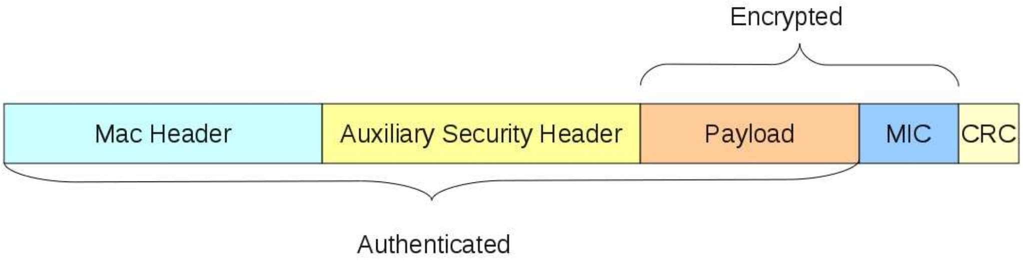


Figure 2.11: CCM Frame Format

This security suite uses CCM mode for both encryption and authentication. Broadly, it first applies integrity protection over the header and data payload using CBC_MAC and then encrypts both the data payload and the MIC using CTR. So CCM combines the fields from both the authentication and encryption operations such as the MIC and the frame counter. These fields serve the same function as above. As CBC_MAC, also CCM has three variants depending on the MIC size. Figure 2.11 shows the format of this packet.

2.6 Key identifier modes

The Key Identifier Mode subfield is two bits in length and indicates whether the key used in order to protect the frame can be derived implicitly or explicitly. Furthermore, it is used to indicate the particular representations of the Key Identifier field if it si derived explicitly. The Key Identifier Mode subfield shall be set to one of the values listed in the following sections. This field specifies the mode used in order to identify and retrieve the key used by the originator of the received frame. This parameter is ignored if the SecurityLevel parameter is set to 0x00.

There are four different modes: KeyIdMode0, KeyIdMode1, KeyIdMode2 and KeyIdMode3. The increment of the index corresponds to an enhancement of their peculiarities, but also of their complexity. The KeyIdMode0 just allows two or more nodes to send or receive data secured with a static, uniform, single key. The KeyIdMode1 provides more keys, selected from a single KeyTable thanks to the KeyIndex subfield of the Auxiliary Security Header. The keyTable is located at the default index written in the MacDefaultKeySource field of the PIB structure. This mode allows a key change just changing the index of the default key table. The two modes left are the most complex and complete. The KeyIdMode2 and KeyIdMode3 provide, besides the KeyIndex, also a KeySourceAddress which locates a different KeyTable filled with some different KeyDescriptors. So distinct devices are allowed to secure data using keys provided by different sources.

2.6.1 KeyIdMode0

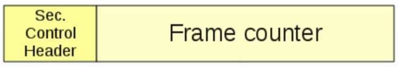


Figure 2.12: KeyIdMode0 Security Subheader Format

The key is determined implicitly from the originator and receipient(s) of the frame, as indicated by the frame header whose structure is shown in Figure 2.12.

2.6.2 KeyIdMode1

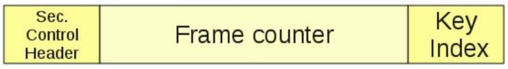


Figure 2.13: KeyIdMode1 Security Subheader Format

The key is determined from the 1-octet Key Index subfield of the Key Identifier field of the auxiliary security header in conjunction with macDefaultKeySource. The structure of the Auxialiary Security Header for this KeyIdMode is shown in Figure 2.13.

2.6.3 KeyIdMode2

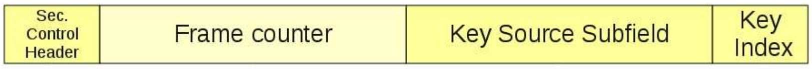


Figure 2.14: KeyIdMode2 Security Subheader Format

The key is determined explicitly from the 4-octet Key Source subfield and the 1-octet Key Index subfield of the Key Identifier field. The structure of the Auxialiary Security Header for this KeyIdMode is shown in Figure 2.14.

2.6.4 KeyIdMode3

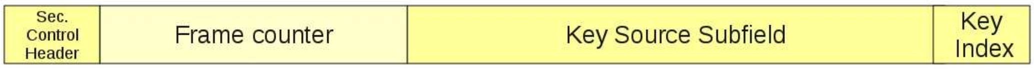


Figure 2.15: KeyIdMode3 Security Subheader Format

The key is determined explicitly from the 8-octet Key Source subfield and the 1-octet Key Index subfield of the Key Identifier field. The structure of the Auxialiary Security Header for this KeyIdMode is shown in Figure 2.15.

Chapter 3

Tmote sky and CC2420

3.1 Tmote sky

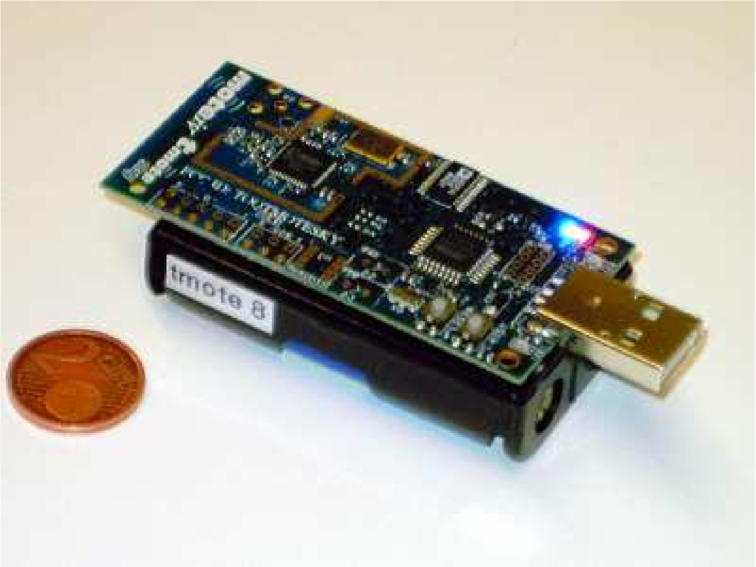


Figure 3.1: A simple Tmote Sky mote

A sensor node, also known as a 'mote', is a node used in a wireless sensor network. It can perform various kind of processing: it can monitor applications, it can gather sensory information and it can set up network connections with other motes forming a network. As can be seen in Figure 3.1, one point of strength about motes is their small dimension, combined with industry standards like USB, so providing flexible interconnection with peripherals [5].

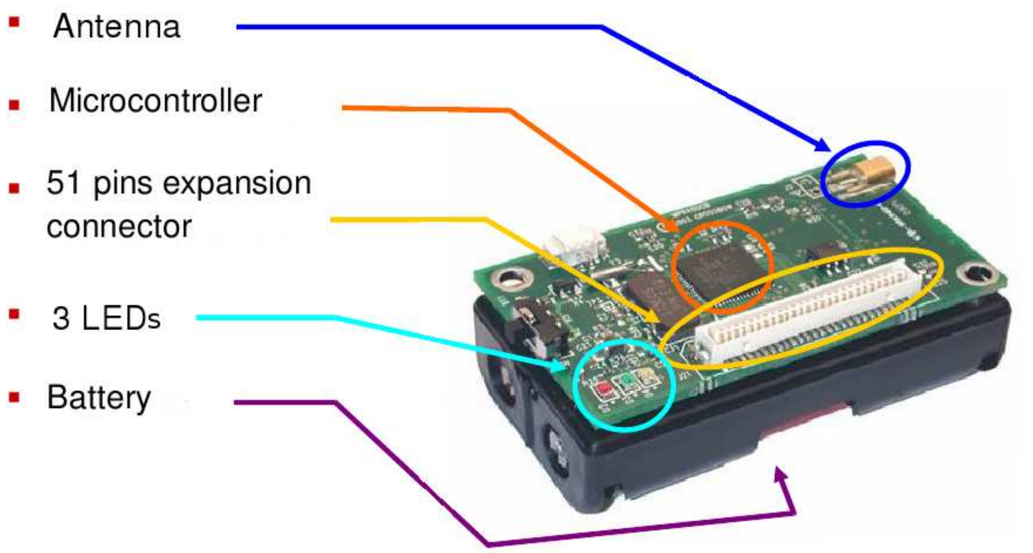


Figure 3.2: Components of a Tmote Sky mote

In the Figure 3.2, we can see the components of the mote: an antenna, to send and receive data; a microcontroller to perform tasks, process data and control the functionality of other components in the sensor node; three leds, useful to signal certain events or for debugging and an AA battery slot. A generic sensor has five subsystems (Figure 3.1), each one with a specific task:

- Sensing subsystem;
- Processing subsystem;
- Communication subsystem;
- Actuation subsystem;
- Power management subsystem.

The sensing subsystem is designed to get information about the environment used by other subsystems. The processing subsystem is designated to take data from the sensing subsystem and to elaborate them so making it possible they can be used by others. The communication subsystem sends and receives packets. The power management subsystem concerns all the operations about

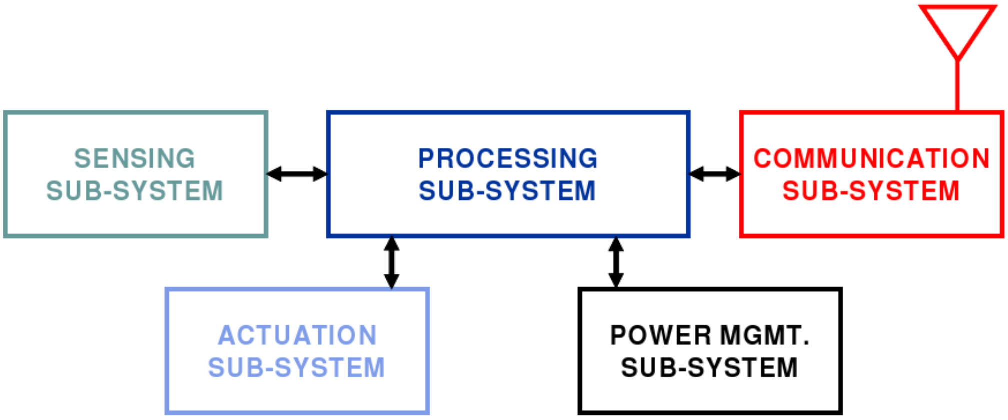


Figure 3.3: Functional blocks of a generic mote

battery managing (e.g. power saving). Finally, the actuation subsystem gets information from sensing and processing and decides how to control and make the system evolve.

The motes used developing this thesis are Tmote sky: it is an open source platform designed to enable cutting-edge experimentation for the research community. As [5] explains, the key Features of this suite are:

- IEEE 802.15.4/ZigBee compliant RF transceiver;;
- Interoperability with other IEEE 802.15.4 devices;
- 8MHz CC2420 microcontroller (10k RAM, 48k Flash);
- Integrated onboard antenna;
- Low current consumption;
- Programming and data collection via USB;
- Runs TinyOS 1.1.10 or higher.

3.2 CC2420 chipset

CC2420 is the chipset Tmote sky motes use. It provides hardware support for the cryptographic primitives and is used in several applications: Zigbee and TinyOS systems, home and building automation, industrial control and wireless sensor networks.

The CC2420 is a single-chip 2.4 GHz IEEE 802.15.4 compliant transceiver designed for low power and low voltage wireless applications. It provides extensive hardware support for packet handling, data buffering, burst transmissions, data encryption, data authentication, clear channel assessment, link quality indication and packet timing information. Between its many features, we can highlight the separate transmit and receive FIFOs, the IEEE 802.15.4 MAC hardware support (CRC_16 computation, Energy Detection, Link Quality detection, etc.) and IEEE 802.15.4 MAC hardware security (CTR encryption/decryption, CBC_MAC authentication CCM encryption/decryption and authentication, stand-alone AES encryption) [1].

3.2.1 Configuration and Data Interface

There are thirty-three 16-bit configuration and status registers, fifteen command strobe registers, and two 8-bit registers to access the separate transmit and receive FIFOs. Each data register is addressed by a 6-bit address. In each register read or write cycle, twenty-fours bits are read. Also the configuration registers can be read by the microcontroller.

CC2420 then returns the data from the addressed register in sixteen clock cycles. After the transfer, the CC2420 status byte is returned on the SO pin. The status byte contains 6 status bit whose configuration is described in Table 3.1. A SNOP (no operation) command strobe may be used to read the status byte.

Command strobes may be viewed as single byte instruction to CC2420. These commands must be used to enable receive mode, start decryption etc. All command strobes can be viewed in Table 3.2.

