Authentication and Information Integrity in Wireless Sensor Networks

Diploma Thesis by Moritz Killat

submitted to Prof. Dr. Hannes Hartenstein University of Karlsruhe, Germany

carried out at NEC Network Laboratories Heidelberg, Germany

Supervisor: Dr. Dirk Westhoff (NEC)
I, Moritz Killat, declare that I have written this thesis independently, solely based on the literature that I have referenced at the end of this document. I have not used any further aid.

Karlsruhe, 31st of March, 2005
Acknowledgment

Many thanks to NEC Europe Ltd. in Heidelberg, Germany, for giving me the opportunity to work on such an interesting topic in a very kind environment. In particular, I want to thank Dr. Dirk Westhoff and João Girão who supported my work in several ways. They helped me with fruitful discussions that often let new ideas cross my mind, with advice for papers that brought forward my work a lot and with all their experience in this technology.
Abstract

Authentication and information integrity in Wireless Sensor Networks is a challenging problem. As sensors are conceived to be used in hundreds or thousands in the near future their costs are required to be extremely cheap. This aim is reflected in their equipment that only allows low computation - and restricted energy power as well as the absence of tamper resistant memory. A protocol providing authentication and information integrity for sensor networks needs to consider these constraints by still providing an appropriate level of security.

In this work we address this challenge and present a protocol applicable to different architectures of networks. In particular, we discuss the problem of compromised nodes, the disclosure of sensitive data coming along with a corruption and we discuss procedures that let the entire network return into a safe state. On the basis of an implementation and by an analytical prove we show that our proposal adheres to the desired level of security while spending less energy.
Contents

1 Introduction 1

2 Problem Statement 5
   2.1 Wireless Sensor Networks ............................... 5
   2.2 Reference Model ........................................ 6
   2.3 Threat Model ........................................... 7
   2.4 Objectives ............................................. 7
      2.4.1 Desirable Capabilities ............................. 8
   2.5 Summary .............................................. 8

3 Scenario 11
   3.1 Service with a pull-Characteristic .................... 12
   3.2 Service with a push-Characteristic .................... 13

4 Related Work 15
   4.1 Public Key Cryptography ............................... 15
   4.2 Distributing Public Key Operations ..................... 15
   4.3 Symmetric Schemes .................................... 16

5 Our Proposal 19
   5.1 Clearly Arranged WSN Supporting a Single Service .... 19
      5.1.1 Hash Chains and the TESLA Protocol ............... 20
      5.1.2 Preconditions ..................................... 21
      5.1.3 Procedure ....................................... 22
   5.2 Clearly Arranged WSN Supporting Several Services ..... 23
      5.2.1 Preconditions ..................................... 23
      5.2.2 Procedure ....................................... 24
   5.3 Large WSNs ............................................ 24
      5.3.1 Preconditions ..................................... 24
      5.3.2 Procedure ....................................... 25
   5.4 Communication from the Sensors to the Aggregator(s) ... 25
      5.4.1 Preconditions ..................................... 25
      5.4.2 Procedure ....................................... 26

6 Compromised Nodes 27
   6.1 Password Substitution for Small WSNs ................. 27
      6.1.1 Preconditions ..................................... 27
      6.1.2 Procedure ....................................... 28
   6.2 Password Substitution for Large WSNs .................. 29
      6.2.1 Preconditions ..................................... 29
      6.2.2 Procedure ....................................... 30
      6.2.3 Extending the Sensor Network ..................... 32
7 Implementation

7.1 TinyOS and nesC ........................................... 35
7.2 Implemented Modules ........................................ 37
  7.2.1 PacketDifferentiatorM ................................. 38
  7.2.2 SREQM .................................................. 39
  7.2.3 Sink Implementation ............................... 40
7.3 Measurements ................................................. 41

8 Analysis of the Proposal ...................................... 43

8.1 Energy Model .................................................. 44
  8.1.1 Energy Map of Our Proposal .......................... 46
  8.1.2 Energy Map of TESLA ................................ 47
  8.1.3 Energy Map of Public Key Cryptography ............. 48
  8.1.4 Conclusion ................................................. 52
8.2 Security Model ................................................. 55
  8.2.1 Formal Evaluation of the Proposal ..................... 56
  8.2.2 Security of the Transmission Channel ............... 59
  8.2.3 Security of the Sensors ............................... 62

9 Summary and Outlook .......................................... 65

References ......................................................... 67

A Installation Instructions ....................................... 69

B Source Code of PacketDifferentiatorM .......................... 73

C Source Code of SREQM .......................................... 85

D Source Code of Sink .............................................. 87
  D.1 Java Implementation ..................................... 87
  D.2 nesC Implementation ..................................... 95
LIST OF FIGURES

List of Figures

2.1 Wireless Sensor Network as a cone .......................... 7
3.1 WSN in the DAIDALOS project ................................. 11
3.2 Service with a pull-characteristic ............................. 12
3.3 Immediate Response of the Web-Service ...................... 13
3.4 Service with a push-characteristic ............................ 13
5.1 TESLA Protocol ............................................... 20
5.2 Relation between keys and time frames in TESLA ............... 21
6.1 Password generation for sensor $S_j$ .......................... 28
7.1 Survey of the Modules ........................................ 38
7.2 Deriving a hash function from a symmetric cipher ............... 42
8.1 Energy Consumption – Our Proposal vs. RSA .................. 54
8.2 Energy Consumption – Our Proposal vs. TESLA ................ 55
8.3 Impact of the usage of TESLA ................................. 56

List of Tables

8.1 Authentication scheme for large WSN ......................... 44
8.2 Energy Scheme for the Protocol described in Table 8.1 ........ 46
8.3 Schema of TESLA ............................................ 47
8.4 Energy Scheme for TESLA .................................... 48
8.5 Public Key Cryptographic in both directions .................. 50
8.6 Energy Scheme for the Protocol described in Table 8.5 ........ 51
8.7 Public Key Cryptographic in one direction .................... 52
8.8 Energy Scheme for the Protocol described in Table 8.7 ........ 53

Listings

7.1 Abstract Module .............................................. 35
7.2 Abstract Configuration ....................................... 36
7.3 Abstract Interface ........................................... 37
A.1 Makefile .................................................. 69
A.2 Sensor Installation-Script ................................... 69
A.3 Java Run-Script ............................................... 70
B.4 PacketDifferentiatorM.nc .................................... 73
C.5 SREQM.nc .................................................. 85
D.6 Sink.java .................................................. 87
D.7 SinkM.nc .................................................. 95
1. Introduction

Authentication is an expression that is used, maybe even abused, in a very broad sense. By any means, it aims to guarantee that an entity is who it claim to be or that information has not been altered by unauthorized parties before reaching the recipient. Both terms must be satisfied to achieve message authentication with which we deal in this work. Message authentication demands that a party $B$ receiving a message is assured of the identity of the party $A$ which originated the message [21]. This also includes information integrity, i.e. the assurance that a message has not been manipulated, as in case of a spoofed message $A$ would no longer be the originator of the message. In this work we conceive authentication as message authentication. If we solely address the problem of assuring a party of another’s identity, we use the term source authentication.

Authentication in wireless sensor networks (WSN) is a challenging problem. Some well known mechanisms to solve authentication in general are not applicable as they require costly computations which is incompatible to the restricted capabilities of a sensor in terms of computation power. Several works in the literature addressed the problem of low-cost authentication and presented procedures that are solely based on power-saving operations. These solutions work fine under specific constraints but have some significant drawbacks if the application does not meet these constraints. This work discusses the problem of low-cost authentication for Wireless Sensor Networks. A protocol is presented that, depending on the structure of the WSN and depending on the requirements defined by the application, presents an efficient solution. We distinguish and discuss this problem for three different kinds of WSN: (a) a small WSN that supports a single service, (b) a small WSN that supports several services and (c) large WSNs that, additionally, are able to address specific regions of the whole network. The first two, (a) and (b), are conceivable for different services that improve our daily life. The security level of driving, for example, would increase if a vehicle gets informed on the status of the street in front of critical areas as bridges. Contrariwise, the latter scenario (c) aims at monitoring larger areas as, for example, forests to detect fire or to observe wildlife animals’ movement.

We assume a similar structure for both kind of WSN: a central node, called sink, is either linked to a fixed network or represents the end-point of
the data collecting process. It receives the measured values from the sensors, evaluates them and updates the corresponding service(s) in the network. Depending on the scenario the WSN consists of a variable amount of sensors. On a command sent out by the sink node, these sensors start measuring their environment and transmit the values back to the sink node. To prevent the sensors of being flooded by these messages, one or more aggregation nodes (depending on the scenario) collect the messages and forward a cumulative result to the sink. Recapitulating, the needed protocol has to protect two kinds of communication: a multicast communication from the sink to the sensors and an inverse multicast communication, that is often denoted as concat or reverse multicast communication in the literature, back from the sensors to the sink, respectively to the aggregator(s). More precisely, each message sent out by the sink node destined for the sensors needs a source authentication-, an integrity- and a replay protection. Accordingly, each aggregator has to check the same mechanisms for each received message originated by a sensor whose data it aggregates.

In contrast to the sink node, which is linked to a fixed network and therefore possesses virtually unlimited energy and computation power, the capabilities of the sensors and the aggregators are strictly restricted. In this work we refer to the Mica2-platform which limits the sensor’s processor speed to 4MH\text{z} and its memory to 4kB. Furthermore, the sensors’ overall energy is tightly restricted. Moreover, due to the used operating system on the sensors (TinyOS), respectively due to the radio frequency module that is used, the payload of each message is upper-bounded to 29 bytes. We assume that all nodes are supplied by the same manufacturer and, hence, can be quipped with initial information in pre-deployment.

With respect to the threat model, we assume, that an adversary has a multitude of possibilities for an attack on the network: he can read, forge and intercept the messages transmitted at an arbitrary point in time. He can break into a sensor and read out its memory as the sensors, due to costs, are not equipped with tamper resistance memory. Ultimately, the attacker possesses unlimited energy and computation power.

The contribution of this work is the presentation of an authentication protocol that was developed by having the strict constraints of Wireless Sensor Networks and the varying claims of different applications in mind. Our considerations are not limited to the hardware restrictions that come along with a sensor but also to the multitude of attacks that an adversary can apply. Finally, we demonstrate the efficiency of our protocol concerning energy by an analytical comparison to other approaches.

This work is organized as follows. In Chapter 2 we concretize the problem: we present a general overview over Wireless Sensor Networks, give detailed informations on the capabilities of the sensors and, finally, specify the problem which underlies this work. In the following Chapter 3, we describe an application that makes use of WSNs: we distinguish different
access characteristics that, depending on the application, are conceived to be used. After an overview over previous authentication protocols in the literature in Chapter 4, we present our own proposal for three different kind of networks in Chapter 5. Chapter 6 addresses the particular problem of a sensor’s corruption in WSNs and the presentation of means that let the network return to a safe state. Chapter 7 deals with our implementation. We give a short introduction to the used programming language nesC and outline some specific components of the code. Finally in Chapter 8, we analyze our proposal concerning the achievable level of security and, in comparison to previous protocols, the amount of energy spent for each authentication. In Chapter 9, we complete this work with a final overview over the achieved work and have a outlook to further desirable steps.
1 INTRODUCTION
2. Problem Statement

In this Chapter we point out the aim of this work. First we outline a short overview over Wireless Sensor Networks, the involved components, their capabilities and constraints. This description is followed by a short presentation of the configuration we are using. We refer to the components' positions in the network and to the communication that occurs between the components. After having sketched a sufficient picture of the network, we formulate the problem itself.

2.1 Wireless Sensor Networks

Within the very close past the range of computer networks was extended through a new class of computers: wireless, battery powered sensors. Each of them is equipped with a low cost processor, a sensing and a radio module. With further improvements especially in terms of costs they are expected to be used in hundreds or thousands within one network in the near future [15]. In contrast to conventional networks, Wireless Sensor Networks (WSNs) especially have to focus on power saving operations and should therefore minimize the usage of the radio interface. The authors of [16] identified the power consumption required for transmitting a single bit to be equivalent to 800 instructions executed on the sensor itself.

WSNs often consist of three different components: a powerful sink node \( R \), the low-powered aggregators \( A_l \) and of sensors \( S_j \). An aggregator aggregates sensed data that it has received from sensors. More precisely, on the one hand it reduces the traffic in the network by computing and forwarding a cumulative value that still keeps an appropriate average information content and on the other hand, as a consequence, it prevents the sink node from being flooded with messages. The sink node is distinguished from the others by its position in the network, by its capabilities concerning computation power and amount of memory and by its possibilities of getting compromised. We assume the sink to possess exclusively tamper resistance memory which prevents attackers from gaining any informations by corrupting the sink. Ideally, the sink node has a central position in the network itself and possesses a connection to a fixed network to which the WSN is linked to.
Contrariwise, the aggregator(s) and the sensors are only equipped with little computation power and a small amount of memory that is, particularly, non-tamper-resistant. Our work concentrates on the Mica2-motes that are under development at UC Berkeley (see [16]). They are equipped with a 4MHz Atmel microprocessor, with 4 kB of RAM and 128 kB of code space. The radio frequency module is sending in the ISM band at the center frequency of 917 MHz with a rate of 38 kBaud/s using Manchester Encoding. By supplying power with two AA batteries, the authors of [16] estimated an overall lifetime of about 2 months for the sensors if they are continuously sending one message per second. On this Mica2-hardware architecture we are using the TinyOS-operating system. Our choice to use this proprietary operating system instead of the to-be IEEE 802.15.4-standard solely depended on the sensors that were at ours disposal. The module in TinyOS, responsible for controlling the radio frequency, limits the maximal payload per packet to 29 Bytes. However, as the future IEEE-standard is still in discussion, it is not clear up to now whether the standard deviates from this restriction.

2.2 Reference Model

We assume an alignment of the components as follows. As proposed in the previous sub chapter, the sink node is placed in a central position. This may be the center of a circle or the apex of a cone (see Figure 2.1, arrows describe the way of communication). Such an alignment is not binding but the proposed solution achieves the best performance under these presumptions. The sensors have the longest distance to the sink node. Between them, one or more aggregators collect the sensors’ data and forward a cumulative result to the sink node. Depending on the size of the network several steps of aggregators are conceivable, i.e. we are dealing with a kind of hierarchy of aggregators in which one node aggregates or at least forwards the results of aggregators from a lower level to the sink node (see Figure 2.1, \(A_2\), \(A_3\) and \(A_4\)).

