ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Beat Gebistorf

Secure Messaging for Wireless Sensor Networks

Semester Thesis, SA-2009-03
February 2009 until June 2009

Professor: Prof. Dr. Roger Wattenhofer
Advisor: Philipp Sommer & Roland Flury

Abstract

Wireless sensor nodes are deployed unnoticed in our daily life to support and assist us, e.g. monitoring room temperature or detecting fire. To receive trusted information from these sensor nodes they have to send authenticated messages. Even more confidential messages are required for sensitive information. In a nutshell this means to send secure messages.
This semester thesis explains the design and implementation of secure messaging on the example of the Meshbean900 and Pixie sensor node platforms. Both include a ZigBit900 module with dedicated hardware to perform cryptographic computations (128bit AES encryption) time- and power-efficiently. The procedure for the cryptographic computations are built on the specifications of the IEEE 802.15.4 communication protocol and implemented in TinyOS, an operating system for sensor nodes. The resulting implementation enables applications on the Meshbean900 and Pixie platforms to add authentication, integrity, confidentiality, and replay protection to outgoing messages.

Acknowledgments

First of all I would like to express my sincere gratitude to Prof. Dr. Roger Wattenhofer for giving me the opportunity to write this semester thesis in his research group.

I would also like to thank my advisors Philipp Sommer and Roland Flury for their constant support during this semester thesis. They always helped me to solve the difficult problems of this thesis. Without their assistance, this work would never have been possible.

Furthermore, I would like to thank my fiancée, my family and my flat share colleagues for supporting and motivating me during this thesis.

Contents

List of Figures

1 Introduction

This semester thesis explains the integration of security features on a specific sensor node platform such as Meshbean900 or Pixie. This integration includes the design, implementation and evaluation of the security features. In particular these features are the encryption and decryption of data, the authentication of messages, the replay protection of messages and the managing of keys.

1.1 Motivation

In the past few years Wireless Sensor Networks (WSN) found their application in many different areas, e.g. bird observation, glacier monitoring, vital sign monitoring or sniper localisation [16]. Several autonomous and spatially distributed sensor nodes which communicate with each other over wireless channels form a WSN. In a WSN the sensor nodes monitor cooperatively their surrounding area to measure parameters like brightness, temperature, pressure, sound or motion. Furthermore they are able to process the measured data with limited resources. As some WSN process sensitive data, they have to be able to secure the processed data tamper-proof.

Sensor nodes are normally low-power and low-cost devices with the requirement of a long autonomous lifetime. Therefore the nodes have to use the available power carefully and avoid expensive computations or radio transmissions. Reducing the computational power consumption can be achieved by implementing large or often used computations in hardware rather than

software. Considering this idea the Meshbean900 and Pixie platforms contain the ZigBit900 module [11] which includes a 128bit AES (Advanced Encryption Standard) hardware encryption module. Therefore the computational intensive AES cryptography can be performed power- and time-efficient on the dedicated hardware. Cryptography is needed to enable authenticity, confidentiality and replay protection of exchanged messages between sensor nodes. Authenticity is very important in WSN to get reliable monitoring information; confidentiality prevents revelation of exchanged data for unauthorized parties; and replay protection disables attackers to record messages and misuse the records for replay attacks [18].

1.2 Goals

Enable wireless sensor nodes, in particular the Meshbean900 and Pixie platforms, to send secure messages is the main goal of this semester thesis. This leads to the following subgoals:

- Implementing components in TinyOS for the ZigBit900 module to secure messages.
- Secure messages according to the IEEE 802.15.4 [8] communication protocol.
- Provide a basic key management.

1.3 Structure

This semester thesis is structured as follows:

- Chapter 2 introduces the IEEE 802.15.4 communication protocol, the ZigBit900 module, TinyOS, and the sensor node platforms Meshbean900 and Pixie in the first part. The second part describes two related implementations to the one in this thesis. Finally the terms security, AES, and Counter with CBC-MAC are explained in the last part.
- Chapter 3 covers the design of the cryptography stack to secure messages on the Meshbean900 and Pixie platforms.
- Chapter 4 describes the TinyOS components of the implementation.
- Chapter 5 illustrates the performance evaluation.
- Chapter 6 proposes future work and concludes the semester thesis.

2

Related Work

Explanations of relevant terms and elements concerning secure messaging in WSN are provided in this chapter. The first section gives an insight in sensor nodes, the communication standard IEEE 802.15.4 employed in this thesis, the ZigBit900 module, the Meshbean900 and Pixie platforms, and the operating system TinyOS. Section two gives a brief overview of security properties. It explains the terms en-/decryption, message authentication, message integrity, replay protection, introduces AES and describes the cryptographic Counter with CBC-MAC (CCM) mode.

2.1 Wireless Sensor Networks

A WSN consists of several independent sensor nodes. These sensor nodes are mainly deployed to perform monitoring tasks in a cooperative manner. The monitoring tasks can be divided in three categories [12]. The first category covers monitoring spaces and their characteristics, e.g. temperature of a room. The second encompasses the monitoring of things with their properties, e.g. speed of a car. The third consists of monitoring the interactions of things with each other and with the surrounding space, e.g. transformations of crevices. To be able to fulfil these tasks the sensor nodes are equipped with according sensors, e.g. light, pressure or temperature sensors. More properties of sensor nodes and of the Pixie and Meshbean900 platforms as representatives of sensor nodes are shown in this section. This includes the ZigBit900 module which is embedded in both platforms enabling them to

perform computations and communication. Furthermore the IEEE 802.15.4 communication protocol supported by ZigBit900 and used by the majority of wireless sensor nodes will be explained. The last part of this section describes TinyOS. For this semester thesis TinyOS as an operating system for sensor nodes is chosen to run on the Pixie and Meshbean900 platforms.

2.1.1 Sensor Nodes

Generally sensor nodes are small devices equipped with a microprocessor, data storage, sensors, analog-digital converters, a data transceiver, an energy source and a controller that connects the elements together [12]. One main goal of sensor nodes is to consume as little power as possible to guarantee a long lifetime without the need to change the power supply. This requirement arises from the fact that many sensor nodes are difficult to access physically, e.g. distributed in the mountains, on seas or room ceilings.

2.1.2 IEEE 802.15.4

The IEEE 802.15.4 standard is a communication protocol customized for wireless sensor nodes. It defines the wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate, low complexity and low power communication in wireless personal area networks (WPAN) [8].

Table 2.1: 8 security levels specified in IEEE 802.15.4

Identifier	Security Level	Description
0x00	None	No security applied to the message
0x01	MIC-32	Authentication with 4 bytes MAC
0x02	MIC-64	Authentication with 8 bytes MAC
0x03	MIC-128	Authentication with 16 bytes MAC
0x04	Encryption	Encryption of the payload
0x05	Encryption & MIC-32	Authentication with 4 bytes MAC and encryption of the payload & MAC
0x06	Encryption & MIC-64	Authentication with 8 bytes MAC and encryption of the payload & MAC
0x07	Encryption & MIC-128	Authentication with 16 bytes MAC and encryption of the payload & MAC

Security

The specifications of IEEE 802.15.4 relating to security include the definition of 8 security levels. These security levels enable an application to send messages with an adequate degree of security (see section 2.3 for security details). The application has to decide how much security overhead it can bear to get a higher security. It is a trade-off decision left open to the application. The possible security levels are described in table 2.1. The used term MIC (Message Integrity Check) is an alias for MAC whereas MIC-X means a MAC with length of X bits.
IEEE 802.15.4 prescribes message frame formats for the different security levels as shown in figure 2.1.

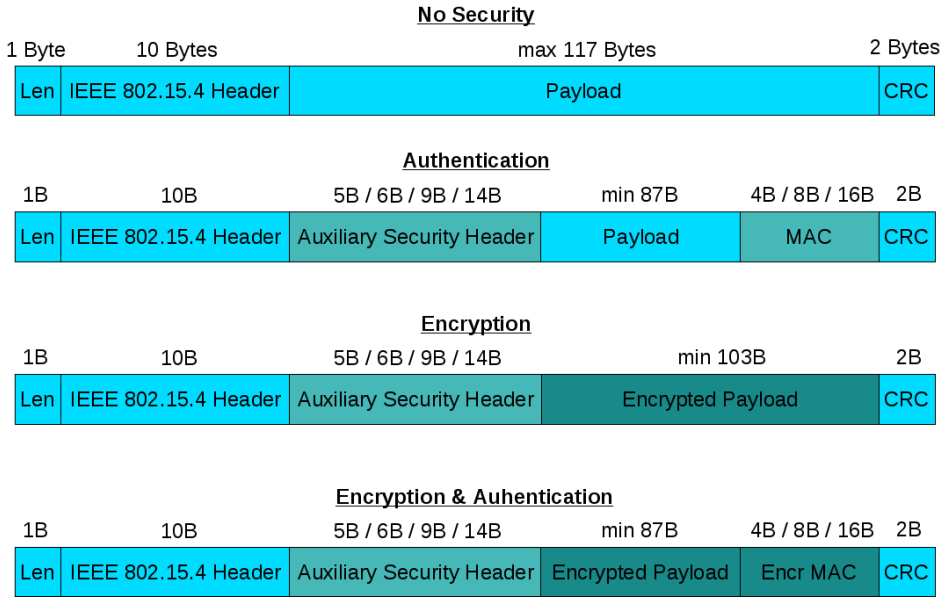

Figure 2.1: Frame formats according to IEEE 802.15.4 for different security levels.