3.2.2 RAM access

CC240 also has 368 bytes RAM that can be accessed through the SPI interface. These registers contain a one-to -one mapping of the FIFO registers, the KEY0 and the KEY1 registers, the RXNONCE and the TXNONCE registers. This

Bit #	Name	Description
7		Reserved, ignore value
6	XOSC16M_STABLE	Indicates whether the 16 MHz oscillator is running or not 0 : The 16 MHz crystal oscillator is not running 1 : The 16 MHz crystal oscillator is running
5	TX_UNDERFLOW	Indicates whether a FIFO underflow has occurred during transmission. It must be cleared manually with a SFLUSHTX command strobe. 0 : No underflow has occurred 1 : An underflow has occurred
4	ENC_BUSY	Indicates whether the encryption module is busy 0 : Encryption module is idle 1 : Encryption module is busy
3	TX_ACTIVE	Indicates whether RF transmission is active 0 : RF Transmission is idle 1 : RF Transmission is active
2	LOCK	Indicates whether the frequency synthesizer PLL is in lock or not 0 : The PLL is out of lock 1 : The PLL is in lock
1	RSSI_VALID	Indicates whether the RSSI value is valid or not. 0 : The RSSI value is not valid 1 : The RSSI value is valid, always true when reception has been enabled at least 8 symbol periods
0		Reserved, ignore value

Table 3.1: Status byte flags

Address	**Register**	**Description**
0x00	SNOP	No Operation (has no other effect than reading out status-bits)
0x01	SXOSCON	Turn on the crystal oscillator (set XOSC16M_PD = 0 and BIAS_PD = 0)
0x02	STXCAL	Enable and calibrate frequency synthesizer for TX. Go from RX/TX to a wait state where only the synthesizer is running.
0x03	SRXON	Enable RX
0x04	STXON	Enable TX after calibration (if not already performed) Start TX in-line encryption if SPI_SEC_MODE = 0
0x05	STXONCCA	If CCA indicates a clear channel:Enable calibration, then TX. Start in-line encryption if SPI_SEC_MODE = 0, else do nothing.
0x06	SRFOFF	Disable RX/TX and frequency synthesizer
0x07	SXOSCOFF	Turn off the crystal oscillator and RF
0x08	SFLUSHRX	Flush the RX FIFO buffer and reset the demodulator. Always read at least one byte from the RXFIFO before issuing the SFLUSHRX command strobe
0x09	SFLUSHRX	Flush the TX FIFO buffer
0x0A	SACK	Send acknowledge frame, with pending field cleared
0x0B	SACKPEND	Send acknowledge frame, with pending field set
0x0C	SRXDEC	Start RXFIFO in-line decryption/authentication (as set by SPI_SEC_MODE)
0x0D	STXENC	Start TXFIFO in-line encryption/authentication (as set by SPI_SEC_MODE), without starting TX
0x0E	SAES	AES Stand alone encryption strobe. SPI_SEC_MODE is not required to be 0, but the encryption module must be idle. If not, the strobe is ignored

Table 3.2: Strobe configuration registers overview

mapping is very useful to make debugging, in fact the TXFIFO is write only, but it may be read back using RAM access.

Data are read and written one byte at a time, as with RAM access. The RXFIFO is both writeable and readable. KEY0 and KEY1 registers contain a 16-bit key used for ciphering/deciphering operation. After a key is written in any of these registers, it is selected and then used when reading the SEC_TXKEYSEL/SEC_RXKEYSEL bit in SECCTRL0 register. TXNONCE and RXNONCE contain nonce.

3.2.3 Security operations

CC2420 features hardware IEEE 802.15.4 MAC security operations. This includes counter mode (CTR) encryption/decryption, CBC_MAC authentication and CCM encryption and authentication. All security operations are based on AES encryption using 128 bit keys and they are performed within the transmit and receive FIFOs on a per-frame basis.

The SAES, STXENC and SRXDEC command strobes are used to start security operations in CC2420 as will be described in the following sections. The ENC_BUSY status bit may be used to monitor when a security operation has been completed. Security command strobes issued while the security engine is busy, will be ignored and the ongoing operation will be completed. The CC2420 RAM space has storage space for two individual keys (KEY0 and KEY1).

Transmit, receive and stand-alone encryption may select one of these two keys relying on the three control bits SEC_TXKEYSEL, SEC_RXKEYSEL and SEC_SAKEYSEL in the SECCTRL0 register. A way of establishing the keys used for encryption and authentication must be decided considering the particular application requirements. IEEE 802.15.4, in fact, does not define how this is done, it is left to the higher layers of the protocol. However, the nonce must be correctly initialized before starting any reception or transmission [7].

The in-line security mode is set in SECCTRL0.SEC_MODE to one of the following modes:

- NO_SEC (disabled);
- CBC_MAC (authentication);

- CTR (encryption/decryption);
- CCM (authentication and encryption/decryption).

When enabled, transmission (TX) in-line security is started in two different ways: the first one is issuing the STXENC command strobe, so in-line security will be performed within the TXFIFO buffer, but a transmission will not be started. The second one is issuing the STXON or STXONCCA command strobe, so in-line security will be performed within the TXFIFO and a transmission of the ciphertext is started.

When enabled, reception (RX) in-line security is started issuing a SRXDEC command strobe, so the first frame in the RXFIFO buffer is decrypted/authenticated as set by the current security mode. RX in-line security operations are always performed on the first frame currently inside the RXFIFO, even if parts of this has already been read out over the Serial Peripheral Interface (SPI). This allows the receiver to first read the source address out, making it possible to decide which key to use before doing authentication of the complete frame. In CTR or CCM mode it is of course important that bytes to be decrypted are not read out before the security operation is started.

3.2.3.1 CTR mode encryption/decryption

CTR mode encryption/decryption is performed by CC2420 on MAC frames within the TXFIFO/RXFIFO respectively. SECCTRL1.SEC_TXL/SEC_RXL flags in this control register set the number of bytes between the length field and the first byte to be encrypted/decrypted respectively, so controlling the number of plaintext bytes in the current frame.

When encryption is initiated, the plaintext in the TXFIFO is then encrypted. The encryption module will encrypt all the plaintext currently available or it will wait if not everything is prebuffered. The encryption operation may also be started without any data in the TXFIFO at all, and data will be encrypted as soon as they are written to the TXFIFO. When decryption is initiated issuing the SRXDEC command strobe, the ciphertext of the RXFIFO is then decrypted.

3.2.3.2 CBC_MAC

CBC_MAC in-line authentication is provided by CC2420 hardware. When enabling CBC_MAC in-line TXFIFO authentication, the generated MIC is written to the TXFIFO for transmission. The frame length must include the MIC. SECCTRL1.SEC_TXL/SEC_RXL flags in this control register set the number of bytes between the length field and the first byte to be authenticated. Normally it is set to 0 for MAC authentication. SECCTRL0.SEC_M flag in this control register set the MIC length M, encoded as $(M - 2)/2$. SECCTRL0.SEC_CBC_HEAD flag defines if the authentication length is used as the first byte of data to be authenticated or not. This bit should be set to one.

When enabling CBC-MAC in-line RXFIFO authentication, the generated MIC is compared to the MIC in the RXFIFO. The last byte of the MIC is replaced in the RXFIFO with 0x00 if MIC is correct or 0xFF if MIC is incorrect.

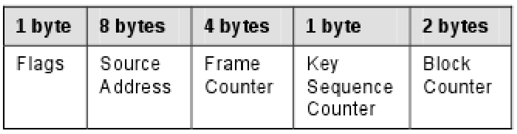
3.2.3.3 CCM

CCM combines CTR mode encryption and CBC_MAC authentication in a single operation. SECCTRL0.SEC_M flag sets the MIC length M, encoded as $(M - 2)/2$. SECCTRL0.SEC_CBC_HEAD flag defines if the authentication length is used as the first byte of data to be authenticated or not. This bit should be set to one. SECCTRL1.SEC_TXL/SEC_RXL sets the number of bytes after the length field to be authenticated but not encrypted. The MIC is generated and verified in the same way descibed in the CBC-MAC subsection.

3.2.3.4 Nonce structure

The receive and transmit nonces used for encryption and decryption are located in RAM, starting from addresses 0x110 and 0x140 respectively. They are both sixteen bytes. The nonce must be correctly initialized before receiving or transmitting secured frames. The format of the nonce is shown in Figure 3.4.

The standard imposes the block counter to be set to one, the key sequence counter is controlled by a layer above the MAC layer. The frame counter must be increased at each new frame by the MAC layer. The source address is the 64-bit IEEE address.



1 byte	8 bytes	4 bytes	1 byte	2 bytes
Flags	Source Address	Frame Counter	Key Sequence Counter	Block Counter

Figure 3.4: IEEE 802.15.4 Nonce

CC2420 gives the user full flexibility in selecting the flags for nonces according to the chosen security level. The flag setting is stored in the most significant byte of the nonce. The flag byte used for encryption and authentication is then generated as shown in Figure 3.5.

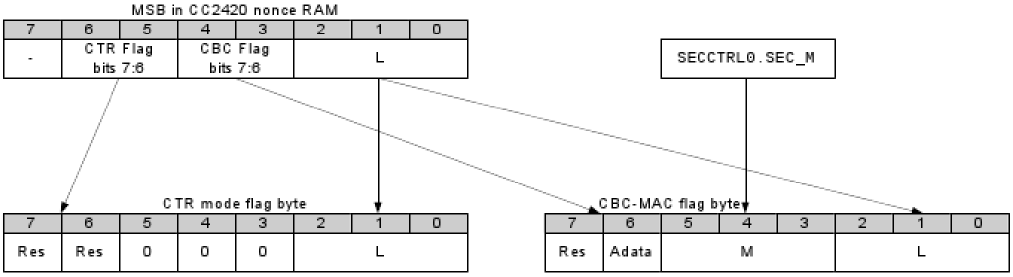


Figure 3.5: CC2420 Security Flag Byte

Chapter 4

Security implementation

4.1 Overview

IEEE 802.15.4 offers several ways to secure a frame: packets can be only encrypted, only authenticated or both encrypted and authenticated. In our scenario, we consider some Reduced Function Devices (RFDs) transmitting data with different security levels to the Full Function Device (FFD) coordinator, that decrypts messages and sends back acks. When security is turned off, upper layers does not send to MAC any security parameters: when the frame is built, security routines are not called and the frame is sent in clear.

At the startup of the application, both RFDs and FFD set their security data structures, such as Key Table and Device Table, according to the security parameters they are going to use. Then, while parsing the frames, the coordinator understands whether it has security or not and, eventually, recognizes the security level, behaving appropriately

When security is active, the first step consists in verifying consistency between table contents and security parameters. If these checks succeed, the Auxiliary Security Header is built and inserted into the MAC frame. Finally, the proper security routines are called before sending the frame.

The coordinator, while parsing the frame, understands that it is secured and then proceeds to unsecure it in a coherent manner. It makes the consistency checks too, so verifying the correctness of its table contents. If everything works, it recognizes the security level and unsecures the packet relying on the opportune security routines [10].

The security parameters mentioned before are overviewed by Table 4.1 and will be deeply explained later.

Name	**Type**	**Range**	**Description**
SecurityLevel	Integer	0x00-0x07	The security level to be used
KeyIdMode	Integer	0x00-0x03	The mode used in order to identify the key to be used
Key Source	Set of 0 4, or 8 octets	As specified by the KeyIdMode parameter	The originator of the key to be used. It is ignored if the KeyIdMode parameter is ignored or set to 0x00.
KeyIndex	Integer	0x01-0xff	The index of the key to be used. It is ignored if the KeyIdMode parameter is ignored or set to 0x00

Table 4.1: Security Parameters

This security structure is used by application layer to manage the security parameters when the frame is created. The implementation of the security structure is[1]:

```
typedef struct ieee154_security {
  uint8_t SecurityLevel;
  uint8_t KeyIdMode;
  uint8_t KeySource[8];
  uint8_t KeyIndex;
}ieee154_security_t;
```

The following section deeply describes the security data structures implementation and their functionalities.

4.2 Security data structures

The PIB security-related attributes are presented in Table 4.2. Among them, the MacKeyTable and the MacDeviceTable are the most important security

[1] All the code is Copyright (c) 2008, Technische Universitaet Berlin All rights reserved

data structure. Mainly, they are a set of KeyDescriptor and DeviceDescriptor respectively.