Basically, we have to distinguish two kinds of communication in the network: a multicast communication from the sink node to the sensors and a concast, an inverse multicast, communication, as shown in Figure 2.1, back from the sensors to the sink node. The multicast communication is usually used by the sink for polling the sensors periodically. On receipt of such a message, the sensors start their computations and send back their values to the aggregators. These aggregate or just forward the values until the messages reach the sink. In exceptional circumstances, however, the sensors have the additional possibility of sending aperiodically trap-messages.
2.3 Threat Model

We feel that a restriction of an adversary’s capabilities anyhow does not match realistic demands. Thus, we concede virtually unlimited energy and computation power as well as a multitude of possibilities to an attacker: he is able to read, forge and intercept messages at an arbitrary point in time. Furthermore, as the sensors are not equipped with tamper resistant memory he can break a sensor and read out its memory in order to gain sensitive data.

Beside the interest in disturbing the service provided by the network by manipulating data, we assume the attacker to have an incentive to turn off the network as soon as possible. This aim can either be achieved by destroying several sensors or by exhausting the limited energy capacity of the sensors. While the former cannot be prevented by software means, we take the latter into our considerations.

2.4 Objectives

The aim of our work is the presentation of an authentication protocol that matches the significant constraints of a Wireless Sensor Network. Therefore, we will not only focus on achieving an appropriate level of security but we will also take the consumption of energy into account to achieve a maximal
lifetime of the whole network. In other words, we have to simultaneously optimize in two dimensions: security and energy consumption which, as we will see later, are not independent to each other.

Concerning the security issue, on the one hand we have to take care of the communication over the untrusted air channel. Both described communication directions require the same protection aims:

- **Source Authentication**, to prevent an adversary either from requesting the sensors to start their computations or from impersonating a sensor and sending arbitrary measured values to an aggregator.

- **Replay Protection**, to prevent an adversary from retransmitting any previously used message in order to cause the same activity again.

- **Integrity Protection**, to prevent an adversary either from altering a command originated by the sink node or from forging any measured or aggregated value.

On the other hand, however, in contrast to conventional networks, we have to consider that sensors do not possess tamper resistant memory. Consequently, the corruption of a sensor could reveal sensitive data that jeopardizes the security of the whole network.

### 2.4.1 Desirable Capabilities

A special interest lies in the behavior of the security mechanisms towards changes in the network’s structure itself. How does the network react on nodes that are, all of a sudden, not working anymore? Is there a way to securely integrate further nodes to the network at a later point in time? Desirable is the automatic detection and a following self-reconfiguration without the intervention of the manufacturer.

### 2.5 Summary

We presuppose the following constraints for the components.

<table>
<thead>
<tr>
<th></th>
<th>Sink</th>
<th>Aggregator</th>
<th>Sensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation Power</td>
<td>unrestricted</td>
<td>4 MHz</td>
<td>4 MHz</td>
</tr>
<tr>
<td>RAM</td>
<td>unrestricted</td>
<td>4 kB</td>
<td>4 kB</td>
</tr>
<tr>
<td>code space</td>
<td>unrestricted</td>
<td>128 kB</td>
<td>128 kB</td>
</tr>
<tr>
<td>tamper-resistant Memory</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>availability of Energy</td>
<td>unrestricted</td>
<td>restricted</td>
<td>restricted</td>
</tr>
<tr>
<td>Transmitting Range</td>
<td>undefined</td>
<td>restricted</td>
<td>restricted</td>
</tr>
<tr>
<td>compromiseable</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

The communication directions require the following general conditions.
2.5 Summary

<table>
<thead>
<tr>
<th></th>
<th>$R \rightarrow {A_i, S_j}$</th>
<th>$S_j \rightarrow A_i$</th>
<th>$A_i \rightarrow A_m$</th>
<th>$A_m \rightarrow R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>max Payload</td>
<td>29 Bytes</td>
<td>29 Bytes</td>
<td>29 Bytes</td>
<td>29 Bytes</td>
</tr>
<tr>
<td>Authentication</td>
<td>required</td>
<td>required</td>
<td>required</td>
<td>required</td>
</tr>
<tr>
<td>Replay Protection</td>
<td>required</td>
<td>required</td>
<td>required</td>
<td>required</td>
</tr>
<tr>
<td>Integrity</td>
<td>required</td>
<td>required</td>
<td>required</td>
<td>required</td>
</tr>
</tbody>
</table>
3. Scenario

In the introduction we have already mentioned some scenarios that require the application of Wireless Sensor Networks. In the following we take a closer look to a scenario that is borrowed from the EU project DAIDALOS in which context this work has been carried out. In this regard, sensor networks are conceived to observe conditions of a street like temperature in order to ease, respectively to raise the security of a car drive.

Figure 3.1 depicts the usage of WSNs in the DAIDALOS project: specific points in the street or in its surroundings are inspected and, finally, worked up in order to provide a service to the participants. The application of the services is conceivable in different ways. Within a planning phase, for example, a navigation system could take the information gained by these services into account in order to compute a route that, exemplarily, avoids streets currently suffering from frost or from aqua-planning. The security of actual driving itself would benefit, if the driver gets warned of traffic jams in front of hardly visible passages of the street that frequently suffer from a lot of traffic. Similarly, parts of a road that are often crossed by animals could be defused by “attention-messages” triggered by the detection of movement in the close surrounding of the street. As the examples have already shown, different access-characteristics to the services are imaginable: a poll-characteristic triggered by participants for synchronous status reports and a push-characteristic that asynchronously informs service subscribers about exceptional circumstances.

Figure 3.1: WSN in the DAIDALOS project
The drivers get access to these services through a well-known service access point in the Internet. From time to time, this service access point demands its connected WSNs, respectively the sinks of these networks which act as a proxy, to update their previous measurements. On receipt of such a command the sink lets the sensors in its network start their computations and, finally, sends back an updated value to the service access point. For any reasons an adversary could have an interest in disturbing this service and be it that it is just a competitor of the service provider. Therefore, the described traffic needs at least authentication, perhaps even a confidentiality protection. While our proposed solution deals with the traffic between the sink and the sensors, other conventional mechanisms are conceivable between the sink and the service access point. Both do not suffer from such critical constraints as the sensors do and, hence, can use typical solutions as from the area of Public Key Cryptography.

3.1 Service with a pull-Characteristic

![Diagram](image)

Figure 3.2: Service with a pull-characteristic

The diagram shown in Figure 3.2 depicts the usage of a pull-service. The driver, called Bart, accesses the Web-Service through a well defined access point specifying the desired service (e.g. a service that is offered through a specific WSN in front of a particular bridge). The Web-Service demands the
corresponding WSN through a fixed connection to update its measurement (shown in Figure 3.2) or, in case of a sufficient up-to-date stored value, answers immediately (illustrated in the Service-Request initiated by Minnie in Figure 3.3).

![Figure 3.3: Immediate Response of the Web-Service](image)

### 3.2 Service with a push-Characteristic

Once subscribed to a service, the driver can get informed asynchronously which we consider to be a push-service. Cautioning a driver about an animal
suddenly crossing the street, for example, falls in this category. Figure 3.4 visualizes this service. Triggered by an event that was detected by a sensor, an emergency message is forwarded towards the sink, maybe passing some aggregators. Depending on an internal decision on the importance of the event, it is forwarded to the Web-Service and finally to the participants.
4. RELATED WORK

This Chapter is organized as follows. We start with a short discussion on the suitability of public key mechanisms for our scenario as they appropriately solve the problem of authentication in conventional environments. In the following we will solely discuss proposals that address the multicast communication from the sink to the sensors, as to our best knowledge, an \( n:1 \)-authentication as needed in our scenario hasn’t been an issue in the literature so far. Some schemes were proposed which deal with symmetric key negotiation that may be used for hop-by-hop authentication but an individual authentication between each sensor and the sink hasn’t been discussed so far. Particularly, this authentication direction is a challenging problem, as each secret used to recognize a specific node may be not sufficiently secured due to the lack of tamper resistant memory.

4.1 Public Key Cryptography

Well known public key mechanisms that satisfactorily solve the problem of authentication are hardly applicable due to exhausting computations and data storage (for a further discussion we refer to Chapter 8). Some research addressed this problem and presented public key mechanisms that spend less energy as for example Rabin’s public key scheme [3] or NTRU Public Key Cryptosystem [4]. Both require a much lower power consumption than conventional public key mechanisms but still go beyond the scope of a sensor’s restriction. Based on these procedures [1] proposed a hardware assisted approach. The modification of hardware mostly accelerates the performance of intensive operations and saves energy, as well. But with such a modification, however, additional costs come along that may go beyond the scope. For this reason, we don’t take the proposal made in [1] into considerations.

4.2 Distributing Public Key Operations

Gennaro and Rohatgi suggested to distribute the costs of a common public key computation to several packets [2]. The first of these packets is signed using a public key operation. As each of the packets contains a hash value of its successor, every following packet is secured as well. Hence, the cost of a single public key operation is amortized by its distribution to the length of
such a chain. Unfortunately, this proposal demands the knowledge of each packet in a chain in advance and does not tolerate packet losses as the chain would be interrupted which results in the loss of authentication. As packet losses are not unusual in wireless communication and the pre-knowledge of packets does not match a WSN characteristic, this proposal is not suitable for our application.

Further considerations by Perrig, Canetti, Tygar and Song addressed these problems. They proposed the EMSS mechanism that is still based on the idea of distributing the costs of a public key operation to a multitude of packets but does not depend on the pre-knowledge of a stream anymore [5]. To overcome the problem of lost packets, every packet contains a set of hash values of previous packets. The last packet of a stream is signed using a public key operations. Therefore, the receiver of such a stream needs to store all previous packets before receiving the signature packet to be assured about their validity. Two problems come along with this proposal. Firstly, the storage of previous packets requires costly memory that may not be available on a sensor. Secondly, the additional transmission of a set of hash values which is attached to each packet is not applicable due to the limited size of a packet to 29 Bytes of payload.

4.3 Symmetric Schemes

In spite of the distribution, all these proposals still suffer from the expensive usage of public key operations. Instead, Canetti et al. proposed an approach that is based on the usage of symmetric keys [6]. As an adversary could easily get knowledge of these keys by cracking a single sensor and, hence, could compromise the whole network, the keys have to be distributed in a secure way. Therefore each sensor possesses a subset of all keys. Moreover, some Message Authentication Codes (MAC) using different keys are attached to each message sent out by the sink node. Each sensor receiving such a message verifies whether the MACs corresponding to the keys the sensor owns are valid. If all of them are, the sensor infers the message’s validity. Again, regarding our constraints this proposal fails because it would need a too large packet size due to the multitude of MACs.

Perrig, Canetti et al. proposed the often quoted low cost TESLA authentication protocol [5]. TESLA solely uses one MAC attached to each packet which is generated by using continuously new keys which are published at a later point in time. On receipt of such a message, a recipient stores the packet, waits for the revelation of the key and checks its validity. If this verification fails, the packet is discarded. To prevent an adversary from using its own keys, each key belongs to a specific hash chain generated by the sender (see [8]). Initially, each receiver obtains the last value of the chain and can therefore infer the validity of the key. Further, the misuse of an already published key is avoided by linking a key to a specific span in
4.3 Symmetric Schemes

If a node receives a message whose MAC was generated by the usage of an already expired key, it discards the packet as well. Beside the necessity of loose time synchronization between the sender and the receiver which requires additional effort, TESLA is well optimized for data streams as a key can be revealed in succeeding packets (for further details, see Chapter 5.1.1). In our case, however, the communication characteristic differs from a stream as we assume a characteristic similar to polling. Hence, the transmission of an additional packet is necessary to reveal the just applied key.

In [9] the authors presented a low cost re-recognition-mechanism, called Zero Common-Knowledge (ZCK). Re-recognition differs from authentication as it assures two communicating parties of having already communicated with each other before but not of communicating with a specific identity. More precisely, having insecurely exchanged some informations at an arbitrary point in time, both parties can prove each other in the following message interchange of being the party with whom the initial data were once exchanged. Let us look to the problem more in detail. Alice and Bob both exchange at an arbitrary point in time the last element of an own created hash chain. In the following Alice can transmit messages to Bob that are authenticated by a MAC using the last unrevealed element of Alice’s hash chain. Bob answers with his last unrevealed element and therefore assures Alice of having received the message. In the last step, after having received and verified Bob’s element of the hash chain, Alice publishes the element used in the MAC which lets Bob verify the authentication protection of the message. As you cannot exclude an adversary’s participation in the initial exchange process of the hash chain values, you cannot infer the identity of your counterpart. We will later see that a sensor network’s capabilities allows us to confine to re-recognition proceedings. For the most general case Zero Common Knowledge is proven to accomplish this aim with a minimal number of messages. Anyhow, as our focus is solely directed to WSNs, we expect to reduce this number by exploiting specific constraints characterizing these networks.

The authors of [9] have also proposed an extension to ZCK, called Identity Certified Authentication (IC), that overcomes the initial authentication problem. We are not going to discuss this protocol for the following two reasons: (a) we assume having the possibility of equipping each node in pre-deployment with several data. Therefore, each node holding another’s hash chain value for the ZCK-protocol can trust in its reliability, respectively in the credibility of its counterpart. (b) As we have already criticized the amount of traffic in ZCK and are looking for solutions that reduce this overhead, it is senseless to go through an extension.
5. Our Proposal

In this Chapter we present a protocol that guarantees a multicast-authentication from the sink node to the sensors and a concast-authentication from the sensors back to the sink. Concerning to our application, this protocol minimizes the amount of messages needed for each authentication process while keeping an adequate level of security. Our proposal is applicable to three different architectures of WSN. These build up on each other and, hence, our presentation starts with the simplest case and ends with the most complex one. All of these variants share the same communication principle back to the sink, respectively to the aggregator(s). This direction is discussed at the end of this Chapter but as it is also important for the protocol used from the sink node to the sensors, we will often refer to the inverse direction before.

5.1 Clearly Arranged WSN Supporting a Single Service

We conceive a “clearly arranged WSN” as a small WSN that contains at most of one aggregation node. We assume the issuing of a command to only a part of the whole network as senseless. As the network solely supports one service, every sensor receiving a command which demands it to measure its environment needs simply to be assured, that the sink node sent out this message and not an adversary. More precisely, it is only necessary to assure the receiver about the originator of an arbitrary message. As we are only dealing with one, parameterless command, the message does not need an integrity protection. Source authentication, however, is required as otherwise an adversary could exhaust the network’s energy by continuously sending orders to the sensors.