The IEEE 802.15.4 header contains the frame control field (FCF). The security enable bit within FCF marks whether security features are applied to the message (bit set to one) or not (bit set to zero). If security features are applied to the message an additional header, the auxiliary security header, is concatenated to the IEEE 802.15.4 header. Included in the auxiliary security header are information to enable the receiver of the secured message to check the MAC of the message and/or decrypt the encrypted parts (see section 2.3.2 and 2.3.1 for explanations). The auxiliary security header consists of the the security control field (1 byte), the frame counter (4 bytes) and the key identifier (variable between 0 and 9 bytes) (see figure 2.2). The frame counter is increased for each outgoing message and provides replay

protection (see section 2.3.3). The key identifier field contains information about the key which has to be applied to check the MAC and/or decrypt the message. The key is specified by the key source in the meaning of the device which distributed the key and the key index for the case that multiple keys of the same key source exist. The security control field consists of the security

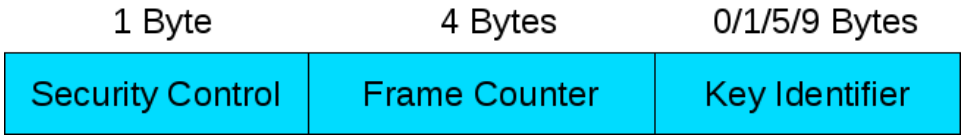

Figure 2.2: Auxiliary security header format

level information (3 bits), the key identifier mode (2 bits) and 3 reserved bits (see figure 2.3). The security level information encodes the identifier to

Figure 2.3: Security control field format

specify the security level applied to the message. The key identifier mode defines whether the key has to be determined implicitly by regarding the sender and receiver addresses of the message or explicitly by choosing a key of a key source. 4 different key identifier modes can be encoded with the two bits. Depending on the mode the length of the key source and hence of the key identifier in the auxiliary security header (see figure 2.2) varies from 0 bytes to 9 bytes.

2.1.3 ZigBit900

The ZigBit900 module of MeshNetics contains a radio transceiver (AT86RF212 of the RF2xx family) and a microcontroller (ATmega1281). It addresses the low power requirements of sensor nodes and guarantees a battery lifetime of 10 years [11]. Additionally ZigBit900 supports the IEEE 802.15.4 protocol. The communication between the microcontroller and the radio transceiver is handled using an SPI-bus [5]. The radio transceiver supports 128bit AES hardware encryption which performs 128bit block AES cryptography with a fix key length of 128 bits (see section 2.3.4). This hardware module forms the core of this thesis. The dedicated hardware reduces the time needed to perform the AES computations and hence reduces the power consumption.

SPI A serial peripheral interface (SPI) bus is used for synchronous serial data communication between devices in master/slave mode. The bus consists of four lines: clock, master output slave input, master input slave output and the select line. The master device selects a slave over the select line, provides a clock and initiates the data frame.

Meshbean900

Meshbean900 is a sensor node development board of the Meshbean family produced by MeshNetics containing a ZigBit900 module (see figure 2.4). Integrated in the board are a light and a temperature sensor. Supported interfaces are amongst others JTAG, USB, SPI and USART. For simple debugging cases 3 LEDs are placed on the board.

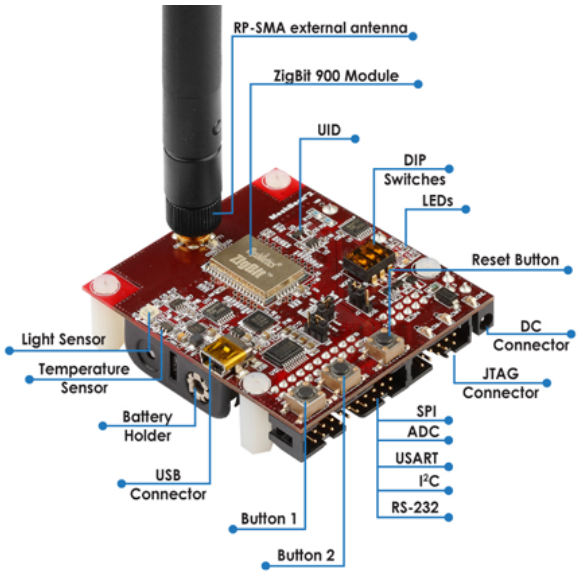

Figure 2.4: Picture of a Meshbean900 sensor node [10].

Pixie

Pixie is a sensor node developed by the DCG (Distributed Computing Group) of the ETH Zurich (see figure 2.5). ZigBit900 forms the core of the Pixie platform. Pixie has by default no sensors. It is used for dedicated applications and provides therefore many open pins to connect devices or sensors.

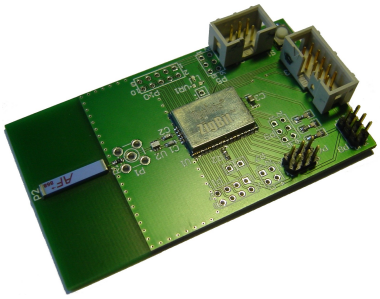

Figure 2.5: Picture of a Pixie sensor node.

2.1.4 TinyOS

TinyOS is an operating system developed for wireless sensor networks. It is open source and uses the nesc programming language [7]. Components build the structure of the operating system and enables a modular composition of needed software elements. Communication between components is assured with interfaces [4]. A component can use interfaces or provide interfaces (see figure 2.6). A component providing an interfaces has to implement all

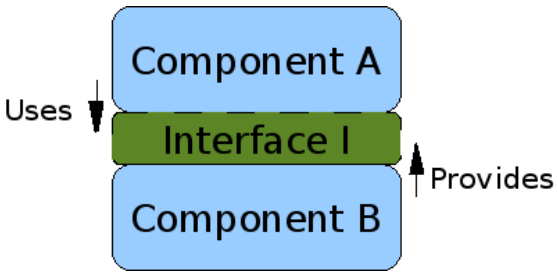

Figure 2.6: Providing and using interfaces in TinyOS.

commands defined in the interface and can signal events to components which use the interface. Components which use an interface can call commands of the interface and have to implement the handling of all events specified in the interface in case that they are triggered (see figure 2.7).

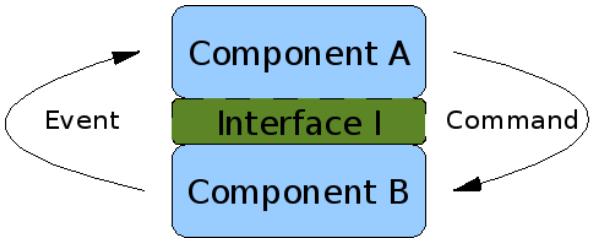

Figure 2.7: Events and commands in TinyOS.

2.2 Related Implementations

Similar to the implementation of this thesis there exist other solutions implementing security features for wireless sensor nodes. The solutions TinySec and AMSecure are described in this section.

TinySec TinySec, a pure software solution for secure messaging, claims to enable encryption and authentication of messages implemented without dedicated hardware and without major performance degradation [14]. Due to memory and computation limitations the security features chosen by TinySec are less secure than the ones of AES [20].

AMSecure AMSecure is a link-layer security component running on TinyOS for the CC2420 radio transceiver [2]. It is built on the IEEE 802.15.4 specifications. It provides message confidentiality, authentication, integrity, replay protection and semantic security. AMSecure relies on the hardware accelerated cryptography of the CC2420 chip. AMSecure therefore builds the counterpart of the implementation in this semester thesis with the following differences:

- AMSecure implements the older IEEE 802.15.4-2003 standard whereas this semester thesis bases upon IEEE 802.15.4-2006 with modified security features
- AMSecure relies on the CC2420 radio transceiver whereas this thesis relies on the AT86RF212 radio transceiver
- AMSecure runs under TinyOS version 1.x whereas the implementation of this thesis runs under TinyOS version 2.1

2.3 Security

Security is mainly achieved with cryptographic algorithms. Generally keys are used by cryptographic algorithms to compute the ciphertext of a plaintext. Different keys lead to different ciphertexts for the same plaintext. Keys are the core secret of a user or device. If a key is revealed, a third party can decrypt encrypted messages and authenticate forged messages. The security terms encryption, decryption, confidentiality, authentication and replay protection are explained in the first part of this section. The AES algorithm and the CCM mode are explained in a second part.

2.3.1 Confidentiality

Confidentiality of data means to assure that information contained in the data is only disclosed to users or devices for which the data was intended. This can be achieved by encrypting the data.

Encryption

Encryption means to transform data in a form which is unreadable for all who do not know the key. The transformation works according to a specified cryptographic algorithm like AES. The resulting string of this transformation is called ciphertext whereas the input is called plaintext. The transformation steps and hence the ciphertext depend on the key.

Decryption

Decryption means to recover a plaintext out of the ciphertext by applying the correct key[1] to the cryptographic algorithm which has the ciphertext as input.

Example Wireless sensor nodes for military purposes have not to send the secret measurements like positioning in plaintext and therefore reveal it to the adversary. Encryption of this data hence is required.

2.3.2 Authenticity

Authenticity of data means to assure that a receiver of the data is able to check whether the data originates from the claimed sender or not. Data integrity comes along with authenticity and means to be able to check whether the data was modified in transmission or not.