Attribute	Identifier	Type	Range	Description	Default
macKeyTable	0x71	List of KeyDescriptor entries	-	A table of KeyDescriptors, each containing keys and related-security information	(empty)
macKeyTableEntries	0x72	Integer	Implementation specific	The number of entries in macKeyTable	0
macDeviceTable	0x73	List of DeviceDescriptor entries	-	A table of DeviceDescriptor entries, each indicating a remote device with which this one securely communicates	(empty)
macDeviceTableEntries	0x74	Integer	Implementation specific	The number of entries in macDeviceTable	0
mac SecurityLevelTable	0x75	Table of SecurityLevel Descriptor entries	-	A table of SecurityLevelDescriptors, holding information about the minimum security level expected	(empty)

continued on next page

continued from previous page

Attribute	Identifier	Type	Range	Description	Default
macSecurityLevelTableEntries	0x76	Integer	Implementation specific	The number of entries in macSecurityLevel Table	0
macFrameCounter	0x77	Integer	0x00000000-0xffffffff	The outgoing frame counter	0x00
macAuto Request SecurityLevel	0x78	Integer	0x00-0x07	The security level used for automatic data request	0x06
macAuto RequestKeyIdMode	0x79	Integer	0x00-0x03	KeyIdMode used for automatic data request	0x00
macAutoRequestKeySource	0x7a	As specified by the macAutoRequestKeyIdMode	-	The originator of the key used for automatic data request	All 0xff
macAutoRequestKeyIndex	0x7b	Integer	0x01-0xff	The index of the key used for automatic data request	All 0xff
macDefaultKeySource	0x7c	Set of 8 octets	-	The originator of the default key used for KeyIdMode1	All 0xff
macPANCoordExtended-	0x7d	IEEE address	An extended 64-bit IEEE address	64-bit address of the PAN coordinator	-

continued on next page

continued from previous page

Attribute	**Identifier**	**Type**	**Range**	**Description**	**Default**
Address					
MacPAN-Coord-Short-Address	0x73	Integer	0x0000-0xffff	16-bit address of the PAN coordinator	0x0000

Table 4.2: MAC security-related PIB

4.2.1 KeyTable

The key table holds KeyDescriptors, particular data structures able to provide key-specific information. These are retrieved thanks to some parameters explicitly contained in the requesting primitive or in the received frame, and are involved in the outgoing frame key retrieval procedure and the incoming frame security material retrieval procedure, as well as the KeyDescriptor lookup procedure. All these procedures will be described starting from Section 4.4. The implementation of this data structure is the following:

```
typedef struct ieee154_macKeyTable_t{
  ieee154_KeyDescriptor_t keydescriptor
[MAX_MAC_KEY_TABLE_ENTRIES];
  bool valid[MAX_MAC_KEY_TABLE_ENTRIES];
}ieee154_macKeyTable_t;
```

4.2.1.1 KeyDescriptor structure

The following rapresents the ieee154_KeyDescriptor_t nesC implementation:

```
typedef struct ieee154_KeyDescriptor_t{
  ieee154_LookupDescriptor_t
          keyidlookupdescriptor [MAX_MAC_KEY_ID_LOOKUP_LIST_ENTRIES];
  uint8_t keyidlookupentries;
  ieee154_KeyDeviceDescriptor_t
        keydevicelis [MAX_MAC_DEVICE_LIST_ENTRIES];
  uint8_t keydevicelistentries;
  ieee154_KeyUsageDescriptor_t
        keyusagelist [MAX_MAC_KEY_USAGE_LIST_ENTRIES];
  uint8_t keyusagelistentries;
  ieee154_Key_t key[16];
}ieee154_KeyDescriptor_t;
```

4.2.2 DeviceTable

The device table holds DeviceDescriptors, containing device-specific addressing information that, when combined with key-specific information from the key table, provide all the keying material needed to secure outgoing and unsecure incoming frames. Device-specific information in the device table is identified based on the originator of the frame, as described in the blacklist checking procedure we are going to describe in Section 4.5.3.1. The implementation of this data structure is:

```
typedef struct ieee154_macDeviceTable_t{
  ieee154_DeviceDescriptor_t devicedescriptor [MAX_SEC_TABLE_ENTRIES];
  bool valid [MAX_SEC_TABLE_ENTRIES];
}ieee154_macDeviceTable_t;
```

4.2.2.1 DeviceDescriptor structure

The following rapresents the ieee154_DeviceDescriptor_t nesC implementation:

```
typedef struct ieee154_DeviceDescriptor_t{
  ieee154_macPANId_t panid;
  ieee154_address_t address;
  ieee154_macFrameCounter_t framecounter;
  bool exempt;
}ieee154_DeviceDescriptor_t;
```

4.2.3 Minimum security level table

The minimum security level table holds information regarding the minimum security level the device expects to have been applied by the originator of a frame, depending on frame type and, if it concerns a MAC command frame, the command frame identifier. Security processing of an incoming frame will fail if the frame is not adequately protected.

4.2.4 Frame counter

The 4-octet frame counter is used to provide replay protection and semantic security of the cryptographic building block used for securing outgoing frames. The frame counter is included in each secured frame and is one of the elements required for the unsecuring operation at the recipient(s).

The frame counter is incremented each time an outgoing frame is secured, as described in the outgoing frame security procedure (Section 4.4). When the frame counter reaches its maximum value of 0xffffffff, the associated keying material is blacklisted, requiring all keys associated with the device to be updated. This provides a mechanism for ensuring that the keying material for every frame is unique and, thereby, provides for sequential freshness.

4.2.5 Automatic request attributes

Automatic request attributes hold all the information needed to secure outgoing frames generated automatically and not as a result of a higher layer primitive, as is the case with automatic data requests.

4.2.6 Default key source

The default key source is an information commonly shared between originator and receipient(s) of a secured frame, which, when combined with additional information explicitly contained in the requesting primitive or in the received frame, allows an originator or a recipient to determine the key required for securing or unsecuring this frame, respectively.

This provides a mechanism for significantly reducing the overhead of security information contained in secured frames in particular use cases.

4.2.7 PAN coordinator address

The address of the PAN coordinator is an information commonly shared between all devices in a PAN, which, when combined with additional information explicitly contained in the requesting primitive or in the received frame, allows an originator of a frame directed to the PAN coordinator or a recipient of a frame originating from the PAN coordinator to determine the key and security-related information required for securing or unsecuring this frame.

4.3 Security functional description

Security implementation is optional on a device. When a device does not implement security, it shall not provide a mechanism for the MAC sublayer to perform any cryptographic transformation on incoming and outgoing frames nor require any PIB attributes associated with security.

A device that implements security, on the other hand, shall provide a mechanism for the MAC sublayer to provide cryptographic transformations on incoming and outgoing frames using information in the PIB attributes associated with security when the macSecurityEnabled attribute is set to TRUE.

If the MAC sublayer is required to transmit a frame or receives an incoming frame, the MAC sublayer shall process the frame as descibed in Figure 4.1 and Figure 4.4, respectively.

4.4 Outgoing frame security procedure

The inputs to this procedure are:

- the frame to be secured;
- SecurityLevel;
- KeyIdMode;
- KeySource;
- KeyIndex.

The outputs from this procedure are the status of the procedure and, if it is SUCCESS, the secured frame.

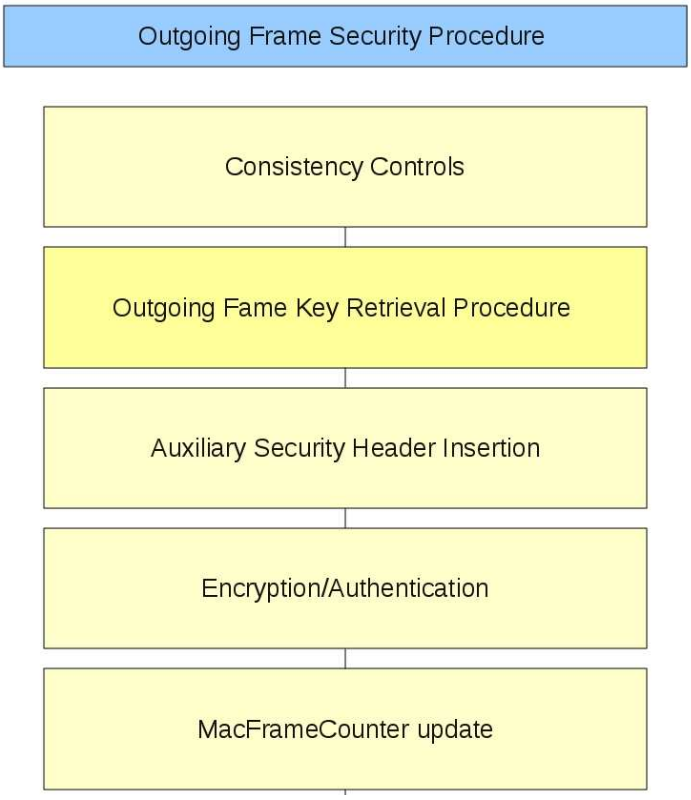

Outgoing Frame Security Procedure
Consistency Controls
Outgoing Fame Key Retrieval Procedure
Auxiliary Security Header Insertion
Encryption/Authentication
MacFrameCounter update

Figure 4.1: Outgoing frame security procedure schema

Fist of all, the procedure does some consistency controls and, if everything works, it shall obtain the key using the outgoing frame key retrieval procedure as described in Figure 4.3. In case of failure, the procedure shall return with a status of UNAVAILABLE_KEY. Otherwise, it proceeds with the auxiliary security header insertion and the encryption/authentication. Finally, the procedure updates the frame counter to the macFrameCounter attribute. If the frame counter has the value 0xffffffff, the procedure shall return with a status of COUNTER_ERROR.

4.4.1 Outgoing frame consistency checks

Figure 4.2 describes the consistency checks we are going to enumerate.

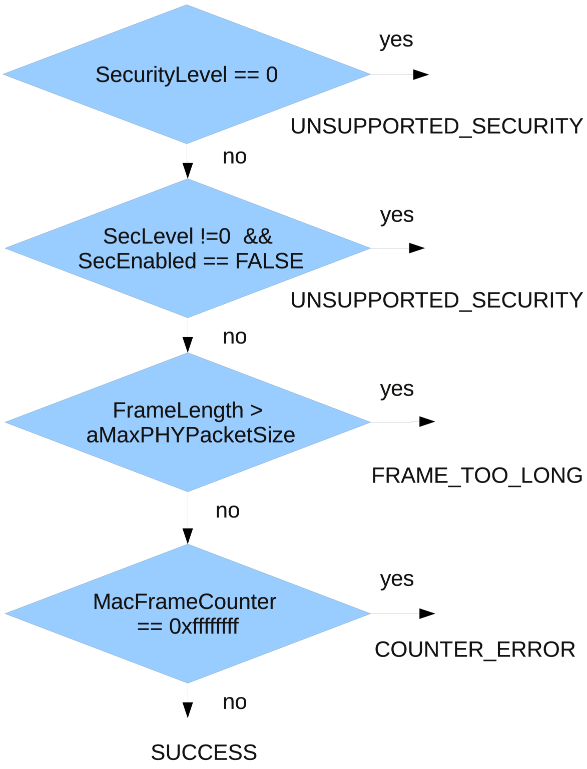


Figure 4.2: Outgoing frame consistency checks

If the Security Enabled subfield of the Frame Control field of the frame to be secured is set to zero, the procedure shall set the security level to zero.

If the Security Enabled subfield of the Frame Control field of the frame to be secured is set to one, the procedure shall set the security level to the SecurityLevel parameter. If the resulting security level is zero, the procedure shall return with a status of UNSUPPORTED_SECURITY.

If the macSecurityEnabled attribute is set to FALSE and the security level is not equal to zero, the procedure shall return with a status of UNSUPPORTED_SECURITY.

Then, the procedure shall determine whether the frame to be secured fits the maximum length of MAC frames payload: the procedure shall set the size (M) of the Authentication field to zero if the security level is equal to zero and shall determine this value from the security level otherwise. The procedure shall determine the AuxLen, that is the length, in octets, of the auxiliary security header using the KeyIdMode and the security level. If this check fails, the procedure shall return with a status of FRAME_TOO_LONG.

4.4.2 Outgoing frame key retrieval procedure

The inputs to this procedure are:

- the frame to be secured;
- SecurityLevel;
- KeyIdMode;
- KeySource;
- KeyIndex.

The outputs from this procedure are a passed or failed status and, if passed, a key.

If the KeyIdMode parameter is set to KeyIdMode0 (i.e. implicit key identification), the outgoing frame key retrieval procedure shall determine the key

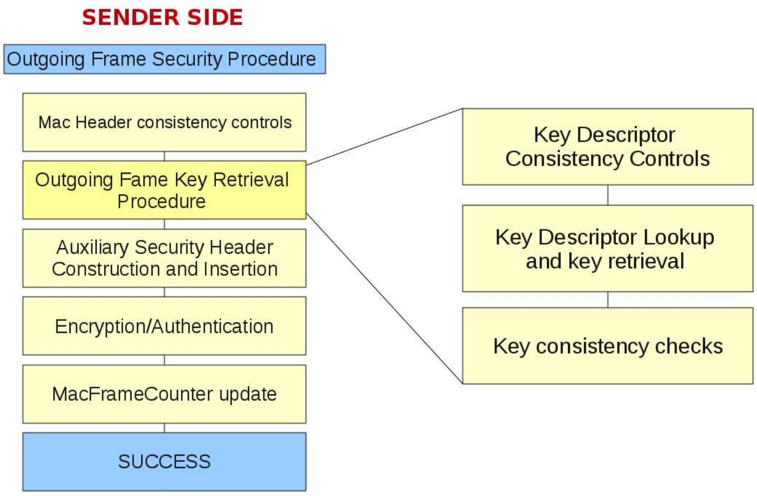

SENDER SIDE
Outgoing Frame Security Procedure
Mac Header consistency controls
Outgoing Fame Key Retrieval Procedure
Auxiliary Security Header Construction and Insertion
Encryption/Authentication
MacFrameCounter update
SUCCESS
Key Descriptor Consistency Controls
Key Descriptor Lookup and key retrieval
Key consistency checks

Figure 4.3: Key retrieval procedure schema

lookup data and key lookup size as follows:

- if the short addressing mode is used, the key lookup data shall be set to the 2-octets SourcePANIdentifier field of the frame right-concatenated with the 2-octets macPANCoordShortAddress attribute, in turn right-concatenated with the single octet 0x00. The key lookup size shall be set to five;
- if the extended addressing mode is used, the key lookup data shall be set to the 8-octet macPANCoordExtendedAddress attribute right-concatenated with the single octet 0x00. The key lookup size shall be set to nine;
- if the Destination Addressing Mode subfield of the Frame Control field of the frame is set to 0x02, the key lookup data shall be set to the 2-octet Destination PAN Identifier field of the frame right-concatenated with the 2-octet Destination Address field of the frame and with the single octet 0x00. The key lookup size shall be set to five;

- if the Destination Addressing Mode subfield of the Frame Control field of the frame is set to 0x03, the key lookup data shall be set to the 8-octet Destination Address field of the frame right-concatenated with the single octet 0x00. The key lookup size shall be set to nine.