Our proposal for the communication from the sink to the sensors mainly bases upon one way hash chain – also known as Lamport’s hash chain [21]. Therefore, we firstly give a short introduction on this mechanism and its influence on the TESLA protocol before discussing its usage in our application.
S → R: \( m, hash(m \| hash^k(X)) \)

wait until time frame for \( hash^k(X) \) expires

S → R: \( hash^k(X), k \)

\( hash^k(X) \) denotes the last unrevealed hash chain kept by the sender \( S \). As each receiver \( R \) possesses a verified value \( hash^n(X) \) with \( n > k \) it can conclude the validity of the used key \( hash^k(X) \).

Figure 5.1: TESLA Protocol

5.1.1 Hash Chains and the TESLA Protocol

Lamport’s hash chain [21] is a simple cryptographic instrument that is applied to a multitude of cryptographic protocols. Having once chosen a secret \( X \), the generator of the chain applies a one way hash function (e.g. SHA-1), in the following denoted by \( hash \), \( n \)-times to \( X \). The last element of this chain, \( hash(hash^{n-1}(X)) \), becomes publicly known. As each previous element is protected by the security of the used one way hash function, the generator of the chain has exclusively the knowledge of all unrevealed elements. Furthermore, any holder of the chain’s last element is able to verify a value, \( hash^k(X) \) with \( k < n \), of being part of the chain by applying the hash function \((n - k)\)-times and comparing the result to the known last element.

A common practice of one way hash chains is their application in authentication procedures as, for example, the TESLA protocol. The Time Efficient Stream Loss-tolerant Authentication (TESLA)-protocol provides low-cost authentication for data streams (depicted in Figure 5.1). Having initially distributed the last element of a one way hash chain to all recipients \( R \), the sender \( S \) uses consecutively the elements of the chain as symmetric keys that flow into the computation of Message Authentication Codes (MACs). After having received a packet, each recipient stores the packet and waits for the disclosure of the used key to verify the MAC. In order to prevent an adversary of reusing an already revealed key, each of them is linked to a specific time frame. Hence, each recipient needs to wait at least for the expiration of the corresponding time frame. In Figure 5.2, for example, packet \( p_j \) is secured by the key \( k_i \) that is disclosed by packets sent during interval \( k_{i+2} \). Accordingly, the packets \( p_{j+3} \) and \( p_{j+4} \) both use the key \( k_{i+4} \) that is revealed within the time span \( k_{i+6} \). On receipt of a key \( (hash^k(X) \) in Figure 5.1) \( R \) audits the validity of the key and checks whether the key belongs to the sender’s hash chain. If one of these conditions do not hold, the receiver discards the packet. To achieve a common understanding of time, respectively of valid time frames, between sender and
5.1 Clearly Arranged WSN Supporting a Single Service

Figure 5.2: Relation between keys and time frames in TESLA

receivers, TESLA requires both parties to be loosely synchronized.

Due to the particular characteristic of Wireless Sensor Network, our proposal does not deal with the general problem of authentication and may, therefore, save the necessity of a second dimension to authenticate a message which happens to be the time used in TESLA.

TESLA provides efficient authentication of one sender to, potentially, a multitude of receivers (1 : n) as each of them solely requires the last element of the sender’s hash chain. Furthermore the corruption of a single receiver does not effect the remaining nodes. The inverse authentication (n : 1), however, i.e. to authenticate a set of senders to one receiver, is not in line with the optimization goal of TESLA as n last elements of hash chains must be deposited at the receiver without gaining any advantage over the exchange of symmetric keys, for example; moreover, symmetric keys would additionally provide confidentiality. Therefore, we will take TESLA only into our considerations when discussing the communication from the sink to the sensors.

Now we turn to our proposal.

5.1.2 Preconditions

The sink generates a one way hash chain in pre-deployment and configures each sensor with the chain’s last element. The length of the hash chain should be chosen as long that it overcomes the supposed lifetime of the network. If this is not possible further research should focus on a procedure that securely switches to another hash chain. Recapitulating, in pre-deployment each node receives the following values.

- $hash^n(X)$, needed to verify a received value
- $n$, needed in the verification process

Moreover, an individual secret needs to be stored at each sensor to secure the communication from the sensor back to the sink. For further details, see Chapter 5.4.1.
• secret\textsubscript{j}, to secure the traffic to the aggregator

5.1.3 Procedure

Each time the sink node wants the sensors to measure their environment, it sends out a new, i.e. un-revealed, value of the initially generated one way hash chain. The broadcasts the following message:

$$R \rightarrow S_j : \text{hash}^i(X), i$$

\(i\), a decreasing number starting at \((n-1)\), needs to be attached to the packet as each receiver must know, how often it has to apply the hash function itself to verify the value.

The necessary verification of the received values at the nodes distinguishes three different cases.

1. The received authentication value, \(\text{hash}^i(X)\), is lower or equal to the expected one, i.e. the sensor expected a value \(\text{hash}^j(X)\) with \(j \geq i\). The sensor accepts the order, starts measuring its environment and sends the result back to the sink, respectively to the aggregator. Next time, the sensor expects the value \(\text{hash}^{i-1}(X)\). The acceptance of values lower than the expected one is justified by packet losses.

2. The sensor received the value \(\text{hash}^i(X)\) while expecting the value \(\text{hash}^{i-1}(X)\). This circumstance results from two reasons that are explained in the following and that the sensor cannot distinguish. As one of them is ascribed to an attack, the sensor only answers with the low probability \(p\). \(p\) depends on the overall amount \(m\) of sensors in the network and is defined in pre-deployment. Let \(P_R\) denote the probability with which the sink receives any message at all. Assuming all sensors receive the command sent out by the sink, \(p\) needs to be chosen as

$$P_R \leq 1 - (1 - p)^m$$

$$p \geq 1 - \sqrt[1 - P_R]{1}$$

to achieve the probability \(P_R\).

The two alternatives mentioned above, that the sensor cannot distinguish, are:

(a) It is a replay attack. The sensor should better discard the packet.

(b) Every answer sent back by any sensor on the sink node’s direct prior command (i.e. on \(\text{hash}^i(X)\)) has been lost. Consequently the sensors that sent back the results now expect the hash-value \(\text{hash}^{i-1}(X)\). The sink node, contrariwise, didn’t receive any results, believed that no sensor received its command and retransmitted the same message again.
5.2 Clearly Arranged WSN Supporting Several Services

3. The sensor received a value $hash^j(X)$ while expecting $hash^i(X)$ with $j < i - 1$. The sensor discards the message. As the sensor had received a command containing the hash value $hash^{j+1}(X)$ that it executed (otherwise the sensor would not expect the hash value $hash^j(X)$), it can be assured that the received message cannot be ascribed to packet loss and the following retransmission. The sink node would not retransmit a command containing the hash value $hash^i(X)$ if it has already sent out a command using the hash value $hash^{j+1}$. Therefore it has been a replay attack.

5.2 Clearly Arranged WSN Supporting Several Services

WSNs that support a single service can be easily secured by a one way hash chain (see previous Chapter). As the static characteristic of the command is lost in a WSN that support several services the source authentication protection becomes more difficult. As long as the set of commands is still manageable, an individual hash chain for each command is conceivable. But with an increasing grade of flexibility the feasibility of this approach decreases. Therefore we take a different approach. We assume that the sink does not switch the commands very frequently and thus suggest to keep the proposal of a static command with the additional possibility of changing its meaning. Hence, compared to the proposal made in Chapter 5.1 additional costs only seldom accrue.

5.2.1 Preconditions

We suggest to authenticate the sink node’s command still by the usage of a one way hash chain, whereas the change of semantics to above referred is secured by using the TESLA protocol (see Chapter 5.1.1). To save memory usage, we suggest to use one hash chain for both, our proposal and TESLA. A manufacturer should take this into consideration when determining the length of the one way hash chain as its elements would no longer only be used as orders for the sensors. TESLA requires a loose time synchronization between the involved nodes which could require further memory usage. Beside the time synchronization, the sensors also need a command synchronization. More precisely, due to packet losses or to attacks a sensor might not receive a command-switching-message and, hence, is still serving the previous command type. To prevent an aggregation node to take these wrong values sent by this sensor into account an increasing number identifies the command. Together with the assumptions made in Chapter 5.1.2 which are still necessary as this proposal builds up on that approach, the sensors need to store the following values

- $hash^n(X)$, to authenticate a command sent out by the sink and used by TESLA
• \( n \), needed in the verification process

• \textit{command\_no}, an increasing number which identifies the currently executed command

• additional memory to achieve loose time synchronization

Again, an additional individual secret has to be stored at each sensor to secure the communication back to the sink (see 5.4.1).

• \( \text{secret}_j \), to secure the traffic to the aggregator

### 5.2.2 Procedure

Basically, this procedure has the same operations as described in the previous approach (see Chapter 5.1.3). Solely a new command is defined for offering the possibility of changing the meaning of a command.

\[
R \rightarrow S_j : \text{new command, command\_no,} \\
\text{hash(new command || command\_no || hash}^i(X))
\]

\[
R \rightarrow S_j : \text{hash}^i(X), i
\]

According to TESLA, the second message will be sent after the time period for \( \text{hash}^i(X) \) expired.

Based on the assumption that the command is changed quite seldom, the costs of an additional transmitted message for revealing the key can be tolerated.

### 5.3 Large WSNs

Basically, large WSNs differ from the networks we have previously discussed in the amount of sensors and aggregation nodes. Further, their configuration may consist of a hierarchy of aggregators. These networks may be driven as single service or as multiple services networks using the aforementioned mechanisms. Anyhow, we characterize them in having the additional possibility of addressing only parts of the whole network, more precisely, to address each single \textit{cluster}. We define a cluster as a set of sensors using a common aggregator.

#### 5.3.1 Preconditions

An appropriate approach to address a subset of sensors is the usage of group keys. The establishment of these secrets is discussed in Chapter 6.2. In this Chapter we just refer to their usage and, hence, have to take additional memory into account that stores the corresponding group key at each sensor. Depending on the abilities of the network (single or multiple services) the following data need to be stored at each sensor.
5.4 Communication from the Sensors to the Aggregator(s)

- $hash^n(X)$, to authenticate a command sent out by the sink (depending on the scenario, also used by TESLA)
- $n$, needed in the verification process
- $secret_j$, to secure the traffic to the aggregator (see Chapter 5.4.1)
- $g_l$, key of the group $l$
- $l$, group identifier

And, in case of a network supporting multiple services

- $command_no$, an increasing number which identifies the currently executed command
- additional memory to achieve loose time synchronization

5.3.2 Procedure

By sending the following command the sink solely demands the members of group $l$ to start their measurements.

$$R \rightarrow S_j : \text{hash}^i(X), i, l, \text{hash}^{hash^i(X) || g_l}$$

Each sensor receiving such a message checks the validity of the authentication value, $\text{hash}^i(X)$, and in case of a successful verification updates the next expecting one way hash chain value. This operation is also necessary for sensors that are not contained in the addressed group as otherwise an adversary could later spuriously ask them to execute a command. More precisely, the values of the hash chain that have been published during the communication with the group members, could have been collected and now misused by an attacker.

The group members answer the sink node’s command if both hash values got successfully verified.

5.4 Communication from the Sensors to the Aggregator(s)

The mechanisms used for the communication direction from the sink to the sensors do not work for the way back as the message’s content often varies. Therefore, we use Message Authentication Codes (MACs) to guarantee authentication.

5.4.1 Preconditions

For the MAC, each sensor $S_j$ needs a symmetric key that it shares with its aggregator. Therefore additional memory space is required, i.e.
• *secret*ₗ, symmetric key shared with the sensor’s aggregator. In the following, we will often call this value *S*ₗ’s password.

• *j*, the sensor’s identifier

Of course, the aggregator itself also needs to store these values to verify a received message. The establishment of these values depends on the network’s configuration and is discussed in the following Chapter 6.

### 5.4.2 Procedure

On receipt of a verified order, each sensor either answers with its measured value, denoted by *M*, or with an error message (see Chapter 5.1.3). The message transmitted from each sensor *S*ₗ to its aggregator *A*, is defined as follows.

\[ Sₗ \rightarrow A : \text{hash}(M || \text{command\_no} || \text{hash}(X) || \text{secret}ₗ) \]

To prevent an adversary of playing replay attacks, each message’s hash value depends on the authentication value which let the sensor start its work (\(\text{hash}(X)\)). Furthermore, individual authentication is ensured by using each sensor’s password, *secret*ₗ. The linkage to the command which has been executed, denoted by *command\_no*, can be omitted if the network only provides a single service (see 5.2). The error message is build up analogically.

\[ Sₗ \rightarrow A : \text{error, hash(error || \text{command\_no} || \text{hash}(X) || \text{secret}ₗ)} \]

Even with the usage of strong one way hash functions as SHA-1, the size of the message *M* is limited to 9 bytes to let the whole packet fit into a TinyOS-packet. We feel that it is adequate to represent a measured value. The following inter-aggregator- and the aggregator-sink-communication, to let the measurements finally reach the sink, both use the same message type as defined above.

The reliance on authentication for this approach depends (a) on the chosen hash function and (b) on the concealment of each sensor’s secret, *secret*ₗ. While the security of the chosen hash function can be adjusted depending on the scenario in which it is used, the concealment of the secrets is critical as the tampering of an aggregation node would reveal some secrets. As we cannot prevent an attacker from compromising a sensor, a procedure to return into a secure state is necessary. This work does not deal with the problem of detecting a compromised aggregator but with the recovery of a safe state which is discussed in the following Chapter.
6. Compromised Nodes

The tampering of a node is problematic, as the attacker can read out the node’s memory and gets knowledge of the sensitive data. We assume that it is not possible to tamper the sink node due to its possession of tamper resistant memory. A compromised sensor does not affect the network’s overall security, as the corresponding aggregation node could simply ignore messages sent by this node. Contrariwise, a compromised aggregator reveals the sensors’ keys, whose data it aggregated. In the following we first describe an efficient mechanism to overcome this problem for clearly arranged WSNs, and second, present a protocol that lets a large WSN return to a safe state after the corruption of an aggregator.

6.1 Password Substitution for Small WSNs

This proposal is divided into two steps: on the one hand, the presentation of an efficient mechanism to let the sensors change their passwords in a secure way and on the other hand, the transmission of the new passwords to a newly chosen aggregator. First of all, nevertheless, some preconditions have to be fulfilled to achieve this mechanisms.