Message Authentication Code

Data integrity and authenticity can be assured by adding a message authentication code (MAC) to the end of a message which is similar to a hash of the message. The receiver can compute the MAC of the message itself and check whether it is the same as delivered in the MAC at the end of the

[1] For symmetric cryptographic algorithms the same key as for the encryption is used. For asymmetric cryptographic algorithm the correspondent private key has to be used.

message[2]. Throughout the rest of this thesis the abbreviation MAC invariably means message authentication code in contrast to the second meaning medium access control.

Example Responsible persons for buildings want their fire alarm system only to go off if a legitimate fire sensor triggers an alarm. Integrity and authenticity prevents attackers to be able to rise a false fire alarm by the means of faked messages.

2.3.3 Replay Protection

Replay protection means to assure that an attacker is not able to record a message and send it successfully[3] to a node at a later point in time. Replay protection can be achieved by adding a unique information to each message. The simplest way is to add the current number of a counter to the message and increase it afterwards. Hence each message contains a unique sequential number. If a receiver receives a message with the same number twice or with a number below the number of the most recently received message from the same sender the messages is rejected. This means that devices have to store the most recent counter numbers of all reachable devices.

Example Many cars can be opened with a wireless sender. Without replay protection an attacker can consequently record the signal and resend it later to the car and open it successfully. A communication protocol with replay protection disables this attack.

2.3.4 Advanced Encryption Standard (AES)

AES is the successor of DES (Data Encryption Standard). It is a symmetric key cryptographic algorithm to efficiently compute the ciphertext of a plaintext using a provided key. The efficiency origins in the fact that within the algorithm only bit-operations like XOR or cyclic shifting are applied. AES is hard to crack because some of these operations are non-linear. AES operates in block cipher mode which means to take whole blocks of fixed length as input. The block size specified for AES is 128 bits. The used keys can have length of 128 bits, 192 bits or 256 bits [6]. Longer keys provide stronger security guarantees.

[2]Obviously the receiver processes the message excluding the MAC at the end of the message to gain the correct result.

[3]Hereby successfully means that the receiver accepts the message.

2.3.5 Counter with CBC-MAC (CCM)

CCM stands for Counter with CBC-MAC [19]. CBC-MAC in turn stands for cipher block chaining message authentication code. CBC-MAC means that the message gets divided into blocks. Each block is XORed with the ciphertext of the previous cryptographic transformation and processed itself by the specified cryptographic algorithm (see figure 2.8).
CCM is a mode of operating a cryptographic algorithm like AES. The cryptographic algorithm has to be a block cipher algorithm with a block length of 128 bits to enable CCM. If an input block is smaller than 128 bit the missing bits have to be padded by zeros.
The CCM mode provides authentication and confidentiality of messages. CCM* is a minor variation of CCM specified in IEEE 802.15.4-2006 [8]. In the following CCM* is considered to illustrate the operations performed to enable message authentication, confidentiality and replay protection. The replay protection is accomplished by using a frame counter in the input of the authentication and encryption computations. This frame counter is unique for each message of a source. It is stored in the auxiliary security header to reveal it to the receiver.

Authentication

The CCM* mode authentication mechanism described in IEEE 802.15.4-2006 takes a message as input and gives a MAC of variable length as output. The first input to the authentication method is a 16 byte initial frame consisting of different fields (see figure 2.8): a flag field (1 byte), a nonce[4] field (13 bytes) and a field containing the message length (2 bytes). The nonce con-

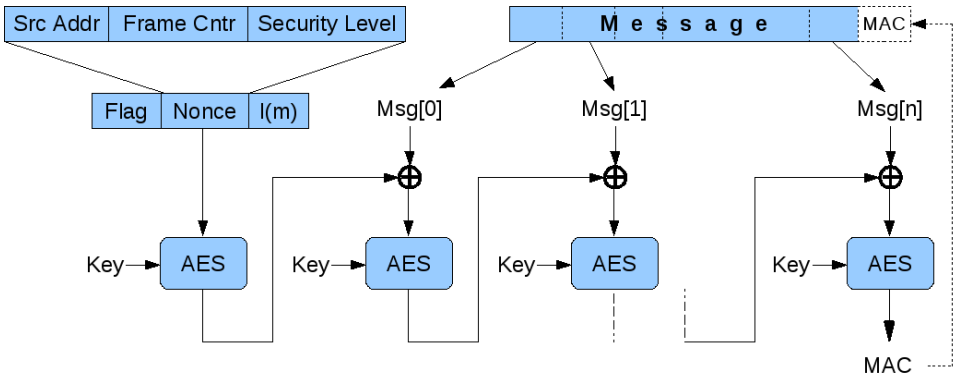

Figure 2.8: Authentication of a message according to CCM* with AES.

[4]Number used once. Included in messages to ensure the uniqueness of the message.

sists of the 8 byte[5] source address of the message, the 4 byte frame counter and the security level used for this message (1 byte). The initial frame provides uniqueness of the input because the source address combined with the frame counter are unique within a network.
The message is parsed and divided into 16 byte (128 bit) blocks. The initial frame gets encrypted with the specified block cipher. The resulting ciphertext of the initial frame gets XORed with the first block of the message and put to the block cipher again. The now resulting ciphertext is XORed with the second message block and put to the block cipher. This procedure is continued until the last block of the message is processed and hence ends the chain. The output of the last computation is the MAC. To achieve different length of the MAC one simply has to take the most significant bits of the resulting MAC until the needed length is reached and cut the rest off. This MAC is added to the end of the message.

Confidentiality

The CCM* mode confidentiality mechanism described in IEEE 802.15.4-2006 takes the payload/MAC of a message as input and gives the encrypted payload/MAC as output. The input to the encryption method is a 16 byte frame consisting of different fields (see figure 2.9): a flag field (1 byte), a nonce field (13 bytes) and a field containing a counter (2 bytes). The nonce

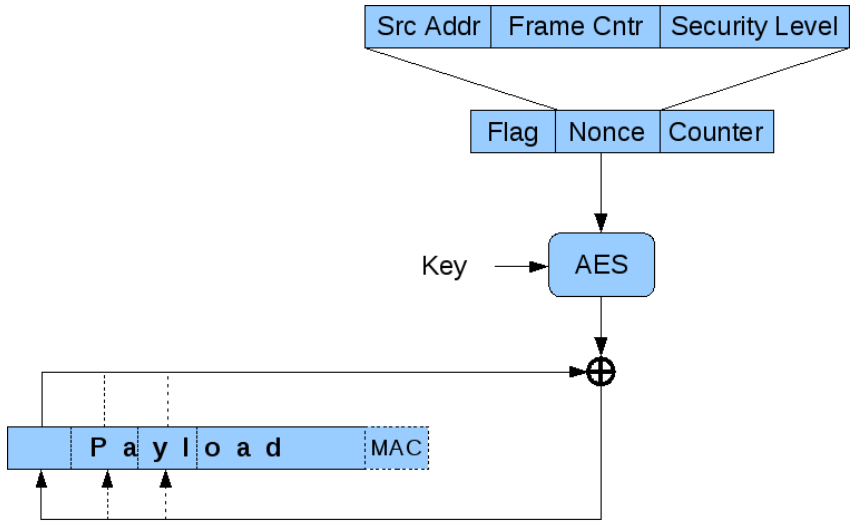

Figure 2.9: Encryption of a message according to CCM* with AES.

[5]The current TinyOS version 2.1.0 uses only a 2 byte source address which can be casted to a 8 byte value. According to the IEEE 802.15.4 specifications each node owns a 8 byte extended address and a 2 byte short address.

is the same as in the authentication. As difference to the authentication the encryption is not performed in a chain. The additional counter is used to assure that each encryption uses a unique input. The frame provides uniqueness with the source address, the frame counter and the additional counter in combination. The additional counter is initialised with the value zero and resetted to zero for each new message. After each encryption of a block the additional counter gets increased by one. The payload (and if existent also the MAC) are parsed and divided into 16 byte blocks. For each block the previously described frame containing a unique number of the additional counter is encrypted with the specified block cipher. The result XORed with the current payload/MAC block replaces the old payload/MAC block.

CCM vs. CCM*

The main differences between CCM and CCM* are the following:

- CCM only accepts MACs of a fixed length whereas CCM* allows to use variable MAC length.
- CCM* supports the security level encryption without authentication in contrast to CCM.

3 Design

To achieve the goal of secure messaging a device has to consider four areas. First of all applications on the device which want do send secured messages need an interface to do so (see figure 3.1). This secure messaging interface is described in the first section of this chapter. The second section illustrates the cryptographic functionalities needed to secure messages. In the third section the schemes for the key management and distribution are considered. Section four introduces the basic functionalities the radio driver has to implement. Finally section five provides an overview of the four described components.

3.1 Secure Messaging

An application which is intended to send sensitive data over a radio channel relies on an interface which provides the functionality to secure messages. This interface has to provide different configuration possibilities which are mentioned in the following list and are explained in more detail afterwards.

- Choose between data authenticity, data confidentiality, replay protection or combinations of them.
- Choose a key.
- Optionally choose the method of data authenticity, data confidentiality, replay protection or combinations of them.