If the KeyIdMode parameter is set to a value not equal to KeyIdMode0, the procedure shall determine the key lookup data and key lookup size as follows:

- if the KeyIdMode parameter is set to KeyIdMode1 (i.e. default key source address), the key lookup data shall be set to the value of the 8-octet macDefaultKeySource attribute right-concatenated with the single octet KeyIndex parameter. The key lookup size shall be set to nine;
- if the KeyIdMode parameter is set to KeyIdMode2 (i.e. short address key source), the key lookup data shall be set to the 4-octet KeySource parameter right-concatenated with the single octet KeyIndex parameter. The key lookup size shall be set to five;
- if the KeyIdMode parameter is set to KeyIdMode3 (i.e. extended address key source), the key lookup data shall be set to the 8-octet KeySource parameter right-concatenated with the single octet KeyIndex parameter. The key lookup size shall be set to nine.

The procedure shall obtain the KeyDescriptor by passing the key lookup data and the key lookup size to the KeyDescriptor lookup procedure.

If that procedure returns with a failed status, this procedure shall also return with a failed status. The MAC sublayer shall set the key to the Key element of the KeyDescriptor. The procedure shall return with a passed status, having obtained the key identifier and the key as well.

4.4.2.1 Key descriptor lookup procedure

The inputs to this procedure are the key lookup data and the key lookup size. The outputs from this procedure are a passed or failed status and, if passed, a

KeyDescriptor.

The procedure, for each KeyDescriptor in the macKeyTable attribute and for each KeyIdLookupDescriptor in the KeyIdLookupList of the KeyDescriptor, shall check whether the LookupDataSize element of the KeyIdLookupDescriptor indicates the same integer value as the key lookup size and whether the LookupData element of the KeyIdLookupDescriptor is equal to the key lookup data. If both checks pass, the procedure shall return with this (matching) KeyDescriptor and a passed status.

Otherwise, the procedure shall return with a failed status.

4.5 Incoming frame security procedure

The input to this procedure is the frame to be unsecured. The outputs from this procedure are:

- the unsecured frame;
- SecurityLevel;
- KeyIdMode;
- KeySource;
- KeyIndex;
- the status of the procedure.

All outputs of this procedure are assumed to be invalid unless and until explicitly set in this procedure.

It is assumed the PIB attributes associating KeyDescriptors in macKeyTable with a single, unique device or a number of devices will have been established by the next higher layer.

Fist of all, the procedure does some consistency controls and, if nothing goes wrong, it moves on doing the incoming security level checking procedure. If that fails, the procedure shall set the unsecured frame to be the frame to be unsecured and return with the unsecured frame, the security level, the key identifier mode, the key source, the key index, and a status of IMPROPER_SECURITY_LEVEL.

Otherwise, it proceeds with the incoming frame security material retrieval procedure, described in Figure 4.6. It the procedure succedees, it passess the KeyDescriptor, the frame type, and, depending on whether the frame is a MAC command frame, the first octet of the MAC payload (i.e., command frame identifier for a MAC command frame) to the incoming key usage policy checking procedure.

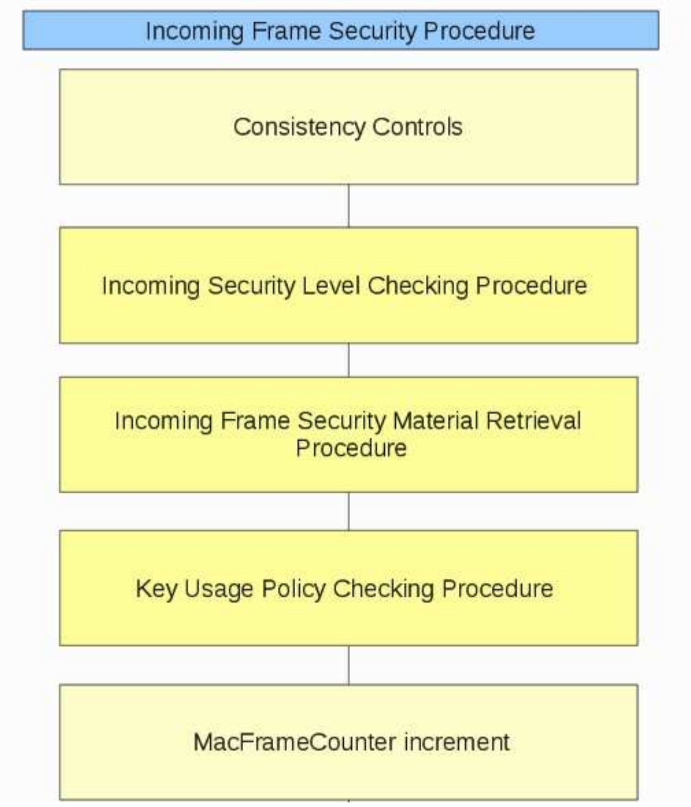

Incoming Frame Security Procedure
Consistency Controls
Incoming Security Level Checking Procedure
Incoming Frame Security Material Retrieval Procedure
Key Usage Policy Checking Procedure
MacFrameCounter increment

Figure 4.4: Incoming frame security procedure schema

If that procedure fails, the procedure shall set the unsecured frame to be the frame to be unsecured and return with the unsecured frame, the security level, the key identifier mode, the key source, the key index, and a status of IMPROPER_KE_TYPE. It follows another checking phase, described in Figure 4.5.

If everything works, the procedure shall then use the ExtAddress element of the DeviceDescriptor, the frame counter, the security level, and the Key element of the KeyDescriptor to produce the unsecured frame according to the CCM inverse transformation process.

If the security level specifies the use of encryption, the decryption operation shall be applied only to the actual payload field within the MAC payload. If the CCM inverse transformation process fails, the procedure shall return with a status of SECURITY_ERROR.

The procedure shall increment the frame counter and set the FrameCounter element of the DeviceDescriptor to the resulting value. If the FrameCounter element is equal to 0xffffffff, the procedure shall set the Blacklisted element of the KeyDeviceDescriptor. The procedure shall return with the unsecured frame, the security level, the key identifier mode, the key source, the key index, and a status of SUCCESS.

4.5.1 Incoming frame consistency checks

Figure 4.5 describes the consistency checks we are going to enumerate. If the Security Enabled subfield of the Frame Control field of the frame to be unsecured is set to zero, the procedure shall set the security level to zero.

If the Security Enabled subfield of the Frame Control field of the frame to be unsecured is set to one and the Frame Version subfield of the Frame Control field of the frame to be unsecured is set to zero, the procedure shall return with a status of UNSUPPORTED_LEGACY.

If the Security Enabled subfield of the Frame Control field of the frame

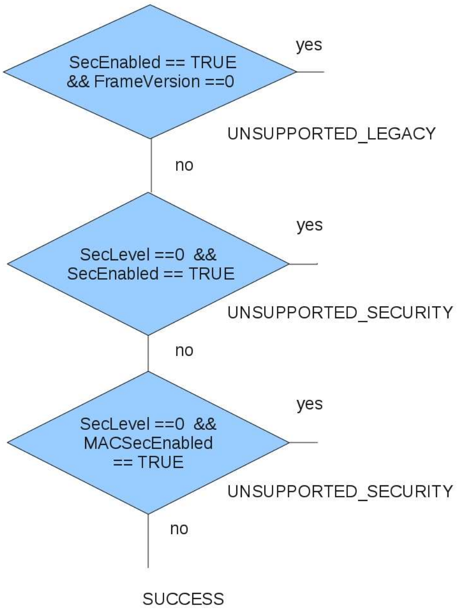

SecEnabled == TRUE && FrameVersion ==0
yes
UNSUPPORTED_LEGACY
no
SecLevel ==0 && SecEnabled == TRUE
yes
UNSUPPORTED_SECURITY
no
SecLevel ==0 && MACSecEnabled == TRUE
yes
UNSUPPORTED_SECURITY
no
SUCCESS

Figure 4.5: Incoming frame consistency checks

to be unsecured is set to one, the procedure shall set the security level and the key identifier mode to the corresponding subfields of the Security Control field of the auxiliary security header of the frame to be unsecured and shall set the key source and key index to the corresponding subfields of the Key Identifier field of the auxiliary security header of the frame to be unsecured, if present. If the resulting security level is zero, the procedure shall set the unsecured frame to be the frame to be unsecured and return a status of UNSUPPORTED_SECURITY.

If the macSecurityEnabled attribute is set to FALSE, the procedure shall set the unsecured frame to be the frame to be unsecured and return with the unsecured frame, the security level, the key identifier mode, the key source, the key index, and a status of SUCCESS if the security level is equal to zero and with a status of UNSUPPORTED_SECURITY otherwise.

4.5.2 Incoming security level checking procedure

The procedure shall determine whether the frame to be unsecured meets the minimum security level by passing the security level, the frame type, and, depending on whether the frame is a MAC command frame, the first octet of the MAC payload (i.e. command frame identifier for a MAC command frame) to this procedure.

The inputs to this procedure are the incoming security level, the frame type and the command frame identifier. The output from this procedure is a passed, failed, or conditionally passed status.

The incoming security level checking for each SecurityLevelDescriptor in the macSecurityLevelTable attribute:

- If the frame type is not equal to MAC command frame (0x03) and the frame type is equal to the FrameType element of the SecurityLevelDescriptor, the procedure shall compare the incoming security level with the SecurityMinimum element of the SecurityLevelDescriptor. If this comparison fails, the procedure shall return with a conditionally passed status if the DeviceOverrideSecurityMinimum element of the SecurityLevelDescriptor is set to TRUE and the security level is set to zero and with a failed status otherwise;

- if the frame type is equal to MAC command frame (0x03), the frame type is equal to the FrameType element of the SecurityLevelDescriptor, and the command frame identifier is equal to the CommandFrameIdentifier element of the SecurityLevelDescriptor, the procedure shall compare the incoming security level with the SecurityMinimum element of the SecurityLevelDescriptor. If this comparison fails, the procedure shall return with a conditionally passed status if the DeviceOverrideSecurityMinimum element of the SecurityLevelDescriptor is set to TRUE and the security level is set to zero and with a failed status otherwise;

- If everything goes right, the procedure shall return with a passed status.

4.5.3 Incoming frame security material retrieval procedure

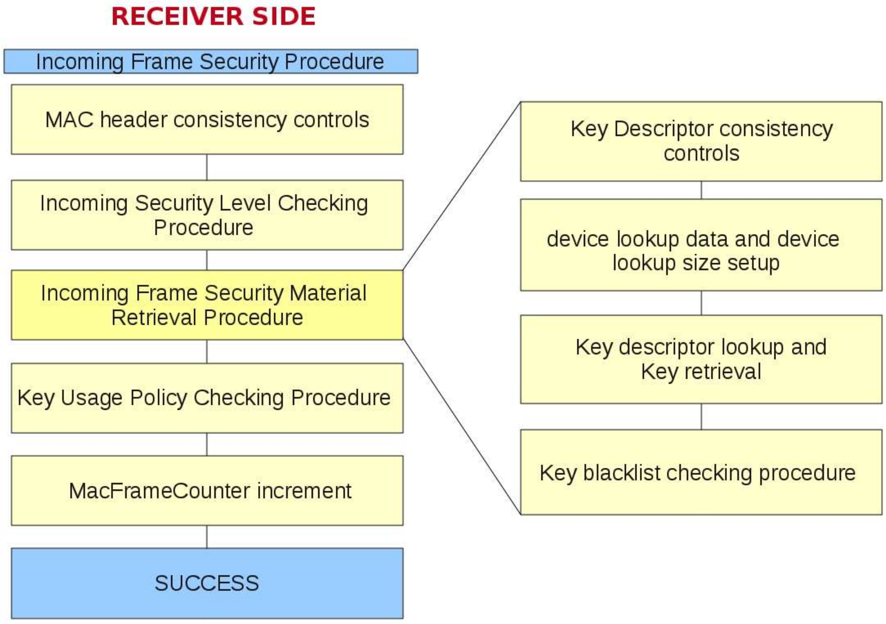

RECEIVER SIDE
Incoming Frame Security Procedure
MAC header consistency controls
Incoming Security Level Checking Procedure
Incoming Frame Security Material Retrieval Procedure
Key Usage Policy Checking Procedure
MacFrameCounter increment
SUCCESS
Key Descriptor consistency controls
device lookup data and device lookup size setup
Key descriptor lookup and Key retrieval
Key blacklist checking procedure

Figure 4.6: Incoming frame security material retrieval procedure schema

The input to this procedure is the frame to be unsecured. The outputs from this procedure are a passed or failed status and, if passed, a KeyDescriptor, a DeviceDescriptor, and a KeyDeviceDescriptor.