6.1.1 Preconditions

Our proposal suggests to equip each node with the whole set of possible passwords in pre-deployment. Depending on the output length of the hash function that is used in the scenario and on the number of possible passwords, an adequate amount of memory is needed at each sensor. Furthermore, each possible aggregator \( A \), i.e. each sensor, shares an individual key \( K_{R,A} \) with the sink to secure the substitution, respectively the transmission, of new passwords. Consequently the following values have to be additionally stored at each sensor \( j \).

- \( \text{number of passwords} \times \text{hash output length} \) (passwords denoted by \( p_j^i \))
- \( K_{R,A} \), individual key shared between each aggregator and the sink to secure the process of password substitution
At the sink’s side, each of the passwords has to be stored as well. Moreover, the sink needs to store a set of keys, denoted by $k_i$, that is used in the password generation process and exclusively known to itself.

### 6.1.2 Procedure

This work does not discuss the problem of detecting a compromised node as, to our opinion, this significantly depends on the scenario in which the network is used. Further research should address this issue. We will solely work on the result of such a detection and offer a procedure to let the network return into a safe state. On suspicion of a revealed password, the sink demands each node to change its password. As an adversary could easily exhaust the set of passwords by sending “password-changing-messages”, the switching process needs to be protected. We suggest to generate each sensor’s set of passwords in a similar way. Starting with an individual secret $p_{0j}$ for each sensor $S_j$, every succeeding password is generated by a hash function using as input the previous password and a key out of a once by the sink chosen key chain $k_0, \ldots, k_{n-1}$. Figure 6.1 illustrates the password generation. Each sensor starts using the last generated key, $p_{nj}^n$. If the sink node

![Diagram of password generation](image)

Figure 6.1: Password generation for sensor $S_j$

wants the sensors to change their keys, it publishes the last un-revealed key, i.e. $k_{n-1}$ at first. Because of the same usage of the key set in the password generation process, each sensor $S_j$ can verify whether

\[
\text{hash}(p_{j}^{i-1} || k_{i-1}) \equiv p_{j}^i \quad i = n, \ldots, 1
\]

holds ($p_{j}^{i-1}$ denotes sensor $S_j$’s latest unused password). If so, the sensors can be assured on the sink node’s aim to let them change their passwords as the sink exclusively has the knowledge of the key set. This mechanism is extremely efficient as a single unprotected message is sufficient to realize a secured password substitution.

\[
R \rightarrow S_j : k_i
\]
6.2 Password Substitution for Large WSNs

The effort, measured in used memory at the sensors for each password, is comparable small. The storage of exemplarily 8 passwords generated by using a strong hash function as SHA-1 requires 3.9 % (160 bytes) of the sensor’s memory.

To guarantee a further authentication of messages, these new passwords need to be set up at a newly chosen aggregator. In small clearly arranged WSNs the sink node apprises the aggregator of the new passwords. An encrypted message, using the key solely known to the sink and to the aggregation node, contains all new secrets.

\[ R \rightarrow A : \{ID_0, p^i_0, ID_1, p^i_1, \ldots, ID_n, p^i_n, hash(ID_0, p^i_0, ID_1, p^i_1, \ldots, ID_n, p^i_n)\}_{K_{R,A}} \]

Depending on the amount of sensors, \( n \), this message has to be split up into several packets to match the capacity of a TinyOS-packet. The exclusion of the compromised aggregator’s new key in the message, lets the whole network return to a safe state.

6.2 Password Substitution for Large WSNs

The aforementioned approach, unfortunately, does not scale for large WSNs (see Chapter 5.3) as (a) no mechanism is defined to support group keys and (b) every aggregator must be briefed by the sink node separately which does not scale for a huge amount of aggregators. Therefore, we prefer an approach in which the difficulty of key distribution is solved locally, i.e. independently within each cluster (see Chapter 5.3).

6.2.1 Preconditions

Our proposal assumes that all nodes share a symmetric key, \( k_{all} \), to secure an initial configuration against passive attacks. Further we assume, that all nodes are saved from being corrupted during this bootstrapping phase. Thus, an attacker has solely the possibility to eavesdrop – maybe encrypted – messages but not to include forged messages to the ongoing protocol. Within this initial configuration, each node negotiates an individual key with all its one-hop-neighbors. This negotiation process requires an unique identifier for each sensor. We estimated the amount of direct neighbors to about 20 which is, from the memory point of view, still manageable for a sensor. Additionally, each node has to store two more keys: one, which it individually shares with the sink and a second one, which is a group key, respectively a cluster key. Recapitulating, the following values have to be stored at each node \( j \).

- \( ID_j \), unique identifier of node \( j \)
- \( k_{all} \), key that is needed in the bootstrapping phase to prevent passive attacks
- $c_l$, a cluster key, which is chosen by the node at run-time
- $l$, identifier of the cluster
- $k_{R,S_j}$, master key that sensor $S_j$ shares with the sink
- $k_{S_j,S_k}$, set of keys that sensor $S_j$ individually shares with all its neighbors $S_k$

### 6.2.2 Procedure

**Bootstrapping phase**

In our proposal we suggest to use a specific alignment of aggregators and the corresponding sensors described by the placement of each cluster’s aggregator one hop closer to the sink as the sensors. This approach requires the nodes to know about their position compared to the sink. For this reason, the sink initially generates a message, containing a hop level field, which is securely broadcasted by the usage of the commonly known key $k_{all}$. On receipt and verification of this message each receiving node increases the hop distances field and sends out the message again.

$$R \rightarrow \{A_l, S_j\} : \text{hop level, hash(hop level || } k_{all})$$

$$A_l \rightarrow \{A_p, S_q\} : \text{hop level, hash(hop level || } k_{all})$$

After knowing the amount of hops to the sink, each node starts negotiating keys with its direct neighbors. We suggest to use a mechanism that is part of the LEAP-protocol [11]. First of all, each sensor derives its own master key shared with the sink by computing

$$k_{R,S_j} = \text{hash(ID}_j \ || \ k_{all}).$$

Sensor $S_j$ discovers its neighborhood by sending “hello messages” that include information on its hop level.

$$S_j \rightarrow S_q : \text{ID}_j, \text{hop level}_j, \text{hash(ID}_j \ || \ \text{hop level}_j \ || \ k_{all})$$

Each neighbored sensor $S_q$ stores these values and answers with its own identifier and hop level.

$$S_q \rightarrow S_j : \text{ID}_q, \text{hop level}_q, \text{hash(ID}_q \ || \ \text{hop level}_q \ || \ k_{R,S_q})$$

This response is secured through the usage of $S_q$’s master key shared with the sink. At first sight, the usage of this key is not obvious but it offers a possibility to which we will refer in the following Chapter 6.2.3. Both now derive their pairwise key by computing

$$k_{S_j,S_q} = \text{hash(ID}_j \ || \ k_{R,S_q}).$$
6.2 Password Substitution for Large WSNs

In the case, that both started the negotiation process to each other simultaneously, i.e. if both have received a response message, they choose

\[ k_{S_j,S_q} = \begin{cases} 
\text{hash}(ID_j || k_{R,S_j}), & ID_j < ID_q \\
\text{hash}(ID_q || k_{R,S_j}), & ID_j > ID_q 
\end{cases} \]

After this process is over, i.e. after a specific duration after the sensor got switched on, each sensor \( S_j \) erases the key \( k_{\text{all}} \). Recapitulating, each sensor possesses after the bootstrapping phase several keys that it shares with its neighbors, information on which hop level itself and the neighbors are placed and a master key, that it individually shares with the sink.

**Operating phase**

In the following, each sensor securely addresses a neighbor from a lower level which it uses as an aggregator. The election process of the nodes used as aggregators is delegated to an external protocol. Although energy optimization is part of the aim of this work, we feel that the aggregator-election-process opens a wide range of research as it was, for instance, done in the LEACH-protocol [12]. Therefore we solely offer the possibility to let these protocols run securely.

After having informed the aggregator \( A_l \) on its election, the aggregator answers with a randomly chosen cluster key \( c_l \).

\[ S_j \rightarrow A_l : ID_j, \text{hash}(ID_j || k_{S_j,A_l}) \]
\[ A_l \rightarrow S_j : \{c_l\}_{k_{S_j,A_l}}, \text{hash}(c_l) \]

Finally, the sink needs to be instructed on the chosen cluster key to be able to address these nodes in the future. This message may be forwarded by several other aggregators before reaching the sink.

\[ A_l \rightarrow A_p : ID_l, \{c_l, \text{hash}(c_l)\}_{k_{R,A_l}}, \text{hash}(ID_l || \{c_l, \text{hash}(c_l)\}_{k_{R,A_l}} || k_{A_l,A_p}) \]
\[ A_p \rightarrow R : ID_l, \{c_l, \text{hash}(c_l)\}_{k_{R,A_l}}, \text{hash}(ID_l || \{c_l, \text{hash}(c_l)\}_{k_{R,A_l}} || k_{R,A_p}) \]

Different possibilities to forward the message to the sink are conceivable. We decided to minimize the energy consumption at the aggregators closer to the sink as they potentially have to forward several “cluster key messages”. Therefore, resulting from the computation of a an encryption and a hash value, we accepted higher costs at the aggregators acting as a cluster head. Each other aggregator on the way to the sink only needs to verify the received message at the costs of one hash computation.

On the aggregator’s suspicion of being compromised or due to energy balancing reasons the sensor switches to another node on the aggregator-level.
6.2.3 Extending the Sensor Network

As it has already been indicated in Chapter 2.4.1, the network should offer the possibility to be increased. This is allowed by the chosen mechanism in the negotiation process during the bootstrapping phase. An additional sensor $S_z$ equipped with the initially used key $k_{all}$ firstly computes its master key shared with the sink through $k_{R.S_z} = \text{hash}(ID_z, k_{all})$ and starts afterwards the detection of its neighborhood. As this new sensor is not aware of its hop position to the sink, it has to gain this information from its neighbors. Therefore, the sensor informs its neighbors about its own level after having negotiated the pairwise keys. $S_z$ addresses its neighborhood in the following way.

$$S_z \rightarrow S_q : ID_z$$

Each receiving sensor answers with the response-message defined in the initial configuration process.

$$S_q \rightarrow S_z : ID_q, \text{hop level}_q, \text{hash}(ID_q || \text{hop level}_q || k_{R.S_q})$$

As $S_z$ possesses $k_{all}$ it can compute $S_q$'s master key and can therefore verify the message authentication code. Furthermore, both parties, $S_q$ and $S_z$, have now all the information needed to compute their pairwise key $k_{S_q,S_z}$.

$$k_{S_q,S_z} = \text{hash}(ID_z || k_{R,S_q})$$

After this negotiation phase is finished, every new spread out sensor erases the key $k_{all}$. Finally, $S_z$ has to inform its neighbors on its hop level. $S_z$ gains this information from the data received during the negotiation of the keys. We have to distinguish three cases.

- $S_z$ is placed in a one-hop-distance to the sink. In the negotiation process $S_z$ has discovered the sink and therefore infers its position.

- $S_z$ is placed anywhere in the middle of the WSN. $S_z$ received three different hop levels during the key-negotiation: the one on which it is placed itself, the one which is one hop closer and the one which is one hop farther to the sink. From these three data it can infer its own level, i.e. the one in the middle.

- $S_z$ is placed at the edge of the WSN. $S_z$ has at least received a message from a level one hop closer to the sink as its own one. Maybe, in the case that there are more nodes on its own level, also messages from this one. For sure, $S_z$ did not receive two hop-levels having a distance of two to each other. Therefore the sensor can conclude, that it is placed at the edge of the network.

After having found out on which hop level itself is placed, it informs every node with which it negotiated a key on the estimation.
\( S_z \rightarrow S_q : \ hop_{level_z}, \ hash(hop_{level_z} || k_{S_q,S_z}) \)

In the bootstrapping phase we assumed that the sensors are safe from corruption. This assumption does not hold anymore in a later extending phase. Consequently, a corruption of the new spread-out sensors before they erased \( k_{all} \) would compromise the whole network. In [11] the authors proposed a protocol to overcome this problem. However, as with this proposal additional overhead comes along, its application depends on the aimed level of security.
COMPROMISED NODES
7. IMPLEMENTATION

7.1 TinyOS and nesC

We implemented our proposal on the Mica2-motes that were at our disposal. These sensors are equipped with the TinyOS operating system that was developed at the UC Berkeley. TinyOS is an event-driven operating system that was designed for sensor network nodes. TinyOS was reimplemented in the programming language nesC—an extension to C that takes care of the principles of TinyOS. nesC is build upon the following concepts [23]: a program consists of components that are assembled to form a whole program. Each of these components is split up into a specification and into an implementation part. With interfaces, that are declared in the former part and that are provided or used, components get statically linked to each other in a bidirectional way: each interface’s user has access to commands implemented by the interface’s provider and must implement a set of events with which the provider usually signals the finalization of a complex operation. The usage of interfaces is not mandatory: it is also possible just to announce the used functions and events but, anyhow, the structure and clarity of the program increases with interfaces. Within a component several threads may run at the same time. These threads are either manually started by activating a task or they have accessed the component through an interface caused by a hardware interrupt. In contrast to tasks that simply follow the “run-to-completion”-rule, interrupt handlers may interrupt themselves and also tasks.

A component is either a module or a configuration. While modules provide a C-code implementation of the specification, a configuration solely satisfies the specification by connecting or wiring several components together (depicted in Listing 7.1 and 7.2).

```plaintext
module MyModule {
  provides {
    interface interface_A;
  }
  uses {
    interface interface_B as B1;
  }
```
interface interface_B as B2;

implementation {
    command void A.do_something() {
        ... call B1.any_function();
        signal A.done();
    }

    event void B1.something_happened(uint8_t e) {
        ... post arbitrary_task();
        ...
    }

    event void B2.something_happened(uint8_t e) {
        ...
    }

    task void arbitrary_task() {
        ...
        atomic {
            ...
        }
        ...
    }

    void arbitrary_function(uint8_t a) {
        ...
    }
}

Listing 7.1: Abstract Module

Each module must implement all commands specified in the provided interfaces and all events declared in the used interfaces. *nesC* does not allow overloading of functions unless they are defined in different interfaces. Tasks, that independently run at undetermined moments, are activated by the keyword *post* (see Listing 7.1, line 20). To avoid any race conditions, threads can be controlled by semaphores expressed by the keyword *atomic* (line 30).

configuration My_Configuration {
    provides {
        interface interface_A as A;
    }
    uses {
    }
}

implementation {
    components My_Module, interface_B.Module as BM_1,
        interface_B.Module as BM_2;
    A = My_Module.interface_A;
    M.B1 -> BM_1;
    M.B2 -> BM_2;
7.2 Implemented Modules

An arrow, as used in Listing 7.2 between the module \( M \) and the component \( BM_1 \) (line 12), binds a used specification (in \( M \)) to an implementation (here in the fictitious component \( BM_1 \)). Through the static linkage, each calling of the interface \( B1 \) in \( M \) is forwarded and executed via the implementation in the component \( BM_1 \) (the signaling of events proceeds vice versa). The equal sign as used in line 11 simply loops through the concerned components. Here, every command provided by the interface \( \text{interface}_A \) and called on the configuration \( \text{My\_Configuration} \) is passed to the implementation provided in the module \( \text{My\_Module} \) (signaling proceeds vice versa).