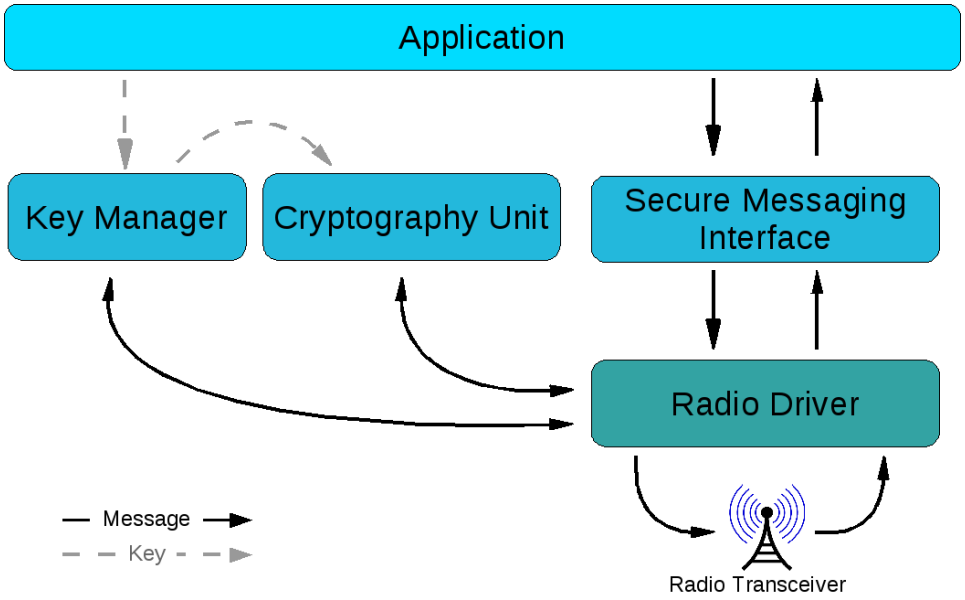

Figure 3.1: Cryptography stack for outgoing messages

This secure messaging interface is also used to forward received messages. The handling to secure messages or recover secured messages is done in the radio driver.

3.1.1 Authenticity, Confidentiality and Replay Protection

An application has to be able to choose the kind of security that should be used, e.g. choose authentication with a 32 bit MAC and no encryption. Whether to use authentication, confidentiality, replay protection or combinations of them. The IEEE 802.15.4 specifications deal with all three of them by using security levels and the CCM* operation mode (see section 2.3).

3.1.2 Key Selection

Keys are the core secret of cryptography. Several circumstances impose to use different keys or change keys regularly. An application using cryptographic mechanisms therefore has to be able to deploy and change a key. This can be done through the key manager, which is explained in section 3.3.

Circumstances In WSN sensor nodes can be removed from or added to the WSN. The sensor nodes might be exposed and easy to capture. It is relatively easy to read out the used keys of a captured sensor node without

tamper-resistance mechanisms [13]. Even without capturing a node, the communication can be monitored due to the broadcast characteristic of wireless communication. Hence the encrypted communication can be eavesdropped to detect the key[1]. Changing keys regularly or distributing and using keys only in a limited range mitigate the risk of keys being detected by attackers.

3.1.3 Cryptographic Methods

Allowing applications to choose between different cryptographic methods reduces undesirable overhead. Cryptographic methods have their characteristics such as speed, payload overhead and level of security. Choosing a method includes considering the tradeoffs. The option to choose according to the IEEE 802.15.4 standard means to choose the security level. The security level includes the choices to enable or disable encryption and to choose the length of the MAC (0, 32, 64 or 128 bits).

3.2 Cryptography

Beneath the secure messaging interface the cryptographic operations have to be performed by a cryptographic unit. The CCM* mode described in IEEE 802.15.4 presents a way to implement security features considering the low power constraints of sensor nodes (refer to 2.3.5). CCM* allows to add authenticity, integrity, confidentiality and replay protection to messages. The IEEE 802.14.5 standard forms the basis of this semester thesis and therefore the CCM* mode is applied in the implementation.

Authenticity

Adding a MAC to messages enables data authenticity and integrity (refer to 2.3.2). Corresponding to IEEE 802.15.4 an application can choose different lengths of the MAC. According to the low power requirements for sensor nodes one prefers shorter messages which need less transmission time and therefore less power. But in contrast, better authenticity and integrity warranties require to choose longer MAC.

Confidentiality

The encryption of messages provides confidentiality (refer to 2.3.1). The drawback of encryption is the longer execution time which means more power consumption.

[1] Not valid for AES. But holds for other cryptographic methods like WEP [17].

Replay Protection

To prevent replay attacks of messages a cryptographic method must guarantee that two identical successive messages are not accepted by the receiver (refer to 2.3.3). The nonce used within CCM* contains a frame counter which gets increased for each outgoing frame. This frame counter is integrated in the auxiliary security header and offers replay protection at the cost that it enlarges the overhead of the message which again means longer transmission times and hence more power consumption.

3.3 Key Management

Because keys are the central part of cryptographic algorithms it is important to manage them carefully with a key manager. The cryptographic unit consequently involves the key manager by asking for appropriate keys. The IEEE 802.15.4 protocol allows to introduce keys between two peer devices (link key) or within a group of devices (group key). This key scheme allows an application to choose between flexibility, key storage costs, key maintenance costs and cryptographic protection tradeoffs [8]. The key manager additionally is responsible to store security specific information beside the key like key identifiers, the frame counter of outgoing frames and frame counters of devices of incoming frames.

Link Key Link keys offer a high level of security. A cracked link key provides potentially less attack surface than a cracked group key and therefore causes less harm than a disclosed group key. Only the two peer devices using the key are afterwards vulnerable. One has to be aware that introducing link keys complicate the key maintenance and require a larger storage space to store the amount of link keys.

Group Key Group keys offer higher flexibility for devices to join or leave groups. Only one key per group has to be stored and the key maintenance effort is relatively small compared to the usage of link keys. Group keys used in peer-to-peer communications protect devices from outsider attacks. Malicious devices within the group however can cause harm to the whole group. Similarly the disclosure of a group key to an outsider makes the whole group vulnerable. The effort to revoke the key and distribute a new one is large.

Additional security information The keys stored in the key manager are distinguishable by key identifiers. Each key is stored in combination

with a key identifier which specifies which device established the keys (key source). The key identifier furthermore specifies which key of this key source is meant if several key from the same source exist (key index). This methodology allows to use a replacement strategy where a new key is introduced while an old one remains applicable. Beside key handling the key manager is responsible to maintain frame counters. Each outgoing frame is uniquely equipped with a frame counter value. If the counter reaches its limit, one has to establish a new key for this link or group and restart the counter. The key manager also has to maintain a list of all connected devices combined with the most recent frame counter value of the last received messages from them to check for replay attacks.
A secured message is only meaningful if the legitimate receiver is able to decrypt it or check the authenticity and integrity. To allow a receiver to perform the correct cryptographic transformations it requires security information. These information are handled by the key manager and are added to outgoing messages in the form of the auxiliary security header or read out of the auxiliary security header in case of incoming messages (see section 2.1.2).

3.4 Radio Driver

As link between the radio transceiver and the secure messaging interface of a device one needs a radio driver. This driver handles the communication of the microcontroller with the radio transceiver and involves all required components.
For both, incoming and outgoing messages, the radio driver forwards messages to the key manager and afterwards according to their security level to the cryptographic unit. If neither en-/decryption nor the concatenating/checking of a MAC is required according to the security level, then the radio driver passes the message only to the key manager to save computation time and therefore power.
As a last step for sending a message the radio driver initializes the transmission of the message over the radio transceiver. For received messages the last step is to pass the message to the secure messaging interface.

3.5 Overview

The previous four sections illustrated the basic components to enable secure messaging on a device. In this section an overview displays the cooperation of these components.

Sending Secured Messages

An application can send a secured message by passing it to the secure messaging interface (see figure 3.1). The application therefore has to specify the level of security and the key which have to be applied, otherwise default values are chosen (see chapter 4). The interface forwards the message to the radio driver which handles the message. It passes the message to the key manager which adds security information to the message (auxiliary security header). Afterwards it sends the message further to the cryptographic unit which adds the MAC and encrypts the payload according to the security level. The cryptographic unit asks the key manager for the appropriate key and performs the needed cryptographic algorithms. After receiving the message from the cryptographic unit the radio driver sends the message to the radio transceiver to transmit it.

Receiving Secured Messages

After receiving a message from the radio transceiver the radio driver forwards the message to the key manager. The key manager extracts the security information out of the auxiliary security header including the security level with which the message is secured. Afterwards it sends the message further to the cryptographic unit. The cryptographic unit asks the key manager for the appropriate key and performs the needed cryptographic algorithms. It checks the MAC and decrypts the payload according to the security level and returns the message to the radio driver. If the MAC is wrong the radio driver rejects the message. Otherwise the message is passed to the key manager again to remove the auxiliary security header. Finally the message reaches the application via the secure messaging interface.

4

Implementation

In this chapter the TinyOS components belonging to the secure messaging implementation of this semester thesis are explained. The primary goal of the implementation is to enable an application to use simple interfaces to secure messages. A secondary goal is to take advantage of the modularity of TinyOS by redirecting the normal send process to the securing components if requested by the application. This means that an application can use the standard interfaces (AMSend and Receive). The application solely has to set a define clause (`#define ENCRYPTION_ENABLED`) and setup a correct wiring at compile time (see section A.1).

4.1 Constructs

This section explains the most important constructs used within this secure messaging implementation which are not specified in IEEE 802.15.4.

Constructs

keyInfo An application can set the security level of outgoing messages with the interface SecurityConf (see section 4.2.1). This interface uses the parameter value keyInfo. keyInfo is a 16bit value which encodes the security level and the key identifier mode.

key key is a struct of 16 8bit values (key_part[0] - key_part[15]) to store a key used for cryptographic computations. Additionally the destination personal area network (PAN) address and the destination address can be stored in the struct. These two 16bit values define for which peer device this key will be used.