As Figure 4.6 describes, the procedure checks if the Key Identifier Mode subfield of the Security Control field of the auxiliary security header of the frame is set to KeyIdMode0. If so, the procedure shall determine the key lookup data and the key lookup size as follows:

- if the source address mode of the FrameControl field is set to 0x00 and the macPANCoordShortAddress attribute is set to a value in the short addressing mode is used, the key lookup data shall be set to the 2-octet Destination PAN Identifier field of the frame right-concatenated with the 2-octet macPANCoordShortAddress attribute right-concatenated with the single octet 0x00. The key lookup size shall be set to five;

- if the source address mode of the Frame Control field of the frame is set to 0x00 and the extended addressing mode is used, the key lookup data shall be set to the 8-octet macPANCoordExtendedAddress attribute right-concatenated with the single octet 0x00. The key lookup size shall be set to nine;
- if the source address mode of the Frame Control field of the frame is set to 0x02, the key lookup data shall be set to the 2-octet Source PAN Identifier field of the frame, or to the 2-octet Destination PAN Identifier field of the frame if the PAN ID Compression subfield of the Frame Control field of the frame is set to one, right-concatenated with the 2-octet SourceAddress field of the frame right-concatenated with the single octet 0x00. The key lookup size shall be set to five;
- if the source address mode of the Frame Control field of the frame is set to 0x03, the key lookup data shall be set to the 8-octet Source Address field of the frame right-concatenated with the single octet 0x00. The key lookup size shall be set to nine.

If the Key Identifier Mode subfield of the Security Control field of the auxiliary security header of the frame is set to a value not equal to KeyIdMode0, the procedure shall determine the key lookup data and key lookup size as follows:

- if the KeyIdMode1 is used, the key lookup data shall be set to the 8-octet macDefaultKeySource attribute right-concatenated with the 1-octet Key Index subfield of the Key Identifier field of the auxiliary security header. The key lookup size shall be set to nine;
- if the KeyIdMode2 is used, the key lookup data shall be set to the right-

concatenation of the 4-octet Key Source subfield and the 1-octet Key Index subfield of the Key Identifier field of the auxiliary security header. The key lookup size shall be set to five;

- if the KeyIdMode2 is used, the key lookup data shall be set to the right-concatenation of the 4-octet Key Source subfield and the 1-octet Key Index subfield of the Key Identifier field of the auxiliary security header. The key lookup size shall be set to five;

- if the KeyIdMode3 is used, the key lookup data shall be set to the right-concatenation of the 8-octet Key Source subfield and the 1-octet Key Index subfield of the Key Identifier field of the auxiliary security header. The key lookup size shall be set to nine.

The procedure shall obtain the KeyDescriptor by passing the key lookup data and the key lookup size to the KeyDescriptor lookup procedure. If that procedure returns with a failed status, the procedure shall also return with a failed status.

The procedure shall determine the device lookup data and the device lookup size as follows:

- if the source address mode of the Frame Control field of the frame is set to 0x00 (i.e. PAN identifier and address fields are not present) and are used the short addresses, the device lookup data shall be set to the 2-octet Destination PAN Identifier field of the frame right-concatenated with the 2-octet macPANCoordShortAddress attribute. The device lookup size shall be set to four;

- if the source address mode of the Frame Control field of the frame is set to 0x00 and are used the extended addresses, the device lookup data shall be set to the 8-octet macPANCoordExtendedAddress attribute. The device lookup size shall be set to eight;

- if the source address mode of the Frame Control field of the frame is set to 0x02 (i.e. Address field contains a 16-bit short address), the device lookup data shall be set to the 2-octet Source PAN Identifier field of the frame, or to the 2-octet Destination PAN Identifier field of the frame (if the PAN ID Compression subfield of the Frame Control field of the frame is set to one), right-concatenated with the 2-octet Source Address field of the frame. The device lookup size shall be set to four;

- if the source address mode of the Frame Control field of the frame is set to 0x03 (i.e. Address field contains a 64-bit extended address), the device lookup data shall be set to the 8-octet Source Address field of the frame. The device lookup size shall be set to eight.

The procedure shall obtain the DeviceDescriptor and the KeyDeviceDescriptor by passing the KeyDescriptor, the device lookup data, and the device lookup size to the blacklist checking procedure, described in Section 4.5.3.1.

If that procedure returns with a failed status, this procedure shall also return with a failed status too. Otherwise, the procedure shall return with a passed status having obtained the KeyDescriptor, the DeviceDescriptor, and the KeyDeviceDescriptor.

4.5.3.1 Blacklist checking procedure

The blacklist checking procedure for each KeyDeviceDescriptor in the KeyDeviceList of the KeyDescriptor:

- obtains the DeviceDescriptor using the DeviceDescriptorHandle element of the KeyDeviceDescriptor;

- if the UniqueDevice element of the KeyDeviceDescriptor is set to TRUE, the procedure shall return with the DeviceDescriptor and the KeyDeviceDescriptor. It returns a passed status if the BlackListed element of the KeyDeviceDescriptor is set to FALSE, or the procedure shall return with a failed status if this Blacklisted element is set to TRUE;

- if the UniqueDevice element of the KeyDeviceDescriptor is set to FALSE, the procedure shall execute the DeviceDescriptor lookup procedure with the device lookup data and the device lookup size as inputs. If the corresponding output of that procedure is a passed status, the procedure shall return with the DeviceDescriptor, the KeyDevice-Descriptor, and a passed status if the Blacklisted element of the KeyDeviceDescriptor is set to FALSE, or the procedure shall return with a failed status if this Blacklisted element is set to TRUE.

Otherwise, the procedure shall return with a failed status.

4.5.4 Post-incoming frame security material retrieval consistency checks

If the Exempt element of the DeviceDescriptor is set to FALSE and if the incoming security level checking procedure of the step above obtained as output the conditionally passed status, the procedure shall return with the unsecured frame, the security level, the key identifier mode, the key source, the key index, and a status of IMPROPER_SECURITY_LEVEL.

The procedure shall set the frame counter to the Frame Counter field of the auxiliary security header of the frame to be unsecured. If the frame counter has the value 0xffffffff, the procedure shall return a status of COUNTER_ERROR.

The procedure shall determine whether the frame counter is greater than or equal to the FrameCounter element of the DeviceDescriptor. If this check fails, the procedure shall set the unsecured frame to be the frame to be unsecured and return with the unsecured frame, the security level, the key identifier mode, the key source, the key index, and a status of COUNTER_ERROR.

4.5.4.1 Incoming key usage policy checking crocedure

The procedure, for each KeyUsageDescriptor in the KeyUsageList of the KeyDescriptor:

- if the frame type is not equal to 0x03 (i.e. MAC Command Frame) and the frame type is equal to the FrameType element of the KeyUsageDescriptor, the procedure shall return with a passed status;
- if the frame type is equal to 0x03, the frame type is equal to the FrameType element of the KeyUsageDescriptor, and the command frame identifier is equal to the CommandFrame-Identifier element of the KeyUsageDescriptor, the procedure shall return with a passed status.

Otherwise, the procedure shall return with a failed status.

Chapter 5

Evaluations and future works

5.1 Image size

According to the tecnical feature of Tmote sky mentioned in Section 3.1, the executable image size can not exceed 48K, otherwise it can not be installed on the mote.

As a conclusion of the development phase, it has been made an evaluation of the memory usage, comparing the TinyOS image size when security was not implemented and when all the security features were fully implemented and used (i.e. code and data structures). As Figure 5.1 shows, it came out that security requires a not negligible amount of memory: about 7K for the RFDs and 6K more for the coordinator. The reason for the security implementation requiring more memory on the RFD than on the FFD Coordinator is that the RFDs must be able to set a different SecLevel and KeyIdMode for each frame, while the Coordinator just parses the frames' header relying on the RFD's decisions. Observe that, in this implementation, only the Coordinator (RFDs) receives (transmit) secured frames.

Going into more depth, it has been considered the costs of the implementation of single security features in terms of memory usage so distinguishing, among different components, which one impacts more on the memory footprint.

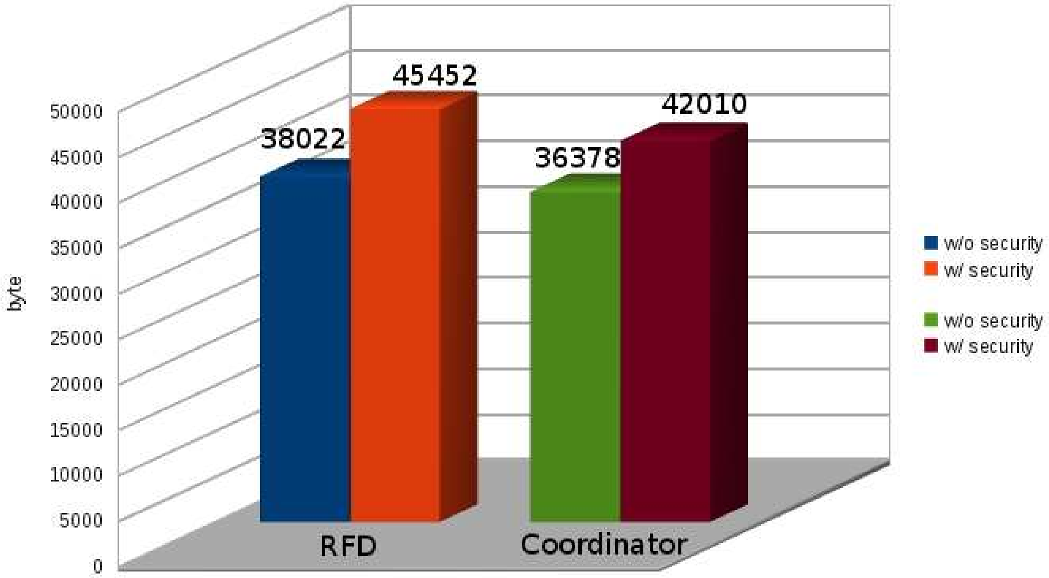


Figure 5.1: Security costs bar chart

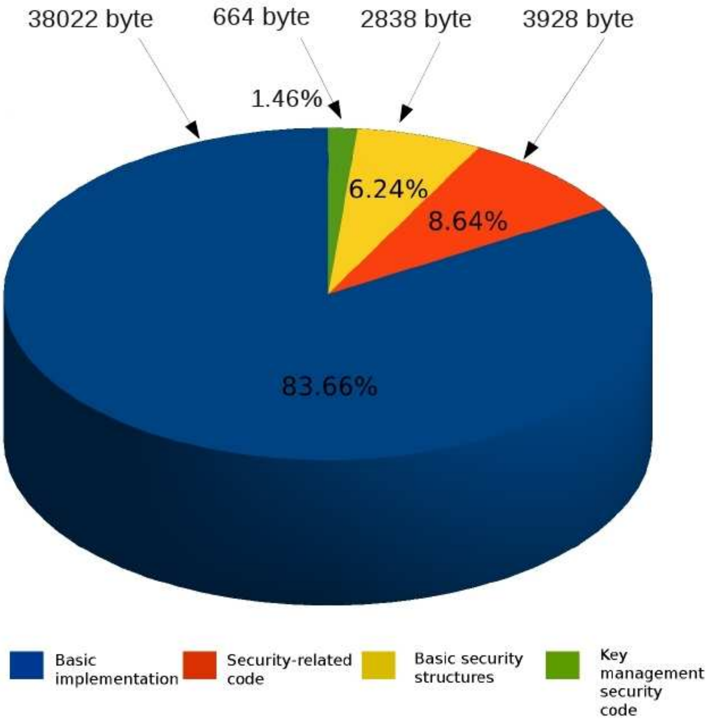


Figure 5.2: Sender side security pie chart

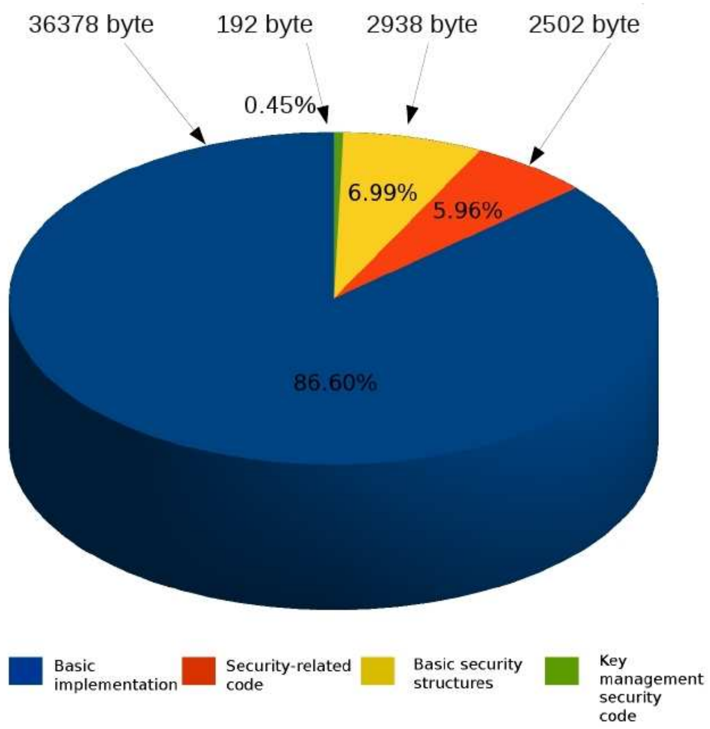


Figure 5.3: Receiver side security pie chart

The security implementation consists in three main components: one for the security data structures such as the key table or the device table, one for the code to get and set the security data structures and one for the key modes setup and management. As shown in Figure 5.2 and Figure 5.3, key management structures need considerably less memory than other security components. It also has been made an evaluation of the memory requirements when multiple keys are used and came out it does not increase the image size considerably, so, by employing a small amount of resources, a KeySource device is able to provide more keys and change it more frequently, enhancing the robustness of the security system.