As aforementioned, interfaces are bidirectional. Their specification comprises commands that must be implemented by the interface’s provider and events that, accordingly, needs to be handled by the interface’s user. Listing 7.3 outlines an abstract interface.

```c
interface interface_A {
  command void do_something();
  event void done();
}
```

Listing 7.3: Abstract Interface

For a more detailed introduction to \( nesC \) we refer to [23].

7.2 Implemented Modules

Our implementation embraces the approach proposed for large sensor networks and consists of almost 2500 lines of code. Figure 7.1 depicts a survey of the involved modules. We specified eight different protocol-packet types for controlling the bootstrapping phase (4 protocol types), for distributing cluster keys (2) and to process service requests by the sink and service responses by the sensors (2). Each of these packet types is implemented within a specific \( nesC \)-module.

On receipt of a message or on the sensor’s impulse to send a message, the corresponding module is called by the centralized module \( \text{PacketDifferentiatorM} \). The \( \text{PacketDifferentiatorM} \) controls the authentication protocol: it contains the protocol logic, connects each implemented module with each other and has access to external modules as the radio interface. Furthermore, it is going to provide interfaces for authenticated sending and receiving to upper layers.

Concerning the sensors, our implementation includes some configuration files, the centralized \( \text{PacketDifferentiatorM} \)-module, the modules dealing with the protocol types, several interfaces as well as a module caring about security issues. Furthermore, we use modules already provided by \( TinyOS \). This includes
• *GenericComm*-module which controls the radio interface and offers interfaces for sending and receiving messages

• several *TimerC*-modules that can be either configured as sending periodically a signal or solely one time

• *RC5M*-module providing an implementation of the RC5 block cipher algorithm

• *RandomLFSR*-module, a random number generator

![Diagram of the modules](image)

Figure 7.1: Survey of the Modules

To represent the powerful sink node, we implemented a *Java*-application running on a conventional computer that is connected via the serial interface to a sensor solely acting as a radio interface. We present both, the *Java*-program as well as the module running on the sensor used as a radio interface, at the end of this Chapter. Now, we have a look to some chosen modules that run on the sensors.

### 7.2.1 *PacketDifferentiatorM*

The *PacketDifferentiatorM*-module is the key-module of the authentication protocol. It is the central connection to all other modules, contains the protocol logic and stores all data the sensor deals with. The source code of this module is depicted in Appendix B to which we will often refer in the following.
The module uses interfaces to send and to receive messages (line 21-22). Our assumed communication characteristic lets the sensors wait for the receipt of a message before they become active. When receiving a message, the module checks the protocol data unit’s header to determine the packet’s type and passes, in case of a known packet type, the message to the corresponding module (l. 240-294). After having investigated the packet, each module answers by signaling the result which lets the PacketDifferentiatorM decide how to proceed (l. 483-644). This reaction depends on the state (l. 45) in which the sensor resides. Within the bootstrapping phase each received message is either a request for a key negotiation or a response on a self initiated negotiation process. Each sensor periodically sends out negotiation requests to discover its neighborhood (l. 176-194). The timer controlling this periodic transmission is started by the receipt of the message containing the sensor’s hop level and is stopped by a second timer determining the end of the bootstrapping phase (l. 127-171). To avoid a sensor never starting the detection of its neighborhood as it has not received the hop-level-message, a third timer starts the key negotiation phase without knowing the sensor’s hop level (l. 197-201). In this case, the hop level needs to be guessed and the neighbors to be informed on this estimation after the bootstrapping phase is finished (l. 151-157). Before entering the service-phase, each sensor chooses an aggregator (l. 160) and obtains a cluster key (l. 163-165). The service-phase is determined by the receipt of service-request message, their validation, their (possible) re-broadcasting (l. 491) and the final sending of a service-response (l. 218-230). Each usage of the radio interfaces passes the function send (l. 469-484) that embeds arbitrary data into a TinyOS-packet.

The PacketDifferentiatorM, furthermore, offers via commands other modules access to several data that are stored in this module (l. 8-14).

### 7.2.2 SREQM

The SREQM-module (service request module) is exemplarily presented for each defined protocol type. In contrast to other protocol type modules, this one does not offer an interface to generate a message as solely a re-broadcasting of a received message is necessary. The source code of the module, to which we refer in the following, is depicted in the Appendix C. The module gets activated by an external call of the receive-function (line 29-39). The function simply checks whether the module is not already processing another message and if some time passed since the last execution (l. 30). This is necessary to avoid the re-processing of the same message received through a neighbor’s broadcast. If these two conditions hold, the module prepares the actual validation of the message within a semaphore (l. 32-39) and returns the acceptance for processing the message. The more complex validation-process is initiated by the activation of a separate task.
The task first checks whether the received value is fresh, i.e. if the value has a lower ordering number in the hash chain as the last one verified before. In case of a bigger number, the verification process fails and in case of an equal number the module accepts the package with the probability RES_PROBABILITY (see Chapter 5.1.3). In the latter case, the module uses the external interface Random to retrieve a random number used to analyze the probability (l. 53). If the ordering number of the received value is valid, the module executes the first step of auditing the received value by calling the hash function several times (l. 62) and returns to wait for the result of the computation (l. 64) as the hash computation is executed in another task. A single hash computation, however, could be executed without starting a separate task but as the sensor might have missed some service requests before, several computations could be necessary. On signalizing the finalization of the hash computation (l. 84) the module checks the validity of the received value (l. 86) and updates, in case of a successful test, the last received value (l. 88). Finally, the verification process ends with the signalization of the result to the PacketDifferentiatorM-module (l. 93-100).

7.2.3 Sink Implementation

The implementation of the sink node consists of two parts: the first one includes the program’s logic that is implemented in Java. The second one, however, is a nesC-module and solely deals as an interface between the radio interface to the sensors and the serial line interface to the sink’s Java-implementation. We start with an investigation of the Java code (see Appendix D.1).

The Java implementation offers a command line interface with which the user controls the sending of service requests and the initial message on the hop level. The connected sensor that is used to loop through messages to the radio interface is represented by the MoteIF-class that is part of the tools provided by TinyOS (line 20). Several message types can be registered at the MoteIF-class in order to let listeners react on an incoming message (l. 42-88). The message-classes required in the registration process were auto-generated on the base of a C-header file consisting of the messages’ definitions. Furthermore, the MoteIF-class offers a send-function that passes messages over the serial interface to the connected sensor (l. 136, 145, 154, 163).

On receipt of an incoming message the corresponding listener passes the message to a specific function processing this message type (e.g. l. 323-354). All computations that may be necessary in a verification process must consider the byte representation of the values: while the sensors work with a Big-Endian representation, Java uses a Little-Endian one. The application of an appropriate function (l. 267-274) bypasses this problem (e.g. l. 332).

Any security process requires the usage of an RC5-implementation. We
7.3 Measurements

used the one that comes along with TinyOS, changed it to a running C-code and gained a system library via the Java Native Interface (JNI). This system library (libjcrypto.so) needs to be loaded on startup of the sink class (l. 481) and can then be used as shown in lines 185 and 189, respectively.

The code of the module running on a sensor to achieve radio access for the Java-application is depicted in Appendix D.2. The module requires several interfaces for communicating with the Java application and the sensors (line 8-13). Due to the auto-generated Java message-classes, each packet sent to the Java-application requires its own interface identified by the specific AM_Type. Another disadvantage that comes along with the auto-generation of the classes is expressed in the constant size of each message. In details, even if a message does not use the full size of the variable use data space, a full-sized message is sent. To avoid sending 29 Bytes of use data each time over the radio interface, our implementation uses the last byte of the payload for the actual size. As shown in line 70, the module adjusts the size of the packet that is sent over the radio interface.

Concerning the other direction, on receipt of a message the module reads the first byte of the payload and decides afterwards which interface is used to pass the message to the Java application (l. 89-112). Accordingly, if the message size is variable, the module uses a fixed size and puts the actual size to the last byte of the payload (l. 105-106).

7.3 Measurements

Code compiled for the Mica2-Motes can also be run in the simulator Atemu. As Atemu prints out the passed clock cycles at specific events, it can be easily used for measurements. In the following we present our results for the successful verification of a sink’s command at a sensor.

Our measurement starts when the radio interface passes a received packet to the PacketDifferentiatorM-module (see Chapter 7.2.1) and ends with the result of the verification process. For these operations our implementation needs 72 507 clock cycles. On a sensor having a 4\(MHz\) processor as we use it, that takes 17.7ms which corresponds to an energy consumption of 4.9\(\cdot\)10\(^{-5}\)mAh. To find out the code’s management overhead, we additionally solely measured the time required for one hash computation. Depending on the implemented security level we obtained values from 10ms up to 14.9ms. These significantly differ from the running time we found for the same hash function in the literature of 2.22ms (see Chapter 8.1). The speed factor of five distinguishing both measurements maybe ascribed to code optimization that we have not pursued as runtime optimization is not an issue in this work. Our hash function implementation is based on the symmetric cipher RC5 coming along with TinyOS. We adopted considerations proposed in [21] to derive a cryptographic hash function from this cipher. The Davies-
Meyer hash-algorithm suggests to divide the input text $x$ into $k$-bit blocks where $k$ is the key size, and padded, if necessary, to complete the last block. A constant initialization vector $IV$ of the size $n$, that also denotes the hash function’s output size, must be pre-specified. Each hash block $H_i$ computes as follows: the previous hash block $H_{i-1}$ (starting with $IV$) gets encrypted using the corresponding input text block $x_i$ as key. $H_i$ is the result of an XOR-operation using as input the result of the encryption and the previous hash block $H_{i-1}$. Figure 7.2 illustrates this procedure.

Figure 7.2: Deriving a hash function from a symmetric cipher

Further research particularly focusing on efficient computations will surely shrink the discrepancy. In any case, our implementation comes along with a management overhead of $17.7\text{ms} - 14.9\text{ms} = 2.8\text{ms}$ that requires 0.000000028\% of the sensor’s overall power.
The presented proposal was developed in order to guarantee authentication while having the strict limitations of sensors in mind. The required protocol needed to be optimized in terms of energy consumption and of provided security level. Both of these properties are not independent from each other as, for example, a higher security level often comes along with additional bits used for the protection which, in turn, forces a larger energy consumption for its transmission. Therefore, each application initially needs to undergo a balancing process to find out the lowest acceptable security standard in order to achieve the longest lifetime for the whole network.

We felt that under the specific constraints of sensors previous protocols still have potential to reduce their energy consumption. Therefore, we firstly present an energy model in this Chapter. We apply this model to our proposal, to TESLA as well as to two different scenarios using Public Key Cryptography. After a final comparison of these approaches, we investigate our proposal in terms of security. We give a formal evaluation of our protocol, analyze the realizable security level of the transmission channel and, finishing, we address the problem of disclosed data through a node’s corruption.

As our implementation solely comprises our proposal for large WSNs and, moreover, this proposal matches the most common application of Wireless Sensor Networks, this Chapter only discusses the energy consumption for this configuration. Since our proposal for small sensor networks is likewise based on the usage of hash functions – though their application is different – the energy consumption for this proposal is easily inferable. More precisely, the protocol presented for large WSNs mainly differs from the one for small WSNs in its comprehension of aggregators. From the omission of these values, we may easily deduce an energy estimation for small WSNs.

In the following analysis we will often refer to the protocol sketch depicted in Table 8.1.
Transmitting Processing

\((x_n, n)\) denotes the last verified element that is known to the aggregators and to the sensors. \((x_k, k)\) denotes the latest hash chain value, that the sink supposes to be unrevealed.

0. \(R\) : stores \((x_k, k)\) with \(k < n\)
{\(A_i, S_j\)} : stores \((x_n, n)\)

1. \(R \rightarrow \{A_i, S_j\} : x_k, k\)
2. \{\(A_i, S_j\)\} : \(k < n, \text{hash}^{n-k}(x_k) = x_n\)
3. \{\(A_i, S_j\)\} : stores \((x_k, k)\) as verified value
4. \(A_i \rightarrow \{A_k, S_j\} : x_k, k\)
5. \(S_j \rightarrow R : m, \text{hash}(m \mid\mid x_k \mid\mid k_{R,S_j})\)
6. \(R : \text{hash}(m \mid\mid x_k \mid\mid k_{R,S_j}) = \text{valid}\)
7. \(R : \text{decreases hash chain to} (x_{k-1}, k - 1)\)

Steps 5 and 6 are also required in large WSNs between sensors and aggregators, between aggregators themselves and between aggregators and the sink node.

Table 8.1: Authentication scheme for large WSN

8.1 Energy Model

In this sub chapter we analyze our proposal in terms of energy consumption. As a reference platform we use the Mica-motes of Crossbow [17] as we have detailed informations on their capabilities. These motes exceed the capabilities of the sensors at our disposal as their processors are running at double speed, i.e. at 8MHz.

The sensors transmit data up to a maximum distance of 150m. As this value is only achievable outdoors under best circumstances, we continue our considerations by assuming an appropriate distance of 50m. The sensor’s radio interface works at a speed of 38.4kBaud using the Manchester Encoding that infers a transmission rate of 19.2kb/s. Consequently, the transmission of a packet solely containing the eight byte output of a hash function (which results in an overall size of 22 bytes) takes 9.2ms. The sensors are powered by two AA batteries which, following [18]’s conservative estimation, supply 2200mAh. [17] specified a current consumption of 8mA in active mode (i.e. the computation of simple operations for 1ms requires 2.22 \(\cdot 10^{-6}\)mAh), a consumption of 10mA for receiving and of 27mA for transmitting data with maximum power. In details, the receipt of a packet as specified above requires 2.56 \(\cdot 10^{-5}\)mAh, while sending requires the costs of 6.90 \(\cdot 10^{-5}\)mAh. Our following computations assume the usage of the symmetric cipher RC5: for encryption and decryption, as well as for a hash function based on this cipher. [9] specified the running time for one hash computation using that
hash function to 2.22 ms. In the following we outline our assumptions for our energy model that is closely related to the model presented in [9].