Security Level Aliases To facilitate setting security levels one can use the following predefined self-explanatory security level aliases: `SEC_LEVEL_NO`, `SEC_LEVEL_MIC_32`, `SEC_LEVEL_MIC_64`, `SEC_LEVEL_MIC_128`, `SEC_LEVEL_ENC`, `SEC_LEVEL_ENC_MIC_32`, `SEC_LEVEL_ENC_MIC_64`, `SEC_LEVEL_ENC_MIC_128`.

Default Key In the header file `Cryptography.h` a group key is defined which is applied to cryptographic operations if no other, more appropriate key is stored.

Number of Keys The number of keys stored in the `KeyManager` component can be defined in the header file `Cryptography.h` with the alias `MAX_KEYS`. The keys are replaced if more than `MAX_KEYS` are set with the command `setKey` in the `KeyManager` interface. The replacement is carried out in "first set, first replaced" order. The default key is excluded from this replacement. To replace the default key one has to call the command `setDefaultKey` in the `KeyManager` interface.

Security Metadata The security level and the key identifier mode for a message are stored in the metadata of the message to handle it in the driver layer. The metadata is not transmitted and is dropped before transmission.

4.2 Secure Messaging

The component `SecActiveMessageP` serves as link to secure messages for an application. It provides the TinyOS interfaces `AMSend`, `Receive` [9] and an additional security interface `SecurityConf` to set security parameters. If `ENCRYPTION_ENABLED` was defined at compile time it is recommended to use `SecActiveMessageP`. Otherwise one has to ensure that the security byte in the metadata of the message is set to zero to disable cryptographic transformations on the message.

4.2.1 Interface SecurityConf

The interface `SecurityConf` contains only the `setKeyInfo` command which takes a message, the length of the message and key information as parameter values. These key information are written into the metadata of the message. The default security level is set to `SEC_LEVEL_ENC_MIC_128` in the meaning of encoding and adding a MAC of 128bit length. The default value applies if an application skips the `SecurityConf` usage.

4.3 Radio Driver

The radio driver component `RF212DriverLayerP` maintains a state machine to keep track with the status of the radio transceiver. It is responsible to send and receive messages via the physical radio transceiver chip using the SPI-bus (please read section 2.1.3 or refer to the datasheet [1]). In the implementation of this semester thesis the radio driver was changed to handle the commands to apply the needed cryptographic operations. The radio driver therefore invokes commands of the `KeyManager` and `Cryptography` components. Because these two components might change the length of the message the radio driver has to be careful to set the length of the message correctly. The message length has to be adjusted using the `RF212DriverConfig.setLength` command each time an auxiliary security header or MAC is added or removed. In the platform initialisation phase called by the command `PlatformInit.init` the key defined in `Cryptography.h` is set as default key.

Send To send a message the radio driver extracts the security level rule for this message from the metadata. If no security is required the driver only ensures that the security enable bit in the frame control field (FCF) is set to zero and continues as without security features. Otherwise it calls the command `KeyManager.setAuxHeader` which adds the auxiliary security header to the message. Afterwards the FCF security enable bit is set to one. Subsequently the function `Cryptography.CCM` is called starting the cryptographic transformations on the message respective to the specified security level. Finally the message is sent via SPI-bus to the radio transceiver.

Receive An incoming message from the radio transceiver causes an interrupt leading to the execution of the `downloadMessage` function of the component `RF212DriverLayerP`. After reading out the message over the SPI-bus the radio driver checks whether the security enable bit in the FCF is set. If it is not set the driver skips the security relevant procedures and sets the security level in the metadata to zero. Otherwise it extracts the key infor-

mation including the security level out of the auxiliary security header by calling `KeyManager.getAuxHead`. The information is stored in the metadata of the message. The function `Cryptography.inverseCCM` computes as next step the decryption and checks the MAC respective to the security level. If the MAC is wrong the radio driver rejects the message similarly to when the checksum of the message is wrong. If either no MAC was used or the MAC was correct, the driver removes the auxiliary security header with the command `KeyManager.removeAuxHeader` and forwards the message to the next layer in the message stack like for unsecured messages.

4.4 Key Management

The component `KeyManagerP` serves as handler of keys and the auxiliary security header. It provides the interface `KeyManager`. If no keys are provided to the Key Manager the default key defined in the header file `Cryptography.h` applies.

4.4.1 Interface KeyManager

The interface `KeyManager` includes the commands `setKey, setDefaultKey, getKey, setDefault, getAuxHeader, removeAuxHeader` and `setAuxHeader`.

setKey Takes a key, a destination PAN and a destination address as parameter values and stores the key in combination with the destination PAN and the destination address.

setDefaultKey Takes a key as parameter value and stores it as the default key.

getKey Takes a destination PAN and destination address as parameter values and returns the most appropriate key for these parameters. If no key is appropriate the default key is returned.

setDefault Sets the key defined in the header file `Cryptography.h` as the default key.

getAuxHeader Takes a message as parameter value and returns the key information including the security level (keyInfo).

removeAuxHeader Takes a message as parameter value and removes the auxiliary security header from the message. It returns the new length of the message.

setAuxHeader Takes a message, its length and key information (keyInfo) as parameter values to add the auxiliary security header to the message. It returns the new length of the message.

4.5 Cryptography

The component `CryptographyP` serves as operator of cryptographic transformations on messages. It provides the `Cryptography` interface and implements the CCM* security operations specified in IEEE 802.15.4 (see section 2.3.5). An important part of these operations forms the CCM* nonce (see figure 4.1) which provides replay protection.
The `CryptographyP` heavily depends on the 128bit AES hardware module on the radio transceiver. To perform an encryption with the AES hardware module one needs 5 steps.

1. Set the 128bit key.
2. Select between electronic code book (ECB) and cipher block chaining (CBC) mode. In CBC mode the hardware automatically XORes the output of the previous encryption with the new input before performing the next encryption. In ECB mode the hardware only considers the current input.
3. Write data to SRAM via SPI-bus.
4. Start the AES operation.
5. Read the output of the operation from SRAM via SPI-bus.

For more details please refer to [8].

To use the `Cryptography` component it is required to ensure that

- the radio transceiver does currently not rest in the `Sleep` state and
- the SPI-bus is not used by another component at the same time.

4.5.1 Interface Cryptography

The interface `Cryptography` includes the commands `CCM` and `inverseCCM`.

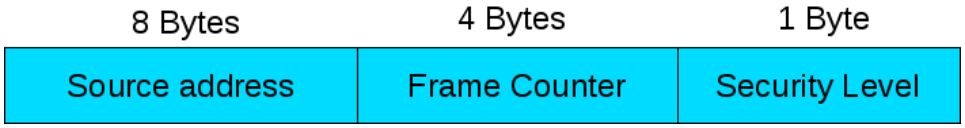

Figure 4.1: Nonce formatting for CCM*.

CCM Takes a message, its length and key information (keyInfo) as parameter values to encrypt the message and add a MAC according to the delivered security level information. It returns the new length of the message if a MAC had to be added.

inverseCCM Takes a message, its length and key information (keyInfo) as parameter values to decrypt the message and removes the MAC according to the delivered security level information. It returns the new length of the message if a MAC had to be removed.

5 Evaluation

This chapter gives an overview of the performance evaluations made for the implementation of this thesis. The evaluations include illustrations of the message size overhead produced by security features in section one and time overhead in section two.

5.1 Message Size Overhead

Security features causes the size of messages to grow (see figure 2.1). The consequences of the growth in size are longer transmission times for the radio transceiver and therefore higher power consumption. As sensor nodes are targeted on long lifetimes they require implementations which use as little power as possible. A developer of an application hence has to perform a proper tradeoff analysis to check whether the higher power consumptions are justifiable to use a higher security level[1].
Figures 5.1 and 5.2 show the overhead of messages produced by all security fields for different security levels. The security fields included in this analysis are the auxiliary security header and the MAC.

The x axis shows the payload whereas the y axis shows the percentage of overhead the security fields produce in comparison to the entire message in-

[1] Higher in this context means the following: Longer MAC-codes are more secure and therefore provide higher security. Encoding provides higher security than no encoding. Encoding and Authentication can not be compared as they target at different security goals.