Besides, it can be seen this pre-integration implementation of security in TinyOS drains almost all the available memory left on Tmote sky. On the other hand, integrating these features in the release 2.1.1 of TinyOS and writing ad hoc security modules and interfaces, the memory footprint reduced its size considerably. This advatage is due to two important aspects: the first one is the complete removal of printf() calls that determined the removal of the printf driver from the image, saving some memory. The second one is the introduction of interfaces, because the compiler is able to optimize the code exploiting the call mechanism, which acts as an inline expansion of the function called.

5.2 Error Management

As explained in Chapter 4, implementig the MAC layer security operations, it has been completed also the implementation of MAC layer security consistency checks described by the standard, introducing all the provided ad hoc status describing differnt kind of failures.

From the point of view of failure management, it would be useful to distinguish between general failure errors and different kinds of retry errors, that can be easily faced by means of a second execution attempt. For example, UNSUPPORTED_SECURITY or FRAME_TOO_LONG could be treated as FAIL errors, because, when they happen, no security operation can be carried on successfully. On the contrary, an IMPROPER_SECURITY_LEVEL or FRAME_COUNTER_ERROR status could be considered RETRY errors, because they are due to a temporary mismatch between security parameters which would be successfully fixed with a retry.

The current implementation of the procedure just returns a FAIL/SUCCESS status. It would be useful to return these status messages in case of failure, so making it possible to exploit them for diagnostic purposes and improve performances thanks to the above-mentioned distinction between FAIL and RETRY errors.

5.3 Test Application

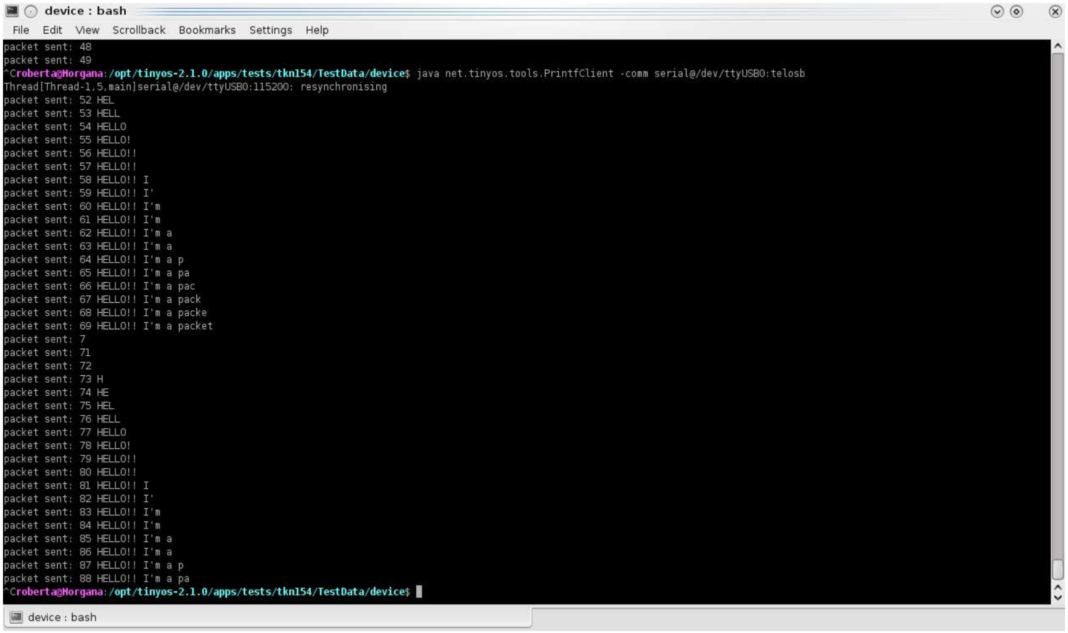
The TestDataSecurity application has been developed in order to test the discussed and implemented security features. This application represents a security-modified version of the TestData application. The classical TestData sees one or more RFDs continously sending packets to the coordinator in the context of a beacon enabled PAN. The TestDataSecurity's scenario consists in some RFDs sending diffent kinds of secured packets to a FFD, acting as a coordinator, which receives these secured packets and usecures them coherently with KeyIdModes and SecLevels. The application still works in a beacon enabled PAN.

5.3.1 Sender side

On the sender side, each RFD starts looking for an association to a PAN coordinator, this is done easily thanks to the beacon the coordinator periodically sends around the network. After that, it retrieves the key and creates secured packets according to the Security Level it has been set for. Figure 5.4 shows a screenshot of the execution output of a Tmote sky behaving as a RFD when sending secured packets whose payload changes both in content and length.

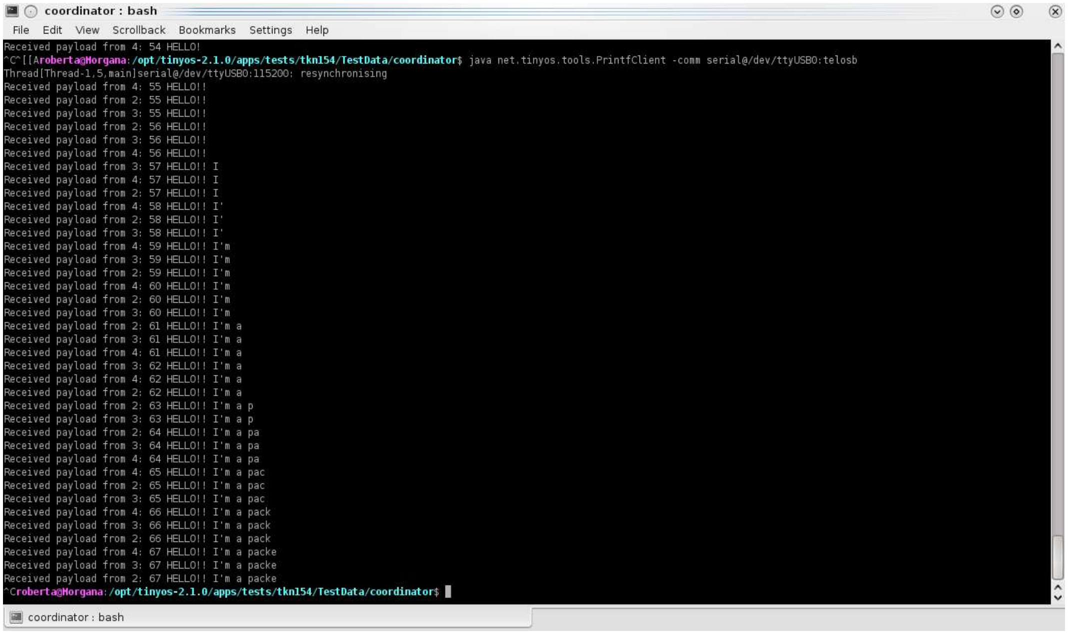
5.3.2 Receiver side

On the receiver side, the PAN coordinator FFD periodically sends beacon to allow the RFDs to join the network. Every time it receives a packet, it retrieves the key and unsecures the received packet, giving a feedback about the decryption/authentication result. Figure 5.5 shows a screenshot of the execution output of a tmote sky behaving as a PAN ccordinator and receiving and unsecuring packets whose payload changes both in content and length and come from different senders, which applies differnt SecLevels.



```
packet sent: 48
packet sent: 49
^Croberta@Morgana:/opt/tinyos-2.1.0/apps/tests/tkn154/TestData/device$ java net.tinyos.tools.PrintfClient -comm serial@/dev/ttyUSB0:telosb
Thread[Thread-1,5,main]serial@/dev/ttyUSB0:115200: resynchronising
packet sent: 52 HEL
packet sent: 53 HELL
packet sent: 54 HELLO
packet sent: 55 HELLO!
packet sent: 56 HELLO!!
packet sent: 57 HELLO!!
packet sent: 58 HELLO!! I
packet sent: 59 HELLO!! I'
packet sent: 60 HELLO!! I'm
packet sent: 61 HELLO!! I'm
packet sent: 62 HELLO!! I'm a
packet sent: 63 HELLO!! I'm a
packet sent: 64 HELLO!! I'm a p
packet sent: 65 HELLO!! I'm a pa
packet sent: 66 HELLO!! I'm a pac
packet sent: 67 HELLO!! I'm a pack
packet sent: 68 HELLO!! I'm a packe
packet sent: 69 HELLO!! I'm a packet
packet sent: 7
packet sent: 71
packet sent: 72
packet sent: 73 H
packet sent: 74 HE
packet sent: 75 HEL
packet sent: 76 HELL
packet sent: 77 HELLO
packet sent: 78 HELLO!
packet sent: 79 HELLO!!
packet sent: 80 HELLO!!
packet sent: 81 HELLO!! I
packet sent: 82 HELLO!! I'
packet sent: 83 HELLO!! I'm
packet sent: 84 HELLO!! I'm
packet sent: 85 HELLO!! I'm a
packet sent: 86 HELLO!! I'm a
packet sent: 87 HELLO!! I'm a p
packet sent: 88 HELLO!! I'm a pa
^Croberta@Morgana:/opt/tinyos-2.1.0/apps/tests/tkn154/TestData/device$
```

Figure 5.4: Sender side execution screenshot



```
Received payload from 4: 54 HELLO!
^C^[[Aroberta@Morgana:/opt/tinyos-2.1.0/apps/tests/tkn154/TestData/coordinator$ java net.tinyos.tools.PrintfClient -comm serial@/dev/ttyUSB0:telosb
Thread[Thread-1,5,main]serial@/dev/ttyUSB0:115200: resynchronising
Received payload from 4: 55 HELLO!!
Received payload from 2: 55 HELLO!!
Received payload from 3: 55 HELLO!!
Received payload from 2: 56 HELLO!!
Received payload from 3: 56 HELLO!!
Received payload from 4: 56 HELLO!!
Received payload from 3: 57 HELLO!! I
Received payload from 4: 57 HELLO!! I
Received payload from 2: 57 HELLO!! I
Received payload from 4: 58 HELLO!! I'
Received payload from 2: 58 HELLO!! I'
Received payload from 3: 58 HELLO!! I'
Received payload from 4: 59 HELLO!! I'm
Received payload from 3: 59 HELLO!! I'm
Received payload from 2: 59 HELLO!! I'm
Received payload from 4: 60 HELLO!! I'm
Received payload from 2: 60 HELLO!! I'm
Received payload from 3: 60 HELLO!! I'm
Received payload from 2: 61 HELLO!! I'm a
Received payload from 3: 61 HELLO!! I'm a
Received payload from 4: 61 HELLO!! I'm a
Received payload from 3: 62 HELLO!! I'm a
Received payload from 4: 62 HELLO!! I'm a
Received payload from 2: 62 HELLO!! I'm a
Received payload from 2: 63 HELLO!! I'm a p
Received payload from 3: 63 HELLO!! I'm a p
Received payload from 2: 64 HELLO!! I'm a pa
Received payload from 3: 64 HELLO!! I'm a pa
Received payload from 4: 64 HELLO!! I'm a pa
Received payload from 4: 65 HELLO!! I'm a pac
Received payload from 2: 65 HELLO!! I'm a pac
Received payload from 3: 65 HELLO!! I'm a pac
Received payload from 4: 66 HELLO!! I'm a pack
Received payload from 3: 66 HELLO!! I'm a pack
Received payload from 2: 66 HELLO!! I'm a pack
Received payload from 4: 67 HELLO!! I'm a packe
Received payload from 3: 67 HELLO!! I'm a packe
Received payload from 2: 67 HELLO!! I'm a packe
^Croberta@Morgana:/opt/tinyos-2.1.0/apps/tests/tkn154/TestData/coordinator$
```

Figure 5.5: Receiver side execution screenshot

5.4 Future works

The MAC layer security implementation preludes complex and useful utilizations in a wide range of monitoring applications which were claming for security and authentication, especially for military and medical application fields.

For the future, it would be interesting to do some further testing activities in more complex scenarios (e.g. to consider nodes both transmitting and receiving protected MAC frames). Furthermore, in order to have a full standard implementation, it is necessary to provide some minor security mechanisms, such as the opportunity to encrypt Beacon and Command frames. Moreover, to improve these security mechanisms, it could be worthwhile, starting from this implementation, to realize a run-time key distribution mechanism, for example by exploiting KeyIdMode2 or KeyIdMode3 to retrieve keys. Run-time key distribution requires concurrency management of security data structures and a method to get and manage a writing lock on the key tables for both the RFDs and the Coordinator. Depending on how it is implemented, the locking could determine the loss of a small number of packets or the incorrect unsecuring of some packets. However, these issues should be easily faced by means of retransmissions.

An integration activity of this suite in the IEEE 802.15.4 implementation of the TinyOS official release is currently ongoing, in cooperation with the *Technische Universitaet Berlin*. According to the results achieved so far, it is possible to noticeably reduce the memory footprint both on the RFDs and the Coordinator by writing ad hoc security modules which are well-optimized by the TinyOS's building policy. In fact, when compiling a specific application, the compiler introduces only the modules that the specific application needs. However, the RFDs' image size still remains bigger than the Coordinator's one because the computational load continues to be more on the sender side. By completely removing the printf() calls, it is possible to further reduce the image size, having more available memory and being able to introduce security in more complex application scenarios (i.e. much more memory would be available for the application).