**Processing Energy** $E_P$

We suppose the energy consumption for an arbitrary computation $\zeta$ to be linear to its execution time $t_{\zeta}$.

$$E_P(t_{\zeta}) := t_{\zeta} \cdot E_P(1)$$

**Transmitting Energy** $E_T$

Although our proposal does not use more than one packet necessary to deliver a message, we discuss this energy model in a more general way. Therefore $p(m)$ denotes the number of packets required for the message $m$. The energy consumption for delivering $p(m)$ packets increases linear to the transmission of a single packet. Moreover, we conceive a quadratic relation between the energy consumption and the transmission distance $d$.

$$E_T(d, p(m)) := d^2 \cdot p(m) \cdot E_T(1, 1)$$

**Receiving Energy** $E_R$

As aforementioned, we don’t expect a message to be split up into several packets but we discuss this in a more general way. Let $p(m)$ denote the number of packets containing message $m$. The required energy to receive this message increases linear to the amount of packets $p(m)$.

$$E_R(p(m)) := p(m) \cdot E_R(1)$$

Concerning the reference platform that is described above, we obtain the following results.

$$
\begin{align*}
E_P(n \ [ms]) & = n \cdot 2.22 \cdot 10^{-6} mAh \\
E_T(d \ [m], p(m) \ [packets]) & = p(m) \cdot 6.9 \cdot 10^{-5} \cdot (d/50)^2 mAh \\
E_R(p(m) \ [packets]) & = p(m) \cdot 2.56 \cdot 10^{-5} mAh
\end{align*}
$$

In the subsequent sub chapter we will base our computations on the following assumptions taken from [9] and [19]. Actually, the measurement for a hash computation is a little exaggerated as it was taken from a hash function producing a 10 byte output (instead of a 8 byte output as in our case). Anyhow, for the purpose of comparison, we adjust our proposal to 10 bytes in Chapter 8.1.4 which leads to a more exact calculation.

<table>
<thead>
<tr>
<th>operation $\zeta$</th>
<th>$t_{\zeta}$</th>
<th>$E_P(\zeta)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H$ hash</td>
<td>2.22 ms</td>
<td>4.9 $\cdot$ 10^{-6} mAh</td>
</tr>
<tr>
<td>$G$ encryption/decryption</td>
<td>4.0 ms</td>
<td>8.9 $\cdot$ 10^{-6} mAh</td>
</tr>
<tr>
<td>$P$ any simple operation</td>
<td>negligible</td>
<td>-</td>
</tr>
</tbody>
</table>
8.1.1 Energy Map of Our Proposal

We list now the energy consumption for the protocol scheme as described in Table 8.1. First, we outline an estimation solely dealing with sensor in one hop distance to the sink. On the base of this model, we later extend our calculations to aggregators. Our computations will only affect energy that is spent at the sensors as the sink has no power limitation. Table 8.2 lists the required energy for the protocol. In the specification of our energy model we assumed a maximal transmission distance of 50\( m \) for the sensors. As the sensors are also considered to be arbitrarily spread out, we cannot assume a specific alignment of the sensors, i.e. each two sensor may be out of range to each other or within a distance of 0\( m \) to 50\( m \). Therefore, we based our computation on an average range of 25\( m \) between two neighbored sensors. Furthermore and due to reasons of simplicity, we assumed a lossless transmission of the messages resulting in single hash computation required in the verification process, i.e. \((n - k) = 1\) (cp. Table 8.1). Of course, and especially in large WSNs, the assumption of an one-hop distance from the sensors to the sink does not hold anymore. Therefore, we have to consider additional spending of energy for each aggregator. Let \(r = 10\) denote the average number of sensors, whose data an aggregator deals with. Beside the aggregation function which we do not take into consideration as there exist several applications, we have to add the energy required for \(r\) receptions and verifications of messages (protocol step 5 and 6). Furthermore, each aggregator has to spend energy for re-broadcasting the sink’s initial message (see protocol step 4). As in the listing above, we solely discuss the reception of the authentication protocol, i.e. the required energy at an aggregator is computed as follows.

<table>
<thead>
<tr>
<th>Energy Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_R(\mathcal{H}) )</td>
</tr>
<tr>
<td>( E_P(\mathcal{P}) ) + ((n - k) \cdot E_P(\mathcal{H}))</td>
</tr>
<tr>
<td># affects only an aggregator</td>
</tr>
<tr>
<td># ( E_P(\mathcal{H}) + E_T(\mathcal{H}) )</td>
</tr>
<tr>
<td>-</td>
</tr>
</tbody>
</table>

with \((n - k) = 1\): \[ \sum = 5.27 \cdot 10^{-5} \text{mAh} \]

\# := message exchange

Table 8.2: Energy Scheme for the Protocol described in Table 8.1
\[ 5.27 \cdot 10^{-5} \text{mAh} + E_T(\mathcal{H}) + r \cdot E_R(\mathcal{H}) + r \cdot E_P(\mathcal{H}) \]
\[ = 5.27 \cdot 10^{-5} \text{mAh} + E_T(25, 1) + r \cdot E_R(1) + r \cdot E_P(2.22) \]
\[ = 5.27 \cdot 10^{-5}\text{mAh} + 1.73 \cdot 10^{-5}\text{mAh} + 10 \cdot 2.56 \cdot 10^{-6}\text{mAh} + 4.44 \cdot 10^{-6}\text{mAh} \]
\[ = 37.04 \cdot 10^{-5}\text{mAh} \]

The discrepancy between the required energy for aggregators and common sensors justifies the usage of appropriate measures that allow a balancing of chosen aggregators in order to achieve a maximal lifetime of the whole network.

### 8.1.2 Energy Map of TESLA

As aforementioned TESLA realizes low-cost authentication based on the security of a one-way hash function. To avoid an adversary of impersonating the sender, TESLA uses one-time keys that are only valid within a specific time frame. Therefore TESLA requires a second additional message which is sent after the time frame expired to reveal the used key and to let the receivers verify the authentication of the previous message. Each used key is part of a one way hash chain which prevents an adversary of using his own keys.

Our scenario, however, differs from the general application for which TESLA is designed. On the one hand, we do not only take a \( 1:n \) authentication into account, as TESLA does, but also the way back, i.e. a \( n:1 \) authentication. Hence, the following energy computation is not comparable to our previous estimation (see Table 8.2) as solely one way is accounted. On the other hand, TESLA wastes energy due to the static characteristic of the message sent out by the sink to let the sensors start their work. We will explain this issue on the basis of the following short depiction of TESLA.

As long as the message \( m \) is static, the additional costs caused by

\[ a) \ R \to \{A_l, S_j\} : \ m, \text{hash}(m \ || \ x_k) \]
\[ b) \ A_l \to \{A_m, S_k\} : \ m, \text{hash}(m \ || \ x_k) \]

wait until time frame for \( x_k \) expires

\[ c) \ R \to \{A_l, S_j\} : \ (x_k, k) \]

\((x_k, k)\) denotes the last unrevealed hash chain kept by the sink. As each sensor possesses a verified value \((x_n, n)\) with \( n > k \) it can conclude the validity of the used key \((x_k, k)\). Here, we assume the message \( m \) to have a fixed and not alterable semantic.

Table 8.3: Schema of TESLA

the transmission of the second message (message \( c \) in Table 8.3) are without
48 8 ANALYSIS OF THE PROPOSAL

Energy Consumption

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a)</td>
<td>$\sharp$</td>
<td>$E_R(1) + E_P(\mathcal{P})$</td>
</tr>
<tr>
<td>b)</td>
<td>$\sharp \ast$</td>
<td>$E_T(1, 25)$</td>
</tr>
<tr>
<td>c)</td>
<td></td>
<td>$E_R(1) + E_P(\mathcal{H})$</td>
</tr>
</tbody>
</table>

for each sensor: $\sum = 5.61 \cdot 10^{-5} \text{mAh}$
for each aggregator: $\sum = 7.34 \cdot 10^{-5} \text{mAh}$

$\sharp :=$ message exchange
$\ast :=$ affects only an aggregator

Table 8.4: Energy Scheme for TESLA

any reason. The sensors cannot gain any information from $m$ and, consequently, the omission of message a) in Table 8.3 would have the same affect. Therefore, we renounce on a comparison of TESLA to our proposal as such a comparison is anyhow unfair to TESLA because TESLA by default needs an additional message (see Table 8.3, message a) which does not exist in our proposal. Rather, we list the energy consumption that TESLA spends for one authenticated message as our proposal seldom requires TESLA as well and in these cases TESLA’s costs additionally accrue to the estimation for our proposal made in Chapter 8.1.1. Remind, that we suggested to use TESLA for non-static messages used to publish the identifiers of corrupted nodes or the change of semantic for a command in WSN supporting several services (see Chapter 5.2). Table 8.4 lists the costs of the TESLA scheme.

8.1.3 Energy Map of Public Key Cryptography

The usage of Public Key Cryptography has several pros and cons. On the one hand, it is often used to solve the problem of key distribution but on the other hand it comes along with huge computation costs. The problem of the key distribution can either be solved by pre-deploying the public keys or by the usage of certificates which bind a public key to the signature of a third trusted party. In the following we discuss the usage of Public Key Cryptography for our scenario in two different ways. In the first one we use it in both directions, i.e. to authenticate the sink towards the sensors and, on the way back, to authenticate the sensors iterative towards the sink. The analysis of this approach leads us to the second way, in which Public Key Cryptography is only used to authenticate the sink to the sensors – the other way back is secured by symmetric cryptography.

Public Key in both directions

As aforementioned, the verification of signatures requires the recipient to possess the sender’s public key. In case of the sink node it is possible to
equip every sensor with the sink’s public key in pre-deployment. In case of the sensors, however, we don’t have the knowledge of the neighborhood of a specific sensor in advance, i.e. we cannot equip the sensor with its neighbors’ public keys. Hence, the usage of certificates is indispensable. The size of a certificate is determined by the size of the public key as well as by the length of the signature securing this key which both varies with the chosen public key algorithm. We choose the Elliptic Curves Digital Signature Algorithm (ECDSA) as, regarding the generation and the verification of a signature, it outperforms other public key mechanisms in terms of energy consumption and of key size. This is documented in [20] where the authors compared the required energy on an 8-bit CPU for generating and verifying signatures using on the one hand ECDSA with a key size of 160 bit and, on the other hand, the RSA counterpart with a key size of 1024 bit. Both operations together, as we need it in our scenario, require $315.9 \text{mJ}$ for RSA and $47.9 \text{mJ}$ for ECDSA.

An ECC-certificate based on a 160-bit ECC-key has a length of 61 bytes which requires the transmission of three TinyOS-packets [9]. As we haven’t taken a look to the bootstrapping phase in the energy analysis of our proposal, we neglect the one-time-costs of exchanging certificates under neighbors. On the base of a 160-bit ECC-key a single signature requires 40 bytes which forces the transmission of two TinyOS-packets of 34 bytes each. Still assuming a transmission rate of $19.2 \text{kb/s}$, the transmission of both packets comes up to an overall duration of $28.33 \text{ms}$. Following our energy, model this results in an energy consumption for sending and receiving such a packet of

\[
E_{T_{\text{ECC}}} (d \ [m]) = 21.25 \cdot 10^{-5} \cdot (d/50)^2 \text{mAh}
\]

\[
E_{R_{\text{ECC}}} = 7.87 \cdot 10^{-5} \text{mAh}
\]

[9] declared the time necessary to compute an ECC signature as $t_{\text{gen}} = 3 \text{s}$ and the time required for a verification as $t_{\text{ver}} = 6 \text{s}$. This results in a consumption of

\[
E_{\text{genECC}} = 6.7 \cdot 10^{-3} \text{mAh}
\]

\[
E_{\text{verECC}} = 13.3 \cdot 10^{-3} \text{mAh}
\]

for generation and for verification. Table 8.5 depicts a modification of the protocol shown in Table 8.1 using public-key signatures instead of one way hash chains and symmetric keys. Table 8.6 outlines the energy consumption for this protocol.

The main drawback of this approach is the significant amount of energy that is spent at each aggregator as this has to compute multiple verifications and one signature. The sense of these verifications is the avoidance of an adversary sending arbitrary messages which would be forwarded to the sink. One might argue that the energy spent for forwarding is less than the energy spent for verification and, hence, the verification process should be simply
\( (k_j^+, k_j^-) \) denote the public and the private key of sensor \( S_j \). \( \{expr\}_k \) denotes the application of the key \( k \) to the expression \( expr \). By the usage of \( i \), which is an increasing number, the sink proves freshness towards the sensors. \( \mathcal{N}(S_j) \) denotes the set of all neighbors of sensor \( S_j \).

0. \( R \) : stores \( k_j^+ \)
   \( S_j \) : stores \( (k_j^-, k_j^+) \), \( S_i \in \mathcal{N}(S_i) \)

1. \( R \to \{Ai, S_j\} : i, \{i\}_{k_R^-} \)
2. \( \{Ai, S_j\} : \{\{i\}_{k_R^-}\}_{k_R^+} = i \)
3. \( \{Ai, S_j\} : \) stores \( i \) as last verified value
4. \( Ai \to \{Ak, S_j\} : i, \{i\}_{k_R^-} \)
5. \( S_j \to \{Ai, R\} : m, \{hash(m || i)\}_{k_j^-} \)
6. \( \{Ai, R\} : \{\{hash(m || i)\}_{k_j^-}\}_{k_j^+} = hash(m || i) \)
7. \( Ai \to \{Ak, R\} : m, \{hash(m || i)\}_{k_j^-} \)
8. \( R \) : \( \{\{hash(m || i)\}_{k_j^-}\}_{k_j^+} = hash(m || i) \)
9. \( R \) : increases \( i \) to \( (i + 1) \)

Depending on the number of hops from a sensor to the sink, steps 7 and 8 can be omitted or have to be applied several times.

Table 8.5: Public Key Cryptographic in both directions

omitted. Anyhow, such a protocol leads to an unsatisfactory result and therefore, in the following, we replace the Public Key Cryptography on the way back to the sink with the symmetric scheme based on hash functions that we suggested in our proposal.