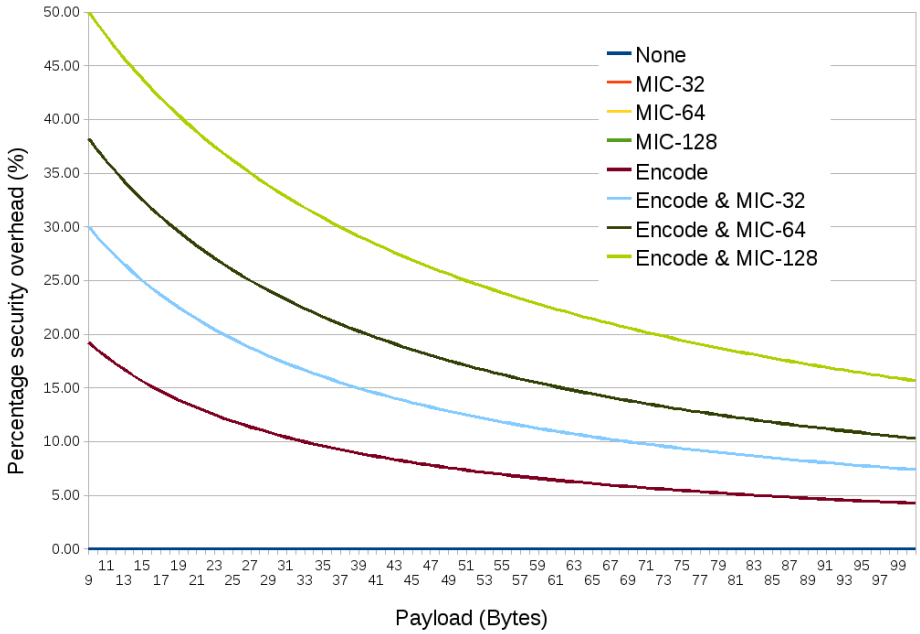

Figure 5.1: Percental message overhead of security fields for different security levels, in the best case.

cluding the message header and the checksum fields. Regarding the IEEE 802.15.4. specifications for security operations one observes that the encryption of a payload does not change the length of the payload, e.g. MIC-32 causes the same overhead in size as Encoding & MIC-32. This can be seen in both figures 5.1 and 5.2. In the evaluation two cases out of four possible are considered. The best case and the worst case regarding the size of the auxiliary security header. Depending on the key identifier mode the size of the auxiliary header can vary from 5 bytes to 14 bytes. In figure 5.1 the key identifier mode is set to determine the key implicitly which leads to a auxiliary security header size of 5 bytes. Figure 5.2 in contrast shows the overhead if the key identifier mode is set to determine the key explicitly form the 8 byte key source and the 1 byte key index subfield which leads to a auxiliary security header size of 14 bytes.

Concluding the figures it is very important to specify the needed security level in the design phase of an application accurately. Choosing a higher security level than needed can increase the message size in the worst case by 60 percent, e.g. choosing MIC-128 instead of no security at a payload size of 8 bytes. But choosing a too low security level offers incentives to attackers to break the system.

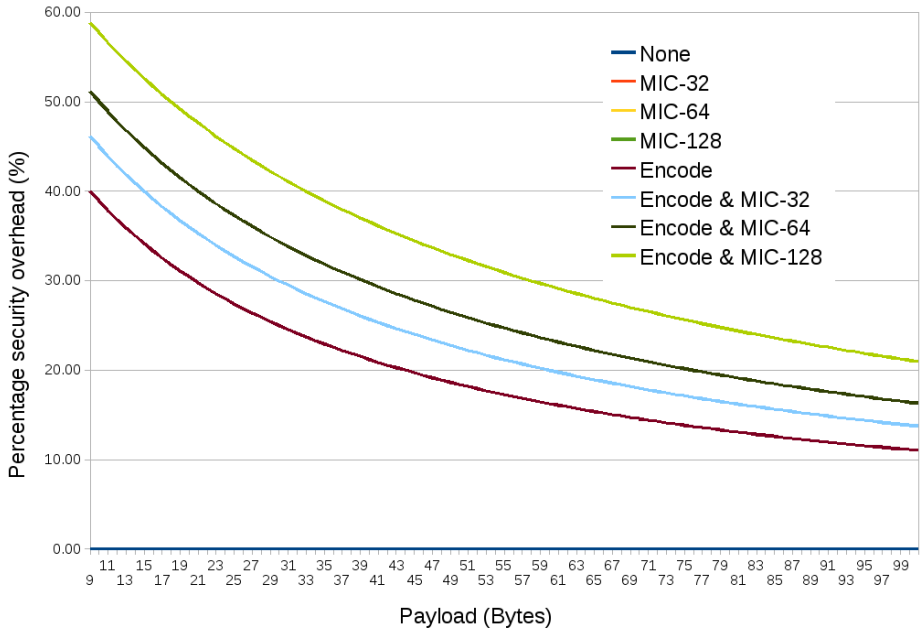

Figure 5.2: Percental message overhead of security fields for different security levels, in the worst case.

5.2 Time Overhead

The computation of security transformations on messages is time-intensive and therefore also power-intensive. It is again true that a developer of an application has to perform a proper tradeoff analysis to check whether the higher power consumptions are justifiable to use a higher security level. Because the higher power consumption shortens the lifetime of a sensor node.

Figures 5.3 and 5.4 show the time consumption for two different scopes. Figure 5.3 shows the average time consumed just for cryptographic methods in the component `Cryptography` on the y axis for different security levels. The x axis counts the bytes of payload included in the message. Figure 5.4 shows the average time consumed between sending a message over the secure messaging interface `SecActiveMessage` and getting the information `sendDone` on the y axis. The x axis counts the bytes of payload included in the message.
In figure 5.3 one is able to recognize the block cipher computation characteristic. The bends of the lower lines in the graph originate from the fact that the time consumption heavily depends on the number of blocks which have to be computed with the AES module on the ZigBit900 module. Each opened block causes an additional encryption cycle, e.g. 17 bytes payload

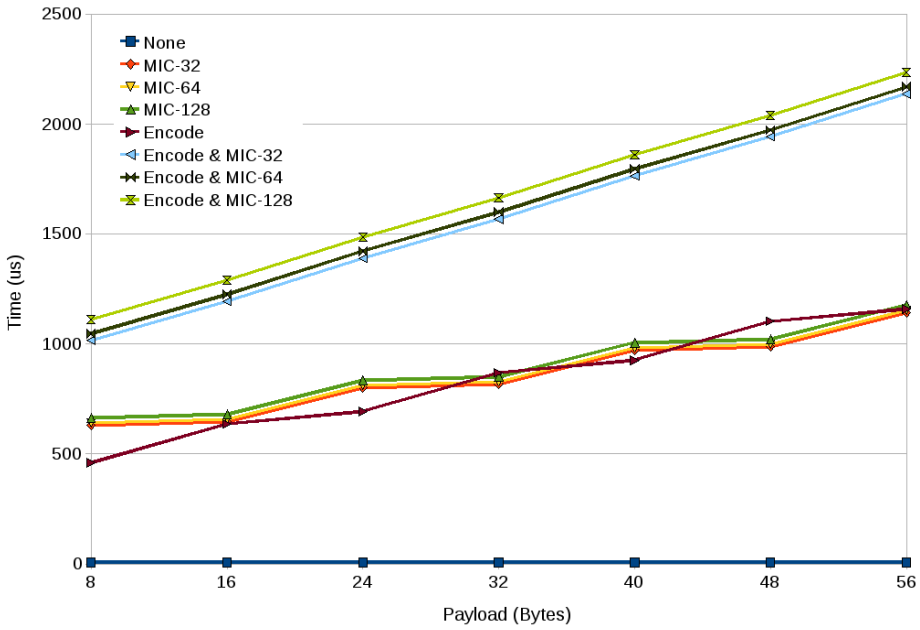

Figure 5.3: Time consumption only of the cryptographic methods for different security levels. The key is determined implicitly. Therefore the auxiliary header has a length of 5 bytes.

causes two encryption cycles[2] as well as 20 bytes payload. For the evaluation case the key is determined implicitly. Therefore the auxiliary header has a length of 5 bytes. The offset between the authentication-only and the encryption-only security levels in the graph originates from the IEEE 802.15.4 header and auxiliary security header. Authentication-only includes the IEEE 802.15.4 header, the auxiliary security header and the payload of the message for the cryptographic computations whereas encryption-only only includes the payload. The resulting difference of 18 bytes (1 block + 2 bytes) result in the offset in the graph. This offset has to little impact regarding time consumption to be visible in figure 5.4.

Concluding the figures it is again very important to specify the needed security level in the design phase of an application accurately. Choosing a higher security level than needed can increase the time needed to send a message in the worst case by 6 ms, e.g. choosing encryption & MIC-128 instead of no security. The longer the sending of a message takes the lower the throughput achieved and the more power is consumed by the process. It can also be seen that encryption of messages takes slightly more time than

[2] 17bytes=16bytes + 1byte. 128bit AES block cipher encryption takes for each cycle 128 bits = 16 Byte.

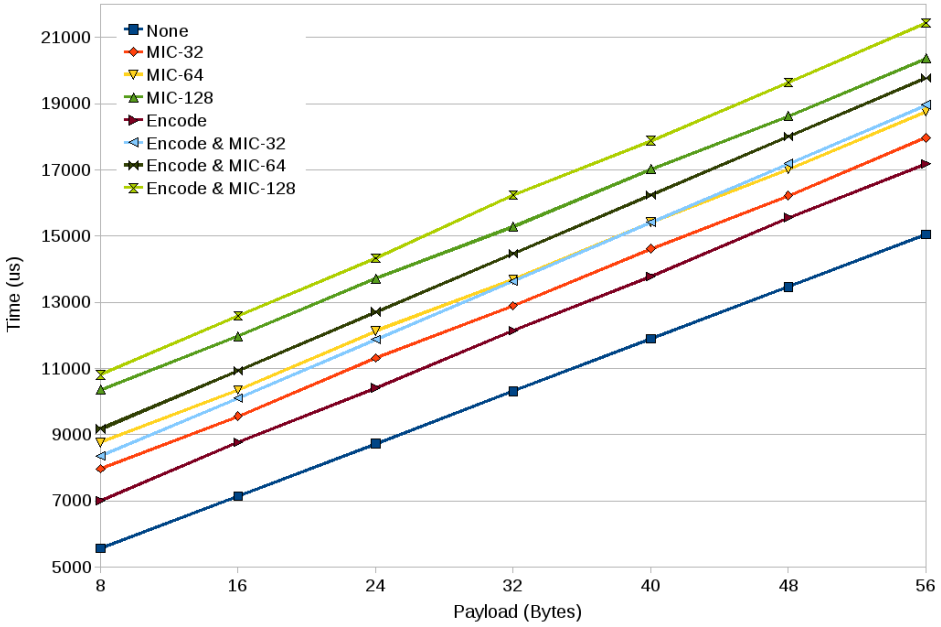

Figure 5.4: Time consumption of sending a secured message for different security levels. The key is determined implicitly. Therefore the auxiliary header has a length of 5 bytes.

authentication. This can be explained by the fact that authentication works in CBC mode in a chained manner (supported by the 128bit AES hardware module) whereas encryption works in ECB mode.