Chapter 6

Conclusion

The IEEE 802.15.4 security suite allows network nodes to protect communications by encrypting and authenticating MAC frames. Such security mechanisms are provided by Tmote sky motes and their CC2420 chipset. This thesis work extends the IEEE 802.15.4 TinyOS implementation, introducing security support on the Tmote sky platform. Security modules have been tested and their impact on memory usage has been evaluated.

In particular, the realized security implementation makes it possible to protect MAC frames in different ways and manage dynamic security information retrieval from specific data structures, according to different available criteria. The security suite has been succesfully tested through a proper application, which allows many sender nodes to securely communicate with a single receiver node. Specifically, MAC data frames having payloads different in content and size have been considered. Finally, a memory footprint of the RFD and FFD code has been produced, highlighting how security impacts on memory usage quite remarkably, almost draining all available motes' memory.

For the future, it would be interesting to consider more complex testing scenarios, include some still missing minor features, and realize a run-time key distribution mechanism. An integration with the TinyOS official release is currently ongoing, together with the Technical University of Berlin.

Appendix A

MAC security procedures

A.1 Incoming frame security procedure

The following is the implementation of the consistency checks described in the section 7.5.8.2.1 of the IEEE 802.15.4 standard.

The inputs to this procedure are the frame to be secured, the security parameters and the mic_length. The output is the return status of the procedure.

```
ieee154_status_t frame_security_procedure
    (ieee154_security_t* security, message_t* frame, uint8_t mic_length)
{
  uint8_t *mhr = MHR(frame);
  ieee154_macSecurityEnabled_t mac_security_enabled;
  uint8_t pktLen = 0;
  uint8_t AddLen = 0;
  uint8_t SecLen = 0;
  eee154_macFrameCounter_t counter;
  error_t error;

  if (mhr[MHR_INDEX_FC1] == FC1_SECURITY_ENABLED &&
      (security->SecurityLevel == NO_SEC))
    return IEEE154_UNSUPPORTED_SECURITY;

  mac_security_enabled = call MLME_GET.macSecurityEnabled();
  if( mac_security_enabled == FALSE &&
      (security->SecurityLevel == NO_SEC))
    return IEEE154_UNSUPPORTED_SECURITY;

  call FrameUtility.getAddressingFieldsLength
    (mhr[MHR_INDEX_FC1], mhr[MHR_INDEX_FC2], &AddLen);
  pktLen += AddLen;

  call FrameUtility.getSecurityHeaderLength
    (mhr[MHR_INDEX_FC1], mhr[AddLen], &SecLen);
  pktLen += SecLen;
  pktLen += call Frame.getPayloadLength(frame);
  pktLen += mic_length;
  pktLen += 2;
  if(pktLen > IEEE154_aMaxPHYPacketSize)
    return IEEE154_FRAME_TOO_LONG;

  counter = call MLME_GET.macFrameCounter();
  if (counter == 0xffffffff)
    security->KeyIndex = (security->KeyIndex + 1)%2;

  error = key_retrieval_procedure
    (frame, security, security->KeyIndex);
  if (error != SUCCESS)
    return IEEE154_UNAVAILABLE_KEY;

    return IEEE154_SUCCESS;
}
```

A.2 Outgoing frame key retrieval procedure

This function is used in order to manage the key descriptor building procedure described in the section 7.5.8.2.2 of the IEEE 802.15.4 standard.

The inputs to this procedure are the frame to be secured and the security parameters. The outputs from this procedure is a SUCCESS/FAIL status and, in case of success, a key.

```
error_t key_retrieval_procedure
   (message_t* frame, ieee154_security_t* security, int key_index)
{
  error_t error;
  uint8_t *mhr = MHR(frame);
  error = key_descriptor_lookup_procedure(5,(mhr[3]));
  if(error == FAIL)
    return FAIL;
  return SUCCESS;
}
```

A.3 Incoming frame security procedure

The following implements the consistency checks described by section 7.5.8.2.3 of the IEEE 802.15.4 standard, The inputs to this procedure is the frame to be unsecured, the output is the status of the procedure.

```
ieee154_status_t frame_security_procedure
  (ieee154_security_t* mysec, uint8_t* mhr, uint8_t mic_length)
{
  uint8_t offset;
  ieee154_status_t status = IEEE154_SUCCESS;
  ieee154_macSecurityEnabled_t security_enabled;
  error_t error;
  uint8_t sec_level;
  ieee154_macFrameCounter_t macCounter;
  ieee154_FrameType_t frame_type;
  ieee154_DeviceDescriptor_t* device_descriptor = NULL;
  ieee154_KeyDescriptor_t* key_descriptor;

  if (mhr[MHR_INDEX_FC1] ==  FC1_SECURITY_ENABLED &&
      (mhr[MHR_INDEX_FC2] & FC2_FRAME_VERSION_MASK) == 0x00)
    status = IEEE154_UNSUPPORTED_LEGACY;

  call FrameUtility.getAddressingFieldsLength
```

```
    (mhr[MHR_INDEX_FC1], mhr[MHR_INDEX_FC2], &offset);
  sec_level = mhr[offset] & SEC_CNTL_LEVEL;

  if (mhr[MHR_INDEX_FC1] == FC1_SECURITY_ENABLED &&
      sec_level == 0x00 && status == IEEE154_SUCCESS)
    status = IEEE154_UNSUPPORTED_SECURITY;

  security_enabled = call MLME_GET.macSecurityEnabled();

  if (security_enabled == FALSE && status == IEEE154_SUCCESS)
    status = IEEE154_UNSUPPORTED_SECURITY;

  mysec->SecurityLevel = sec_level;
  mysec->KeyIdMode = ((mhr[offset++] & 0x18)>> 3);

  macCounter = *((nx_uint32_t*) (&(mhr[offset])));
  offset += 4;

  if (mysec->KeyIdMode != KEYIDMODE0){
    if (mysec->KeyIdMode == KEYIDMODE1)
      mysec->KeyIndex = mhr[offset++];

    else if (mysec->KeyIdMode == KEYIDMODE2){
      mysec->KeySource[0] = mhr[offset++];
      mysec->KeySource[1] = mhr[offset++];
      mysec->KeySource[2] = mhr[offset++];
      mysec->KeySource[3] = mhr[offset++];
      mysec->KeyIndex = mhr[offset++];
    }

    else {
      mysec->KeySource[0] = mhr[offset++];
      mysec->KeySource[1] = mhr[offset++];
      mysec->KeySource[2] = mhr[offset++];
      mysec->KeySource[3] = mhr[offset++];
      mysec->KeySource[4] = mhr[offset++];
      mysec->KeySource[5]  = mhr[offset++];
      mysec->KeySource[6]  = mhr[offset++];
      mysec->KeySource[7]  = mhr[offset++];
      mysec->KeyIndex = mhr[offset++];
    }
  }

  key_descriptor = call MLME_GET.macKeyTable(mysec->KeyIndex);

  device_descriptor->framecounter = ++macCounter;
    if(device_descriptor->framecounter == 0xffffffff)
      key_descriptor->keydevicelist
        [mysec->KeyIndex].blacklisted = TRUE;

  frame_type = mhr[MHR_INDEX_FC1] & FC1_FRAMETYPE_MASK;
  error = seclevel_checking_procedure(sec_level, frame_type, 0);
  if (((error == FAIL) ||
    (error == SUCCESS && conditionally_passed == TRUE))
   && status == IEEE154_SUCCESS)
    status = IEEE154_IMPROPER_SECURITY_LEVEL;

  error = security_material_retrieval_procedure(mhr, mysec, 0);
  if (error == FAIL && status == IEEE154_SUCCESS)
    status = IEEE154_UNAVAILABLE_KEY;

  error = key_usage_policy_checking(key_descriptor,frame_type,0);
  if (error == FAIL && status == IEEE154_SUCCESS)
    status = IEEE154_IMPROPER_KEY_TYPE;

  return status;
}
```

A.4 Incoming frame security material retrieval procedure

This function is used in order to manage the security material retrieval procedure described in the section 7.5.8.2.4 of the IEEE 802.15.4 standard. The inputs to this procedure are the frame to be secured and the security parameters. The outputs from this procedure is a SUCCESS/FAIL status and, in case of success, a key.

```
error_t security_material_retrieval_procedure (uint8_t* mhr,
   ieee154_security_t* security_params, uint8_t key_index)
{
  ieee154_macDefaultKeySource_t* def_key_source = NULL;
  error_t error;
  ieee154_LookupDescriptor_t device_lookup_table;
  ieee154_KeyDescriptor_t* key_descriptor = NULL;

  if (security_params->KeyIdMode == KEYIDMODE0 ||
      security_params->KeyIdMode == KEYIDMODE2)
    error = key_descriptor_lookup_procedure(5, &(mhr[3]));

  else if (security_params->KeyIdMode == KEYIDMODE1){
    def_key_source = call MLME_GET.macDefaultKeySource();
    error = key_descriptor_lookup_procedure(9, def_key_source);
  }
  else
    error = key_descriptor_lookup_procedure(9, &(mhr[3]));

  if(error == FAIL){
    call Leds.led0On();
    return FAIL;
  }

  if((mhr[MHR_INDEX_FC2] & FC2_DEST_MODE_MASK) ==
      FC2_DEST_MODE_SHORT)
  {
    device_lookup_table.lookupdatasize = 4;

    device_lookup_table.lookupdata[0] = mhr[3];
    device_lookup_table.lookupdata[1] = mhr[4];
    device_lookup_table.lookupdata[2] = mhr[5];
    device_lookup_table.lookupdata[3] = mhr[6];
  }

  error = blacklist_checking_procedure
    (key_descriptor, 5, &(mhr[3]));

  if(error == FAIL){
    call Leds.led0On();
    return FAIL;
  }

  return SUCCESS;
}
```

A.5 KeyDescriptor lookup procedure

According to the algorithm dascribed in the section 7.5.8.2.5 of the IEEE 802.15.4 standard, it makes some consistency controls over the retrieved key descriptor. It takes as input the size of the key lookup table and the key lookup data. It returns a SUCCESS/FAIL error_t and, if passed, a valid KeyDescriptor.

```
error_t key_descriptor_lookup_procedure (ieee154_LookUpDataSize_t
  key_lookup_size, ieee154_LookUpData_t*  key_lookup_data)
{
  uint8_t i, j;
  uint8_t lookup_data_size;
  uint8_t mac_key_table_entries;
  ieee154_KeyDescriptor_t* key_descriptor;
  mac_key_table_entries = call MLME_GET.macKeyTableEntries();

  for (i = 0; i < mac_key_table_entries; i++){
    key_descriptor = call MLME_GET.macKeyTable(i);
    lookup_data_size =
      (key_descriptor->keyidlookupdescriptor[0]).lookupdatasize;
    if (key_lookup_size != lookup_data_size)
      return FAIL;
    for (j = 0; j < lookup_data_size -1; j++){
      if (key_descriptor->keyidlookupdescriptor[i].lookupdata[j]
        != key_lookup_data[j])
          return FAIL;
    }
  }

  return SUCCESS;
}
```

A.6 Blacklist checking procedure

According to the algorithm dascribed in the section 7.5.8.2.6 of the IEEE 802.15.4 standard, it makes some controls over the retrieved key descriptor to check whether the key is valid or not. It takes as input the key descriptor pointer, the size of the key lookup table and the key lookup data. The procedure returns a SUCCESS/FAIL error_t and, if passed, a valid KeyDeviceDescriptor and a valid DeviceDescriptor.

```
error_t blacklist_checking_procedure(ieee154_KeyDescriptor_t*
  key_descr, ieee154_LookUpDataSize_t device_lookup_size,
  ieee154_LookUpData_t*  device_lookup_data)
{
  uint8_t i;
  error_t result;
  ieee154_DeviceDescriptor_t* device_descriptor;
  ieee154_KeyDeviceDescriptor_t* key_device_descriptor;
  for (i = 0; i <  key_descr->keydevicelistentries; i++){

    key_device_descriptor = &(key_descr->keydevicelist[i]);
    device_descriptor = call MLME_GET.macDeviceTable
        (key_device_descriptor->devicedescriptorhandle);

    if (key_device_descriptor->uniquedevice == TRUE &&
        key_device_descriptor->blacklisted == FALSE)
      return SUCCESS;

    else if (key_device_descriptor->uniquedevice == FALSE){
      result = device_descriptor_lookup_procedure
        (device_descriptor, device_lookup_size, device_lookup_data);

      if (result == SUCCESS &&
          key_device_descriptor->blacklisted == FALSE)
        return SUCCESS;
    }
  }
  return FAIL;
}
```