Public Key in one direction

Before we go into details of the protocol we have to make some adjustments concerning the level of security. The energy analysis of our proposal was based on a level of security that slightly differs from the one discussed in this Chapter and therefore the results are not instantaneously comparable. In anticipation of Chapter 8.1.4 where we discuss a modification of our proposal that leads to a comparison of all presented protocols, we use henceforth the following modified values. The security of a 160-bit ECC-key as used above corresponds, in our application, approximately to a hash function with an 80-bit output (see [13]). Hence, the time required for sending and receiving a hash output changes as follows (see 8.1.4).
8.1 Energy Model

Energy Consumption

\[ k = 10 \] denotes the average number of sensors with which an aggregator deals with.

<table>
<thead>
<tr>
<th></th>
<th>( E_{R ECC} )</th>
<th>( 7.87 \cdot 10^{-5} \text{mAh} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.</td>
<td>( E_{ver ECC} )</td>
<td>( 13.3 \cdot 10^{-5} \text{mAh} )</td>
</tr>
<tr>
<td>3.</td>
<td>( E_P(\mathcal{I}) )</td>
<td>-</td>
</tr>
<tr>
<td>4.</td>
<td>( E_{TECC}(25) )</td>
<td>( 21.25 \cdot 10^{-5} \cdot (25/50)^2 \text{mAh} )</td>
</tr>
<tr>
<td>5.</td>
<td>( \sharp ) ( E_{gen ECC} + E_{TECC}(25) )</td>
<td>( 6.7 \cdot 10^{-3} \text{mAh} + 21.25 \cdot 10^{-5} \cdot (d/50)^2 \text{mAh} )</td>
</tr>
<tr>
<td>6.</td>
<td>( \star ) ( k \cdot (E_{R ECC} + E_{ver ECC}) )</td>
<td>( 10 \cdot (7.87 \cdot 10^{-5} \text{mAh} + 13.3 \cdot 10^{-5} \text{mAh}) )</td>
</tr>
<tr>
<td>7.</td>
<td>( \sharp \star ) ( E_{gen} + E_{TECC}(25) )</td>
<td>( 6.7 \cdot 10^{-3} \text{mAh} + 21.25 \cdot 10^{-5} \cdot (25/50)^2 \text{mAh} )</td>
</tr>
<tr>
<td>8.</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9.</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

for each sensor: \( \sum = 0.02 \text{mAh} \)
for each aggregator: \( \sum = 0.16 \text{mAh} \)

\[ \sharp \text{:= message exchange} \]
\[ \star \text{:= affects only an aggregator} \]

Table 8.6: Energy Scheme for the Protocol described in Table 8.5

\[
E_T(d [m]) = 7.5 \cdot 10^{-5} \cdot (d/50)^2 \text{mAh} \\
E_R = 2.78 \cdot 10^{-5} \text{mAh}.
\]

Our following considerations focus on the protocol depicted in Table 8.7. As the protocol solely requires the sensors to verify a signature, we have to consider whether the usage of an RSA-signature saves more energy than an ECDSA-signature. On the one hand, a RSA-verification is about 10 times faster than an ECDSA-verification (see [9]) but, on the other hand, it also has a larger size which affects the energy required for transmission. The transmission of an RSA-signature (128 byte) using the assumed 19.2kbit/s requires 82.5ms. This yields an amount of energy spent for sending this message of \( 61.9 \cdot 10^{-5} \text{mAh} \) and spent for receiving of \( 22.9 \cdot 10^{-5} \text{mAh} \). The verification process only takes 0.5s which requires \( 1.1 \cdot 10^{-3} \text{mAh} \). Overall, the usage of RSA-signatures saves

\[
(E_{R_{RSA}} - E_{R_{ECC}}) + (E_{T_{RSA}} - E_{T_{ECC}}) - (E_{ver ECC} - E_{ver RSA})
= (22.9 \cdot 10^{-5} - 7.87 \cdot 10^{-5}) \text{mAh}
+ (61.9 \cdot 10^{-5} - 21.25 \cdot 10^{-5}) \text{mAh}
- (13.3 \cdot 10^{-3} - 1.1 \cdot 10^{-3}) \text{mAh}
= (15.03 \cdot 10^{-5} + 40.65 \cdot 10^{-5} - 12.2 \cdot 10^{-3}) \text{mAh}
= -11.6 \cdot 10^{-3} \text{mAh}
\]

energy of \( 11.6 \cdot 10^{-3} \text{mAh} \). This computation matches for an aggregator as we included the difference for transmitting a message, i.e. a sensor even benefits more.
transmitting processing

\((k^-_R, k^+_R)\) denotes the public and the private key of the sink. \(\{\text{expr}\}_k\) denotes the application of the key \(k\) to the expression \(\text{expr}\). By the usage of \(i\), which is an increasing number, the sink proves freshness towards the sensors.

0. \(R: \text{stores } k^-_R, i\)

1. \(R \rightarrow \{A_i, S_j\} : i, \{i\}_{k^-_R}\)
2. \(\{A_i, S_j\} : \{\{i\}_{k^-_R}\} k^+_R = i\)
3. \(\{A_i, S_j\} : \text{stores } i \text{ as last verified value}\)
4. \(A_i \rightarrow \{A_k, S_j\} : i, \{i\}_{k^-_R}\)
5. \(S_j \rightarrow R \rightarrow m, \text{hash}(m \parallel i \parallel k^-_{R,S_j})\)
6. \(R : \text{hash}(m \parallel i \parallel k^-_{R,S_j}) = \text{valid}\)
7. \(R : \text{increases } i \text{ to } (i+1)\)

Steps 5 and 6 are also required in large WSNs between sensors and aggregators, between aggregators themselves and between aggregators and the sink node.

Table 8.7: Public Key Cryptographic in one direction

Using RSA-signatures instead of ECC-signatures, Table 8.8 embraces the energy consumption.

8.1.4 Conclusion

Within the last sub chapters we presented protocols for our application using symmetric and asymmetric cryptographic schemes and discussed their energy consumption. We showed that even the application of public key cryptography is possible although this comes along with a much higher power consumption which finally results in a shorter lifetime of the whole network. Therefore, the use of Public Key Cryptography should depend on the scenario, especially if the aimed lifetime allows the insertion of asymmetric cryptography. Before we compare the approaches made in the previous sub chapters, we need to slightly adjust their investigated levels of security. Due to the used security level in our implementation and to the Public Key measurements that were at our disposal, we used in the first, pure symmetric, approach, for instance, a hash output size of 8 byte while the asymmetric one was based on a 10 byte output. Hence, in the following, we briefly adjust the security level of our proposal to the one used for the Public Key schemes and, afterwards, give a final comparison of the approaches.

A message solely containing a hash output of 10 bytes can be transmitted within one TinyOS-packet (24 byte) and takes 10ms (having a transmission rate of 19.2kb/s). With respect to our energy model, this requires energy of \(2.78 \cdot 10^{-5} \text{mAh}\) for receiving and \(7.5 \cdot 10^{-5} \text{mAh}\) for transmitting such a message. The adoption of these values for the energy model depicted in Table
8.1 Energy Model

Energy Consumption

$k = 10$ denotes the average number of sensors with which an aggregator deals.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$E_{RSA}$</td>
<td>22.9 $\cdot 10^{-5}$ mAh</td>
</tr>
<tr>
<td>2.</td>
<td>$E_{verRSA}$</td>
<td>1.1 $\cdot 10^{-3}$ mAh</td>
</tr>
<tr>
<td>3.</td>
<td>$E_{P(\mathcal{P})}$</td>
<td>-</td>
</tr>
<tr>
<td>4.</td>
<td>$E_{RSA}(25)$</td>
<td>$61.9 \cdot 10^{-5} \cdot (25/50)^2$ mAh</td>
</tr>
<tr>
<td>5.</td>
<td>$E_{P(\mathcal{H})} + E_{T(25)}$</td>
<td>$4.9 \cdot 10^{-6}$ mAh + 1.88 $\cdot 10^{-5}$ mAh</td>
</tr>
<tr>
<td>6.</td>
<td>$k \cdot (E_{R} + E_{P(\mathcal{H})})$</td>
<td>$10 \cdot (2.78 \cdot 10^{-5}$ mAh + 4.9 $\cdot 10^{-6}$ mAh)</td>
</tr>
</tbody>
</table>

7. - -

for each sensor: $\sum = 1.35 \cdot 10^{-3}$ mAh
for each aggregator: $\sum = 1.83 \cdot 10^{-3}$ mAh

$\sharp :=$ message exchange
$\ast :=$ affects only an aggregator

Table 8.8: Energy Scheme for the Protocol described in Table 8.7

8.2 infers an overall consumption of $5.01 \cdot 10^{-5}$ mAh for the sensors and of $40.31 \cdot 10^{-5}$ mAh for the aggregators. Compared to the proposal using Public Key Cryptography in one direction only 3.7% of the energy that a sensor spends in that approach and 22.0% that an aggregator requires is used. In terms of the sensors, this issue is illustrated in Figure 8.1. The energy performance of the scheme using public key cryptography in both directions is even worse as our proposal solely requires a fraction of 0.25% (for sensor and aggregator) of its energy. We proposed to make use of TESLA if the meaning of the static command sent out by the sink needs to be changed or the identifier of a compromised node has to be published. Figure 8.2 visualizes a comparison of TESLA and our proposal in the direction from the sink to the sensors for the case that such a “command-changing-message” is not required. Anyhow, we assume a rare occurrence of the necessity for these messages, i.e. possibly affecting 10% of all packets. If this ratio significantly increases, TESLA itself outperforms our proposal in terms of energy consumption (depicted in Figure 8.3). Any public key scheme, however, offers a higher grade of flexibility as it is not bounded to a static message type. Just for reason of comparison, we therefore assume the usage of TESLA for each message sent out by the sink. Taking the raised level of security into account, the energy consumption for TESLA increases compared to our calculation in Chapter 8.1.2. Adding this adjusted value together with the consumption mentioned above infers an overall consumption of $11.6 \cdot 10^{-5}$ mAh for sensors and of $48.24 \cdot 10^{-5}$ mAh for aggregators, i.e. 8.6% respectively 26.4% of the energy required for the one-way public key scheme.

In the end, we want to address a problem that concerns all of the approaches that we presented. Every TESLA message needs to be stored
before its validity can be checked after receipt of the message containing the used key. Furthermore, in our scenario, an aggregator has to rebroadcast the message immediately to distribute the informations in the whole network. An adversary could take advantage from this by sending arbitrary messages to the aggregators in order to exhaust their energy that is required for the retransmission. An aggregator does not have the possibility to verify a packet before retransmitting it as otherwise, any recipient of the forwarded message would discard it because of the expired key included in the packet’s MAC. Contrariwise, public key cryptography does not exclude such a verification but pays the cost of a signature verification that is even more expansive than the retransmission. The adversary’s goal of exhausting a node in the sensor network works as well. Our proposal, however, cannot distinguish between an attack and a transmission failure, for example. The affect of such an attack, however, slows down as we distribute the costs of an answer via the used probability $p$ (see Chapter 5.1.3) to the whole network.
8.2 Security Model

The security of communication in WSNs does not only depend on the security of the messages transmitted between two communication partners as it is the case in conventional wireless networks. WSNs come up with the additional challenge that even the components, i.e. the sensors itself, can get corrupted which leads to the disclosure of the sensor’s memory and, consequently, to all secrets known by the component. The aim of any security proposal is, therefore, the confinement of any damage to a minimum – the whole network should not be affected by the corruption of only a part of it. The security of the whole WSN is a kind of minimum function operating on the lower security level given by the mechanisms protecting the communication channel and on the effort necessary to break the sensors open.

This sub chapter is structured as follows: we start with an analysis of our proposal as proposed by the BAN-logic. After this formal evaluation we discuss the security of the transmission channel. We estimate the maximal security level that is achievable with our approach and ponder the suitability for our applications. In the following, we take a look on the security of the sensors themselves, on the distribution of secrets and we discuss the problem...
of comparing this security level to the one presented for the transmission channel.

8.2.1 Formal Evaluation of the Proposal

With the BAN-logic [22] the authors presented a formalism that deals with the beliefs of trustworthy parties, their communication among each other and the evolution of beliefs that results from the parties’ communication. Thus it is a significant tool to explain central concepts of authentication protocols and to analyze each of them step by step. A successful application of the BAN-logic to a protocol is solely a necessary but not a sufficient condition for the correctness of the protocol. Anyhow, in the following we outline a formal investigation of our proposal.

Before we start with our evaluation, we recall some principles of the BAN-logic that we use in the following.

\( P \equiv X \): \( P \) believes in \( X \), or \( P \) would be entitled to believe \( X \). In particular, the principle \( P \) may act as though \( X \) is true.

\( P \prec X \): \( P \) sees \( X \). Someone has sent a message containing \( X \) to \( P \), who can read and repeat \( X \) (possibly after doing some encryption).
8.2 Security Model

$P \sim X$: P once said X. The principle P at some time sent a message including the statement X. It is not known whether the message was sent long ago or during the current run of the protocol, but it is known that P believed X when he sent the message.

$\#(X)$: The formula X is fresh, that is, X has not been sent in a message at any time before the current run of the protocol.

$P \leftrightarrow Q$: P and Q may use the shared key K to communicate. The key K is good, in that it will never be discovered by any principal except P or Q, or a principal trusted by either P or Q.

Furthermore, we make use of the following two rules that were set out in [22].

1. If P believes that the key K is shared with Q and sees a message X encrypted under K, then P believes that Q once said X:

$$P \equiv Q \leftrightarrow P, P \chi \{X\}_K$$

2. If one part of a formula is known to be fresh, then the entire formula must also be fresh:

$$P \equiv \#(X)$$

$$P \equiv \#(X,Y)$$

Our following considerations only concern our proposal for large sensor networks. Furthermore, we omit an investigation of the bootstrapping phase as its correctness belongs to the LEAP-protocol [11] of which parts were taken. Consequently, we start with the result of the bootstrapping phase: each two neighbored sensors share an individual symmetric key with each other. The objective of the protocol is to prove freshness and authentication for each received messages. Our assumptions are geared to the protocol illustrated in Table 8.1. The description of our aim in terms of the BAN-logic looks as follows.

for each sensor $S_j$

$S_j \equiv \#(x_k, k)$

$S_j \equiv R \sim (x_k, k)$

for the sink $R$

$R \equiv \#(hash(m || x_k || k_{R,S_j}))$

$R \equiv S_j \sim hash(m || x_k || k_{R,S_j})$

¿From the bootstrapping phase and the pre-configuration of the nodes (protocol step 0), we can easily derive the following assumptions.
for each sensor $S_j$

\[ S_j \models (x_n, n, k_{R,S_j}) \]

\[ S_j \models R \models (x_n, n) \]

\[ S_j \models \#(x_{n-k}, n-k) \quad \forall k: 0 < k \leq n \]

\[ S_j \models R \leftrightarrow S_j \]

\[ S_j \models R \models S_j \cdot S \]

\[ S_j \models R \equiv S_j \cdot S \]

\[ S_j \models R \equiv S_j \cdot S \]

\[ R \models S_j \]

\[ R \models S_j \equiv R \leftrightarrow S_j \]

\[ R \models \#(x_k) \quad \forall k: 0 \leq k < n \]

We continue with the proof for the sensors and discuss the one for the sink later on.