6 Future Work and Conclusion

This chapter provides proposals for future work in the first section and concludes the thesis in the second section.

6.1 Future Work

The implementation of this thesis covers the main requirements to secure messages. Left open are additional functionalities and security features as listed in the following:

- Extending the component `Cryptography` for general implementations of the IEEE 802.15.4 message formats, e.g. handling a footer.
- Uncoupling of the TinyOS component `Cryptography` to use it independent of the radio driver.
- Introduction and implementation of a key distribution mechanism.
- Implementation of the entire key management mechanism according to the IEEE 802.15.4 specifications.
- Extended performance analysis including power measurements.

6.2 Conclusion

Concluding this semester thesis one perceives that the goals are reached. The implementation enriches TinyOS with components to enable wireless sensor nodes containing a ZigBit900 module to secure messages according to the IEEE 802.15.4 communication protocol. Hence applications on such sensor nodes are able to add confidentiality, authenticity, integrity and replay protection to their messages by using a simple interface. Based on the 128bit AES hardware encryption unit included in the ZigBit900 module, cryptographic computations are performed fast and efficiently and meet the low-power requirement of sensor nodes. Furthermore eight selectable security levels offer flexibility to reduce unnecessary overhead and to apply an adequate security. Finally the key manager component allows applications to choose and change the key used for the cryptographic operations which adds additional security. We conclude that the Meshbean900 and Pixie sensor boards equipped with the ZigBit900 module are fit to face an insecure environment.

A

How To

This appendix provides basic information to alleviate programmers to use the implementation of this semester thesis. In the first section setup requirements to secure messages are explained. The second section gives hints to install the implementation. Finally the third section provides miscellaneous information which may help to develop applications based on secure messaging. Examples to send secured messages and receive secured messages are provided in the components `SecSenderApp` and `SecReceiverApp`, available on the DCG Wiki [3]. To access the DCG Wiki one needs an ETHZ account and the authorization of DCG (Distributed Computing Group of the ETH Zurich). Parts of the code are also provided in section A.3.2.

A.1 Setup

To use the secure messaging components the definition `#define ENCRYPTION_ENABLED` has to be set. This definition tells the preprocessor of the compiler to integrate the needed components and code used to secure messages. Furthermore one has to ensure to wire the needed component `SecActiveMessage` and choose the Interfaces `AMSend`, `Receive` and `SecurityConf` correctly as shown in the following.

A.1.1 Wiring

The wiring is important to include the secure messaging interface `SecActiveMessage`. In the following an exemplary application which uses `SecActiveMessage` to send secured messages is called `SecSenderApp` and one to receive secured messages `SecReceiverApp`. Therefore the following components have to be wired:

- `SecSenderApp.AMSend -> SecActiveMessageC.AMSend`
- `SecSenderApp.SecurityConf -> SecActiveMessageC.SecurityConf`
- `SecReceiverApp.Receive -> SecActiveMessageC.Receive`

A.1.2 Interfaces

The interfaces `AMSend` and `Receive` are the same as provided by the module `ActiveMessageLayerC [9]`. For more information please refer to the TinyOS tutorial [4]. The interface `SecurityConf` is specific for the security implementation. `SecurityConf` is an optional interface. If it is skipped the cryptographic stack will run with default values defined in the header file `Cryptography.h` (key) and the component `SecActiveMessage` (security level and key identifier mode). Details of the three mentioned interfaces are given in section A.3.

A.2 Installation

This section gives some hints to alleviate the installation and debugging of software on the Meshbean900 and Pixie platforms. Details and more information are described in the TinyOS tutorial [4]. Hints:

- An installation guide for the Meshbean900 and Pixie platforms is given on the DCG Wiki (see introduction of this chapter).
- An installation guide for Yeti 2, the TinyOS plugin for Eclipse, is given on the website `http://tos-ide.ethz.ch/wiki/pmwiki.php?n=Site.Installation`
- At USB problems, check the USB permission. Rules for the permission are provided in `/etc/udev/rules.d/` (for Linux distributions).
- For serial port access use the terminal or cutecom (cutecom is a user-friendly GUI). The serial port is useful for debuging purposes.

- For the terminal:
  - Add in the file `/opt/tinyos-2.1.0/support/sdk/java/net/tinyos/packet/BaudRate.java` the new devices "pixie" and "meshbean900" with baud rate `57600`.
  - In file `/opt/tinyos-2.1.0/tinyos.sh`: Modify the line `CLASSPATH=$CLASSPATH:$TOSROOT/support/sdk/java/` to `CLASSPATH=$CLASSPATH:$TOSROOT/support/sdk/java/tinyos.jar`.
  - You can start the terminal using: `java net.tinyos.tools.Listen -comm serial@/dev/ttyUSB0:meshbean900`.
  - The previous step can be simplified to `java net.tinyos.tools.Listen` if you add in the file `.bashrc` the line `MOTECOM=serial@/dev/ttyUSB0:57600`.
- For cutecom: Use as device `"/dev/ttyUSB0"` and as baud rate `"57600"`.

A.3 Miscellaneous

A.3.1 Interfaces

The two interfaces `AMSend` and `Receive` are wired for this thesis to the component `RF212ActiveMessage` (or more accurately to the component `ActiveMessageLayer`). The two interfaces thereby are parametrized interfaces which take an active message type as extra argument. This active message type allows to use the interface multiple time for different purposes. It can be compared to ports in a NAT [15].

AMSend The interface AMSend has two important commands (`send` and `getPayload`) and one event (`sendDone`).

- `send` sends a message. It takes the address of the receiver (`am_addr_t`), a pointer to the message (`message_t*`), and the length of the message (`uint8_t`) as argument values and returns whether the send procedure was successful (`error_t`).
- `getPayload` returns a pointer to the payload. It takes a pointer to the message (`message_t*`) and the length of the message (`uint8_t`) as argument values and returns a pointer to the payload (`void*`).
- `sendDone` informs about the sending status of the message. It provides a pointer to the sent message (`message_t*`) and whether the message was sent successfully, was canceled or failed (`error_t`). A new message has not to be sent until the `sendDone` of the previous one was signalled.

Receive The interface Receive has one event (`receive`).

- `receive` signals that a message was received. It provides a pointer to the received message (`message_t*`), a pointer to the payload (`void*`), and the length of the message (`uint8_t`) as argument values and returns a pointer to the allocated memory for a next message (`message_t*`).

SecurityConf The interface SecurityCond has one command (`setKeyInfo`).

- `setKeyInfo` stores the delivered key information in the metadata of the message. It takes a pointer to the message (`message_t*`), the length of the message (`uint8_t`), and key information (`keyInfo`) as argument values.

A.3.2 Example Application Code

SecSenderAppP Code

```
#include "Cryptography.h"

module SecSenderAppP{
  uses{
  interface Leds;
  interface Boot;
  interface SplitControl as RadioControl;
  interface AMSend[am_id_t tx_id];
  interface SecurityConf;
  }
}
implementation{

  message_t packet;
  am_id_t id = 11;

  event void Boot.booted() {
    call RadioControl.start();
  }

  event void RadioControl.startDone(error_t err) {
    simple_enc_msg_t* newmsg;
    newmsg = (simple_enc_msg_t*)call AMSend.getPayload[id](
      &packet, sizeof(simple_enc_msg_t));
    if (newmsg == NULL) {
      return;
    }
    newmsg->plain[0] = 0x12;
    newmsg->plain[1] = 0x34;
    newmsg->plain[2] = 0x56;
```

```
    newmsg->plain[3] = 0x78;
    newmsg->plain[4] = 0x12;
    newmsg->plain[5] = 0x34;
    newmsg->plain[6] = 0x56;
    newmsg->plain[7] = 0x78;
    newmsg->plain[8] = 0x12;
    newmsg->plain[9] = 0x34;
    newmsg->plain[10] = 0x56;
    newmsg->plain[11] = 0x78;
    newmsg->plain[12] = 0x12;
    newmsg->plain[13] = 0x34;
    newmsg->plain[14] = 0x56;
    newmsg->plain[15] = 0x78;

    call SecurityConf.setKeyInfo(&packet,
      sizeof(simple_enc_msg_t), msgcounter);
    call AMSend.send[id](AM_BROADCAST_ADDR, &packet,
      sizeof(simple_enc_msg_t));
    call Leds.led1Toggle();
  }

  event void RadioControl.stopDone(error_t err) {}

  event void AMSend.sendDone[am_id_t tx_id](message_t *msg,
    error_t error){}
}
```

SecSenderAppC Code (Wiring)