A.7 DeviceDescriptor lookup procedure

According to the algorithm dascribed in the section 7.5.8.2.7 of the IEEE 802.15.4 standard, it makes some consistency controls over the retrieved device descriptor. It takes as input the device descriptor pointer, the size of the device lookup table and the device lookup data. It returns a SUCCESS/FAIL error_t.

```
error_t device_descriptor_lookup_procedure
  (ieee154_DeviceDescriptor_t* device_descr,
  ieee154_LookUpDataSize_t device_lookup_size,
  ieee154_LookUpData_t*  device_lookup_data)
{
  ieee154_LookUpData_t* lookup_data = NULL;
  uint8_t i;

  if (device_lookup_size == 4){
    lookup_data[0] = (device_descr->panid) & 0x00ff;
    lookup_data[1] = ((device_descr->panid) & 0xff00) >> 8;
    lookup_data[2] = (device_descr->address.shortAddress) & 0x00ff;
    lookup_data[3] =
      ((device_descr->address.shortAddress) & 0xff00) >> 8;
  }
  else if (device_lookup_size == 8){
    lookup_data[0] =
      (device_descr->address.extendedAddress) & 0x000000ff;
    lookup_data[1] =
      ((device_descr->address.extendedAddress) & 0x0000ff00) >> 8;
    lookup_data[2] =
      ((device_descr->address.extendedAddress) & 0x00ff0000) >> 16;
    lookup_data[3] =
      ((device_descr->address.extendedAddress) & 0xff000000) >> 8;
  }
  for (i = 0; i < 4; i++){
    if(device_lookup_data[i] != lookup_data[i])
      return FAIL;
  }
  return SUCCESS;
}
```

A.8 Incoming security level checking procedure

This function is used in order to manage the incoming security level checking procedure described in the section 7.5.8.2.8 of the IEEE 802.15.4 standard. The inputs to this procedure is the incoming security level, a frame_type and, if needed, a command frame identifier. The output from this procedure is a SUCCESS/FAIL status.

```
error_t seclevel_checking_procedure( uint8_t sec1 ,
  ieee154_FrameType_t frame_type ,
  ieee154_CommandFrameIdentifier_t command_frame_identifier)
{
  uint8_t entries;
  uint8_t i;
  uint8_t sec2;
  ieee154_SecurityDescriptor_t* security_descr;

  entries = call MLME_GET.macSecurityLevelTableEntries();
  for(i = 0; i < entries; i++){
    security_descr = call MLME_GET.macSecurityLevelTable(i);
    if(security_descr->frametype == frame_type){
      if(frame_type == FC1_FRAMETYPE_CMD &&
          security_descr->commandframeidentifier !=
          command_frame_identifier)
        return FAIL;

      sec2 = security_descr->securityminimum;
      if(((sec1 & 0x04) < (sec2 & 0x04)) ||
         ((sec1 & 0x03) < (sec2 & 0x03))){
        if (sec1 == NO_SEC &&
            security_descr->deviceoverridesecurity == TRUE){
           conditionally_passed = TRUE;
           return SUCCESS;
        }
        else
          return FAIL;
      }
    }
  }
  return SUCCESS;
}
```

A.9 Incoming key usage policy checking procedure

This function implements the key usage policy checking procedure described in the section 7.5.8.2.9 of the IEEE 802.15.4 standard. The inputs to this procedure are the KeyDescriptor, the frame type and the CommandFrameIdentifier. The output from this procedure is a SUCCESS/FAIL status.

```
error_t key_usage_policy_checking
  (ieee154_KeyDescriptor_t* key_descr,
  ieee154_FrameType_t frame_type,
  ieee154_CommandFrameIdentifier_t command_frame_identifier)
{
  uint8_t entries;
  uint8_t i;

  entries = key_descr->keyusagelistentries;

  for(i = 0; i < entries; i++){
    if(key_descr-> keyusagelist[i].frametype == frame_type){

      if(frame_type == FC1_FRAMETYPE_CMD &&
          key_descr-> keyusagelist[i].commandframeidentifier
          != command_frame_identifier)
        return FAIL;
    }
    else
      return FAIL;
  }
  return SUCCESS;
}
```

Appendix B

TinyOS installation and setup

B.1 Installing TinyOS 2.1.1

B.1.1 Installing TinyOS 2.1.1 using TinyOS package repository

If you are running a version of Linux that supports Debian packages, the TinyOS package repository can be used to get the latest version of TinyOS. It can be done by means of the following instructions, as provided by [3].

- Remove any old TinyOS repository from the file /etc/apt/sources.list and add the following line (supported distributions are edgy, feisty, gutsy, hardy, jaunty, karmic, lucid):

  ```
  deb http://tinyos.stanford.edu/tinyos/dists/ubuntu <distribution> main
  ```

- Update your repository cache:

  ```
  sudo apt−get update
  ```

- Run the following to install the latest release of TinyOS and all its supported tools:

  ```
  sudo apt−get install tinyos
  ```

  This will likely give you a message to choose between the two available versions. As an example:

```
sudo apt-get install tinyos-2.1.1
```

- Add the following line to your ~/.bashrc or ~/.profile file in your home directory to set up the environment for TinyOS development at login.

  ```
  #Sourcing the tinyos environment variable setup script
   source /opt/tinyos-2.1.1/tinyos.sh
  ```

B.1.2 Installing TinyOS 2.1.1 using the TinyOS CVS

Another way to install TinyOS is by using the CVS located at [4].

- First of all, login to the CVS:

  ```
  cvs -d:pserver:anonymous@tinyos.cvs.sourceforge.net:/cvsroot/tinyos login
  ```

  just skip when it asks for password;

- then download the correct module

  ```
  cvs -z3 -d:pserver:anonymous@tinyos.cvs.sourceforge.net:/cvsroot/tinyos
  co -P modulename
  ```

  where modulename is tinyos-2.x.

- Finally, set your enviroment variables as follows:

  ```
  export TOSROOT=/home/roby/tos/tinyos-2.x
  export TOSDIR=$TOSROOT/tos
  export MAKERULES=$TOSROOT/support/make/Makerules
  export CLASSPATH=/home/roby/tos/tinyos-2.x/support/sdk/java/:$CLASSPATH
  ```

This installation method allows you to install a new parallel version of TinyOS if you already have one. Note that if you used the TinyOS debian repository in the past, keep in mind that all of the tools have been updated for TinyOS-2.1.1, but still work with all older versions of TinyOS as well. These conflicts should be OK so long as you remove any old packages; they are due to a change in the names of the updated packages installing into the same locations as the outdated ones.

B.2 NESCDT: An editor for nesC in Eclipse

the NESCEDT plugin is an editor for working with nesC code within Eclipse. Fundamentally, it allows you just create a new nescdt project in Eclipse and link those folders from your TinyOS installation and your application trees that are needed. Built-in keywords (which you can change) are used for syntax highlighting. The plugin does not touch, nor does it compete with your build system: you still build your applications with a command-line make.

B.2.1 Installing NESCDT

Start Eclipse and use the update site at [2] in the Eclipse update manager. Then save it in the Eclipse plugin directory (where the other jar plugins also reside). It would usually be /opt/eclipse/plugins on Linux. Finally restart Eclipse.

B.2.2 Using the plugin

Create a new empty nescdt project as showed in Figure B.1 and name it what you want to (for example nescdtsampleproject). When you open some .nc file it will be syntax colored according some some predefined rules in the plugin. There is auto-completion for keywords, types, and all other words found when then plugin scanned the .nc files in the linked folders. Press CTRL + SPACE to get the suggestions.

B.3 Compiling and installing a program

You compile TinyOS applications with the program make. TinyOS uses a powerful and extensible make system that allows you to easily add new platforms and compilation options. The makefile system definitions are located in tinyos-2.x/support/make. If you don't have mote hardware, you can compile it for TOSSIM, the TinyOS simulator.

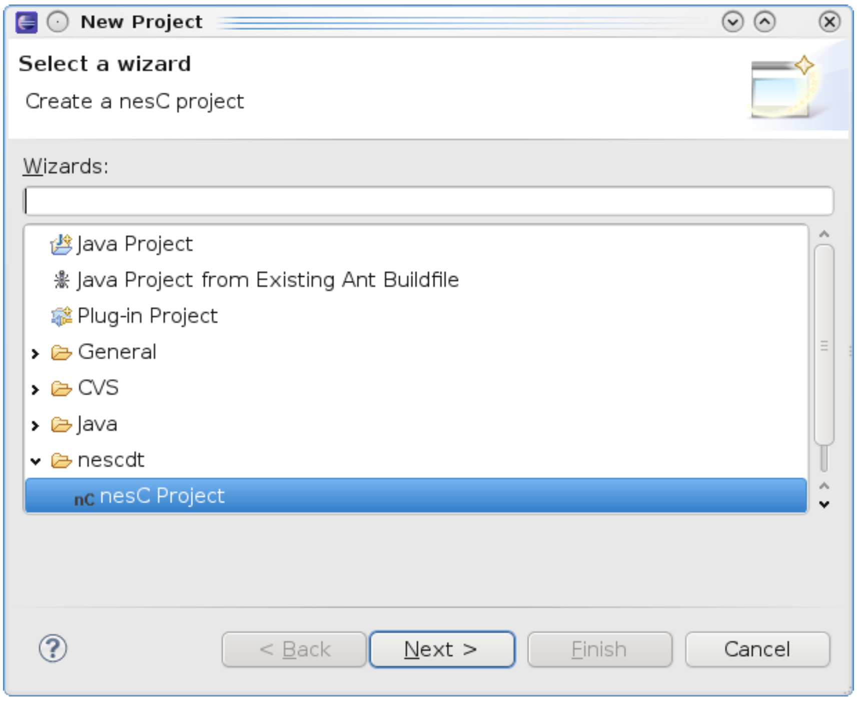


Figure B.1: How to create a new nesC project with NESCDT

The first step is to check that your environment is set up correctly. Run the tos-check-env command:

```
$ tos-check-env
```

This script checks everything that the TinyOS environment needs. If your system says some command is not available and you downloaded from CVS, then you need to compile and build the tools. Go to tinyos-2.x/tools/tinyos and type:

```
$ configure
$ make
$ make install
```

In order to install a program on a tmote sky, move to the program directory and then give the following command:

```
$ make telos install
```

If you need to give idetify a mote by means of a number which rapresents its identity, then give:

```
$ make telos install,<number>
```

B.4 The TinyOS printf library

In TinyOS debugging applications are very arduous. Debugging such a program typically involves flashing the three available LEDs in some intricate sequences. The TinyOS printf library helps debugging by providing the terminal printing functionality to TinyOS applications through motes connected to a PC via their serial interface. Messages are printed by calling printf() commands using a familiar syntax borrowed from the C programming language. In order to use this functionality, you simply need to include a single component in your top level configuration file (PrintfC), and include a ”printf.h” header file in any components that actually call printf() and printfflush().

To install the application on the mote, run the following set of commands:

```
cd $TOSROOT\tutorials\Printf
make telosb install bsl,/dev/ttyUSBXXX
```

To see the output generated by the Printf tutorial application you need to start the PrintfClient by running the following command:

```
java net.tinyos.tools.PrintfClient –comm serial@/dev/ttyUSBXXX:telosb
```

After resetting the mote, the following output should be printed to your screen:

```
Hi I am writing to you from my TinyOS application!!
Here is a uint8: 123
Here is a uint16: 12345
Here is a uint32: 1234567890
```

After this first compilation, you can see the output of all your programs just plugging them by the USB port and then providing the command:

```
java net.tinyos.tools.PrintfClient –comm serial@/dev/ttyUSBXXX:telosb
```

Acknowledgment

Heartfelt thanks to prof. Gianluca Dini and Marco Tiloca of Information Engineering Department of the University of Pisa for the technical support and advices, to Jan Hauer for the support he gave me during the time I spent at Technische Universitaet Berlin and to all guys I met there.

Thanks to my father Renato for having taught me how to face problems with a little bit of irony, to my mother Rosanna and my brother Riccardo for the constant encouragement.

Thanks to my best friend Ofelia ”Ophe” Puglia, I do not need complicated words to thank her because I am sure she understands how I feel. Thanks to my flat-mates: Federica ”Fede” Pantó, Carmen ”Pasca” Pascarelli, Maria Teresa ”Mari” Romano, Valentina ”Vale” Scocca and Manuela Febbraro for these years we spent together and the moral support they give me day by day.

Thanks to Daniel Cesarini, Francesco Giurlanda, Francesco Magno, and all the Green Lab’s students for sharing my difficulties and thanks to Cristiano Carnicelli for bearing all my questions about his work.

Bibliography

[1] Cc2420 2.4 ghz ieee 802.15.4/zigbee rf transceiver, www.chipcon.com/files/cc2420_data_sheet_1_3.pdf.

[2] Nescdt plugin, http://nxtmote.sf.net/nescdtupdate.

[3] Tinyos community forum, http://www.tinyos.net/.

[4] Tinyos cvs, http://sourceforge.net/projects/tinyos/develop.

[5] Tmote iv low power wireless sensor module, http://www.snm.ethz.ch/projects/tmotesky/tmote-sky-datasheet.pdf.

[6] Jon T. Adams. *An Introduction to IEEE STD 802.15.4*. 2100 E.Elliot Road, MD EL536, July 2006.

[7] Cristiano Carnicelli. Implementation of ieee 802.15.4 security on cc2420. Master's thesis, Università di Pisa, February 2010.

[8] Jan-Hinrich Hauer. *TKN15.4: An IEEE 802.15.4 MAC Implementation for TinyOS 2*, March 2009.

[9] Institute of Electrical and Electronics Engineers, Inc., New York. *IEEE Std. 802.15.4-2006, IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specic requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications*

for Low-Rate Wireless Personal Area Networks (WPANs), September 2006.

[10] Ida M. Savino. *Security suite for IEEE 802.15.4 MAC on TinyOS 2.x*, June 2009.