The wiring of SecReceiverApp can be done similarly.

```
#include "Cryptography.h"
#define ENCRYPTION_ENABLED

configuration SecSenderAppC{
}
implementation{
  components SecSenderAppP as App;
  components SecActiveMessageC;
  components MainC, LedsC;
  components ActiveMessageC;
  components RF212ActiveMessageC;

  App.Boot -> MainC.Boot;
  App.RadioControl -> ActiveMessageC;
  App.Leds -> LedsC;

#ifndef ENCRYPTION_ENABLED
  App.AMSend -> RF212ActiveMessageC.AMSend[10];
#else
  App.AMSend -> SecActiveMessageC.AMSend;
  App.SecurityConf -> SecActiveMessageC.SecurityConf;
#endif
}
```

SecReceiverAppP Code

```
#include "Cryptography.h"

module SecReceiverAppP{

  uses{
  interface Leds;
  interface Boot;
  interface SplitControl as RadioControl;
  interface Receive[am_id_t rc_id];
  }
}

implementation{

  event message_t* Receive.receive[am_id_t rc_id](
    message_t* msg, void* payload, uint8_t len){
    call Leds.led1Toggle();
    return msg;
  }

  event void Boot.booted() {
    call RadioControl.start();
  }

  event void RadioControl.startDone(error_t err) {}

  event void RadioControl.stopDone(error_t err) {}
}
```

A.3.3 Alternative Security Features

In addition to the regular CCM* mode according to IEEE 802.15.4 another implementation to add/check MAC and en-/decrypt the message payload are possible. The alternative implementation encrypts the payload in CBC mode and with an IV (initialisation vector). The IV can be specified before compilation in the component `KeyManagerP`. To encrypt the payload in CBC mode it is padded to fit entirely in 16 byte blocks[1] (see figure A.1). The MAC is computed over the whole message, exluding the length and CRC fields. It also uses the CBC mode and requires entire 16 byte blocks. If this is not the case again padding is used.

[1] That is by taking the number of bytes which have to be padded, and padding the end of the payload with this number value until the last 16 byte block is full. Padding in this way is recoverable. A fault arises only if the payload has no partly full block and ends with values corresponding to a correct padding.

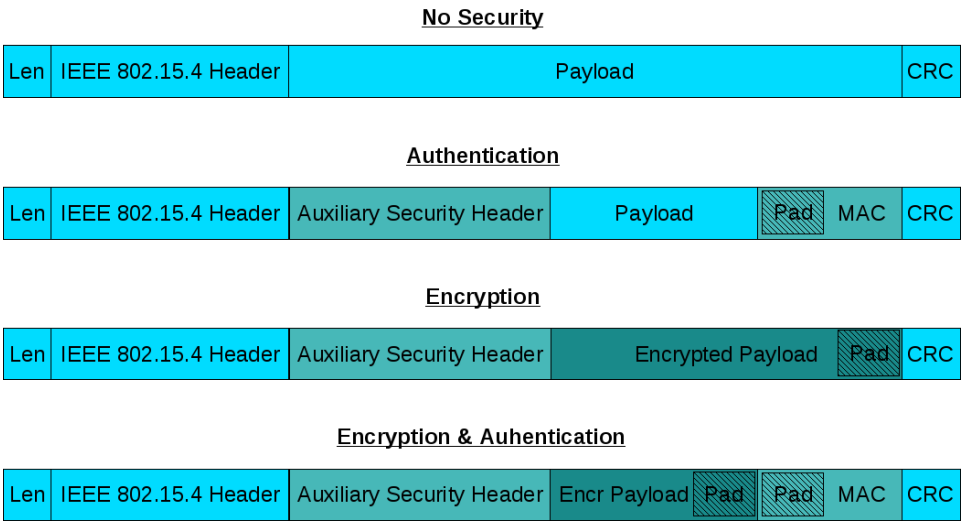

Figure A.1: Frame format of the alternative confidentiality and authentication implementation

To apply the alternative implementation, the driver layer (`RF212DriverLayerP`) has to be changed. One has to replace the following code:

Send

In the transmission part of the driver layer replace

```
length = call Cryptography.CCM(msg, length, keyInfo);
call RF212DriverConfig.setLength(msg, length);
```

with the code

```
// Encryption
if ((keyInfo & SEC_LEVEL_ENC_MASK)==SEC_LEVEL_ENC)
{
  length = call RF212DriverConfig.getLength(msg);
  length = call Cryptography.encrypt(msg, length, keyInfo);
  call RF212DriverConfig.setLength(msg, length);
}

// MAC
if ((keyInfo & SEC_LEVEL_MAC_MASK)!=SEC_LEVEL_NO)
{
  length = call RF212DriverConfig.getLength(msg);
  length = call Cryptography.addMAC(msg, length, keyInfo);
  call RF212DriverConfig.setLength(msg, length);
}
```

Receive

In the receive part of the driver layer replace

```
length = call RF212DriverConfig.getLength(rxMsg);
length = call Cryptography.inverseCCM(rxMsg, length, keyInfo);
call RF212DriverConfig.setLength(rxMsg, length);
```

with the code

```
// MAC
if ((keyInfo & SEC_LEVEL_MAC_MASK)!=SEC_LEVEL_NO)
{
  length = call RF212DriverConfig.getLength(rxMsg);
  length = call Cryptography.checkMAC(rxMsg, length, keyInfo);
  if (length == 0)
  {
    crc = 0;
    printf("False Mac\n");
    length = call RF212DriverConfig.getLength(rxMsg);
    call RF212DriverConfig.setLength(rxMsg, length);
  }
  else call RF212DriverConfig.setLength(rxMsg, length);
}

// Decryption
if ((keyInfo & SEC_LEVEL_ENC_MASK)==SEC_LEVEL_ENC)
{
  length = call RF212DriverConfig.getLength(rxMsg);
  length = call Cryptography.decrypt(rxMsg, length, keyInfo);
  call RF212DriverConfig.setLength(rxMsg, length);
}
```

Further Alternatives

The component `Cryptography` provides additional functionalities which are self-explanatory or explained in the source code itself.

Bibliography

[1] *AT86RF212 datasheet. http://www.atmel.com/dyn/resources/prod_documents/doc8168.pdf [Online; accessed 16-June-2009].*

[2] *CC2420 datasheet. http://enaweb.eng.yale.edu/drupal/system/files/CC2420_Data_Sheet_1_4.pdf [Online; accessed 23-June-2009].*

[3] DCG Wiki. *https://dcg-trac.ethz.ch/fs09/wiki/ [Online; accessed 29-June-2009].*

[4] TinyOS Tutorials. *http://docs.tinyos.net/index.php/TinyOS_Tutorials [Online; accessed 24-June-2009].*

[5] SPI - Serial Peripheral Interface, June 2000. *http://www.mct.net/faq/spi.html [Online; accessed 23-June-2009].*

[6] Announcing the ADVANCED ENCRYPTION STANDARD (AES), Nov 2001. *http://merlot.usc.edu/csac-s06/papers/fips-197.pdf [Online; accessed 23-June-2009].*

[7] nesC: A Programming Language for Deeply Networked Systems, Dec 2004. *http://nescc.sourceforge.net/ [Online; accessed 24-June-2009].*

[8] IEEE Standard for Information technology- Telecommunications and information exchange between systems- Local and metropolitan area networks- Specific requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs). *IEEE Std 802.15.4-2006 (Revision of IEEE Std 802.15.4-2003)*, pages 0–305, 2006.

[9] Mote-mote radio communication, Sept 2008. *http://docs.tinyos.net/index.php/Mote-mote_radio_communication [Online; accessed 16-June-2009].*

[10] Meshbean900, 2009. *http://www.meshnetics.com/dev-tools/meshbean/ [Online; accessed 23-June-2009].*

[11] ZigBit 900 Module with Balanced RF Output, 2009. *http://www.meshnetics.com/zigbee-modules/zigbit900/ [Online; accessed 23-June-2009].*

[12] David Culler, Deborah Estrin, and Mani Srivastava. Guest Editors' Introduction: Overview of Sensor Networks. *IEEE Computer Science*, 2004.

[13] Carl Hartung, James Balasalle, and Richard Han. Node Compromise in Sensor Networks: The Need for Secure Systems. Technical report, University of Colorado, Boulder, 2005.

[14] Karlof, Chris, Sastry, Naveen, Wagner, and David. TinySec: a link layer security architecture for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 162–175, New York, NY, USA, 2004. ACM.

[15] P.Srisuresh, K.Egevang. *Traditional IP Network Address Translator (Traditional NAT).* Network Working Group, January 2001. http://tools.ietf.org/html/rfc3022 [Online; accessed 30-December-2008].

[16] K. Romer and F. Mattern. The Design Space of Wireless Sensor Networks. *Wireless Communications, IEEE*, 11(6):54–61, Dec. 2004.

[17] Stubblefield, Adam, Ioannidis, John, Rubin, and Aviel D. A key recovery attack on the 802.11b wired equivalent privacy protocol (WEP). *ACM Trans. Inf. Syst. Secur.*, 7(2):319–332, 2004.

[18] P. Syverson. A taxonomy of replay attacks [cryptographic protocols]. In *Computer Security Foundations Workshop VII, 1994. CSFW 7. Proceedings*, pages 187–191, Jun 1994.

[19] D. Whiting, R. Housley, and N. Ferguson. *Counter with CBC-MAC (CCM).* Network Working Group, September 2003. *ftp://ftp.rfc-editor.org/in-notes/rfc3610.txt [Online; accessed 16-June-2009].*

[20] Wood, Anthony D., Stankovic, and John A. AMSecure: secure link-layer communication in TinyOS for IEEE 802.15.4-based wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 395–396, New York, NY, USA, 2006. ACM.