Each sensor $S_j$ receiving the message $X = (x_k, k)$ audits the validity of the received value. From the comparison with the last received value, the sensors conclude the freshness of the value – freshness in the sense, that the sensor has not seen it before (protocol step 2). We conclude

\[ S \models \#(X) \quad (8.1) \]

Based on the security of the used hash function, the sensor concludes that all fresh values of the hash chain have been said by the sink and only the sink, as it exclusively knows the first value of the chain, i.e.

\[ \forall k' : 0 \leq k' < k : \quad S_j \models R \models (x_{k'}, k') \]

where $(x_k, k)$ denotes the last verified hash chain value at the sensor. If the received value is fresh, which has already been checked through equation 8.1, the sensor concludes by applying the rule above that the sink has sent this value, i.e.

\[ S_j \models R \models X \quad (8.2) \]

Protocol step 7 assures the freshness of hash chain values in following runs. Taking both results into account (equation 8.1 and 8.2), we conclude an authenticated message has recently been sent from the sink to the sensors.

In case of the sink, we regard the received message $Y = m$, $\text{hash}(m \ || \ x_k \ || \ k_{R,S_j})$ where $x_k$ denotes the latest used hash chain value sent out by the sink. If the sink’s own hash computation matches the received hash value, the sink can infer the freshness of the message as it knows the freshness of the hash chain value that affected the message authentication code. By applying rule 2 of the BAN-logic, we infer the freshness of the received message, i.e.

\[ \frac{R \models \#(x_k)}{R \models \#(Y)} \quad (8.3) \]
8.2 Security Model

As the key \( k_{R,S_j} \) is included in the MAC-value we deduce that \( R \) sees a message “encrypted” by the \( k_{R,S_j} \), i.e.

\[
R \triangleq m, \ \{ \text{hash}(m \parallel x_k) \}_{k_{R,S_j}}
\]

We neglect \( m \) as its relevance in the authentication process solely concentrates on its impact on the MAC. Using the upper modification of the received message combined with rule 1 of the BAN-logic the sink concludes that \( S_j \) once said \( \text{hash}(m \parallel x_k) \)

\[
R \equiv S_j \xleftarrow[k_{R,S_j}]{\rightarrow} R, \ R \triangleq \{ \text{hash}(m \parallel x_k) \}_{k_{R,S_j}}
\]

\[
R \equiv S_j \xleftarrow[\rightarrow]{\sim} \text{hash}(m \parallel x_k)
\]  

and, hence, also said \( Y \). Both results together (equation 8.3 and 8.4), again, infer an authenticated message recently sent from sensor \( S_j \) to the sink.

8.2.2 Security of the Transmission Channel

The transmission between the components is secured by the usage of symmetric keys. The security of the channel is based on two major aspects: the chosen symmetric algorithm and the size of the used key. Our following considerations only deal with algorithms for which a brute force search of the keyspace is known to be the best attack. In case of doubt relating the security of the algorithm it should be simply replaced by a supposed secure one.

Beside the algorithm, the key size has a huge impact on the security of the channel. Lenstra and Verheul [13] estimated a lower bound of a secure key size for commercial applications. They based their computations on an initially made assumption in 1982 and inferred further keys by applying the following observations: deduced from a common interpretation of Moore’s Law, the computation power of PCs and the speed of memory doubles each 18 months which consequently halves the time necessary to break a once chosen key. Moreover, observations of the US Gross National Products have shown, that the budget of organizations – including those concentrating on breaking keys – doubles each 10 years. An investment of this increased amount of money in special hardware accelerates the process of breaking keys as well.

In 1982, the computation power required to break a 56-Bit Data Encryption Standard (DES) key was estimated to be 0.5 Million Mips Years (MMY) which was expected to be an infeasible number. Two years before, a DES key search machine was proposed that breaks such a DES key within two days – at the cost of the one-time investment of $50 million. Although this amount of money seemed to be large enough to discourage attacks on commercial applications, some “specific” organizations could have certainly
afforded such a machine. In spite of the year discrepancy of two years, the authors of [13] assume the machine of 1980 not to be a serious threat for commercial applications in 1982 and agreed therefore on a “common year” in 1982. Lenstra and Verheul found it reasonable to assume a linear coherence between the costs and the speed of the machines – leading, for example, to a $1 million and 100 days machine.

In the following equation, let $y$ denote the year for which we are looking for a secure key size, $m$ the amount of months in which the computation power and the speed of the memory doubles and, finally, $b$ denotes the amount of years that have to pass until the budget of an attacker is twice as big. Our following computations are focused five years to the future, i.e. to the year 2010, as standards for sensors are still under development and changes to the capabilities given by TinyOS could significantly influence security considerations. Concerning our assumptions made above, we use the following configuration: $y = 2010$, $m = 18$ and $b = 10$. The Infeasible number of Mips Years (IMY) in the year $y$ therefore is computed as follows.

$$IMY(y) = 0.5 \cdot 10^6 \cdot 2^{\frac{12(y-1982)}{m}} \cdot 2^{\frac{y-1982}{b}}$$

$$IMY(2010) = 0.5 \cdot 10^6 \cdot 2^{\frac{12(2010-1982)}{18}} \cdot 2^{\frac{2010-1982}{10}} \approx 1.45 \cdot 10^{12}$$

Each double of the IMY-space can be compensated by doubling the key-space as well. Consequently, the recommended size of a symmetric key in the year $y$ can be computed as follows.

$$key\_size(y) = 56 + (y - 1982) \left(\frac{12}{m} + \frac{1}{b}\right)$$

This equation holds for a DES key used in the year $y$ as it is deduced from the initially taken statement of a 56-Bit DES key. If a different encryption algorithm is used, we have to take the varying speed of the algorithms into account as it also affects the time necessary to break a key. In our case, we use the RC5-algorithm. Referring to [14] a one-byte-encryption requires 43 clock cycles using DES and 25 clock cycles using RC5. An adjustment of equation 8.6 leads to the following equation

$$key\_size'(y) = 56 + (y - 1982) \left(\frac{12}{m} + \frac{1}{b}\right) - \log_2 \left(\frac{25}{43}\right)$$

$$key\_size'(2010) = 56 + (2010 - 1982) \left(\frac{12}{18} + \frac{1}{10}\right) - \log_2 \left(\frac{25}{43}\right) \approx 78.25$$

and, finally, to the appropriate key size of $k = 79$ bits. From the size of a sufficient symmetric key Lenstra and Verheul deduct an appropriate output
size of a hash function. Using the estimation that the computation of a hash function is almost the same as the computation of a symmetric encryption and assuming that the Birthday Paradox is the best known attack on a specific hash function, which approximately requires $2^{\text{sizeof(hash \ function)} / 2}$ applications of the function, the authors infer an appropriate hash output size of $2 \cdot k = 158$ bit. Anyway, the literature distinguishes three different properties of cryptographic hash functions: the preimage resistance, the 2nd-preimage resistance and the collision resistance [21]. But only the collision resistance, which declares the infeasibility of finding two arbitrary inputs that have the same hash output, is affected by the Birthday Paradox. In our application, however, such an attack has no relevance and, hence, we keep a hash output size of $k = 79$ bit.

Following the arguments of Lenstra and Verheul presented above, for the costs for hardware we infer from the ($50$ million, 2 days) machine proposed in 1980 to break 0.5 MMY

$\Rightarrow \quad$ $50 \cdot 10^6, 2 \text{ Days, } 0.5 \text{ MMY, in } 1980$

$\Rightarrow \quad$ $200 \cdot 10^6, 1 \text{ Day, } 1 \text{ MMY, in } 1980$

$\Rightarrow \quad$ $50 \cdot 10^6, 1 \text{ Day, } 1 \text{ MMY, in } 2010$

$\Rightarrow \quad$ $190.74, 1 \text{ Day, } 1 \text{ MMY, in } 2010$

to a $190.74$ machine that computes one Million Mips Years within one day in 2010. Following equation (8.5), $1.45 \cdot 10^6$ Million Mips Years is an infeasible number in 2010 which finally results in the costs of $190.74 \cdot 1.45 \cdot 10^6 = \$277$ million for a machine breaking a symmetric key of size $k = 79$ in 2010.

Our proposal deals with two kinds of symmetric keys: individual keys shared between the sensors and the sink node and group keys. While the former gets never sent over the air channel, the latter has to be securely distributed within a cluster (see Chapter 6.2.2). Therefore the maximal key size of a cluster key is restricted through the limited size of a TinyOS-packet. More detailed, as the following message distributes the cluster keys,

$$A_l \rightarrow S_j : \{c_l\}_{k_{S_j, A_l}}, \text{hash}(c_l)$$

the hash output size add together with the encrypted key size (which we expect to be the same as the key size itself) must be less than 30 bytes, i.e.

$$\text{hash\_output\_size + encrypted\_key\_size} \leq 29$$

(8.8)

Even under this constraint, our proposal satisfies the security level proposed by Lenstra and Verheul as, in addition, the hash output size and the size of the encrypted key only claim 20 bytes. Anyhow, we should recall, that Lenstra and Verheul estimated their computations in order to protect commercial applications, whereas our level of security should be assessed much
lower. The required level of security should be carefully investigated for each scenario and finally be set to the lowest possible bound. Each waste of bits for security infers additional costs for computation and for transmission and finally to a shorter lifetime of the whole network (see Chapter 8.1).

8.2.3 Security of the Sensors

In contrast to conventional wireless networks, WSNs come up with the additional challenge of getting corrupted as they do not possess tamper resistant memory. Hence, keys that were once securely negotiated can get revealed which dangers the overall security of the network. Similar attacks are also conceivable in conventional networks by applying so called Side Channel Attacks that bypass the security of tamper resistant memory through the measurement, for instance, of physical states as the energy consumption. In our case, however, such an effort is not necessary and we conceive the reading of a sensor’s memory as an easy operation. An even bigger challenge for an attacker might be the accessibility to a sensor. WSNs are also conceived to be thrown out of airplanes over areas that are possibly hardly within reach. Even if an adversary got access to a single sensor there might be still a lot of effort required to access further sensors. As such difficulties are very hard to measure it is complicated to find a metric in order to compare it in terms of costs or time with other security issues. Furthermore, depending on the application in which the network is used, an imaginable metric, respectively the values, would enormously vary and, thus, a general security conclusion of a once given configuration of a network is useless. Anyhow, as it is our aim to compare the provided security levels of the sensors and the transmission channel, we have to remind that the transmitting distance of a sensor is quite small. Consequently, an attack on the security channel as discussed in the previous sub chapter requires, somehow, the accessibility to the sensors as well. Although, this access might be easier to achieve, we feel that an attack on the sensor’s memory is much more promising.

A way of confining the damage caused by a memory revelation is a deposition of sensitive data at each node whose unveiling can only do less harm, i.e. only a small part of the network is concerned. Ideally, the disclosure of a single key only affects one other node. In our proposal, we only use keys in the communication direction from the sensors back to the sink that are indeed pairwise individual and therefore adhere to the desired security level. The other way, from the sink to the sensors, is secured by the one way hash chain whose last element is anyway publicly known. The confidentiality of each negotiated key is only depending on two neighbored sensors. Hence, the corruption of a single sensor solely affects any communication link passing these two involved nodes. [10] distinguishes between an unaware and a smart attacker. In contrast to the unaware attacker, the smart one has knowledge of the network’s alignment and, thus, knows whose corruption...
would cause the largest damage, i.e. which broken link affects the largest branch of the network. A corruption of all one hop nodes from the sink, for instance, would immediately paralyze the whole network which could not get compensated anymore. We don’t want to go deeper into this discussion as the maintenance of communication links is not specific to WSNs but to every other, especially wireless, network.
ANALYSIS OF THE PROPOSAL
9. **Summary and Outlook**

In this work we discussed the problem of authentication in Wireless Sensor Networks. We realized that we cannot generalize sensor networks as their application is conceivable in many cases. In this work, our focus on sensor networks was geared to a scenario as considered in the EU project *DAIDA-LOS*. Therein, WSNs are intended to inspect perilous parts of a street in order to raise the security of driving. We described these networks as *small* but also had *large* networks in mind that could possibly be used in larger areas as forests, for example.

We analyzed the communication characteristics in these networks and presented an approach that, based on the particularities of the communication, efficiently achieves our aim. Our proposal does not provide authentication in the most general case but for WSNs as we described them in this work. We have shown that our protocol outperforms other authentication protocols in the literature – that contrariwise guarantee low-cost authentication in general – in terms of energy consumption while maintaining the same level of security. We implemented our approach for large WSNs on the *Mica2*-motes of Crossbow. Although we did not especially focus on optimized code, the program’s overhead is very small. Further research should address code optimization and particularly investigate the interaction with other implementation running on the motes. In this context, especially, the question of sufficient memory should be examined by future work.

In particular, we addressed the problem of compromised nodes as the sensors, due to costs, are not equipped with tamper resistant memory. Hence, an adversary can break open a sensor in order to gain sensitive data. We had to address this threat in our proposal to prevent an immediate paralysis of the entire network. Especially the corruption of an aggregator, that comes along with the disclosure of several secrets, required a procedure that let the network return into a safe state. In this work we always assumed that the knowledge of a corrupted node’s identifier could be used to alert the rest of the network. In our opinion, the problem of detecting compromised nodes opens a wide range of research. Our first intuition assumed such a detection to depend on the application in which the network is used. Within a network providing movement detection, for example, a corrupted node would possibly not react at all while in a network provid-
ing temperature measurements a node might misbehave by sending wrong values. Future work should also address this problem.

The security in our proposal is mostly based on the security of a cryptographic hash function as used in one way hash chains or in MACs. During the work on this thesis one of the most secure assumed hash functions, $MD5$, was reported to be insecure. We did not place much emphasis on this discovery as $MD5$ was anyway supposed to be replaced by the stronger $SHA-1$ algorithm in the near future. By the end of this work, some rumors arose claiming $SHA-1$ to be vulnerable to collision attacks. If this came true it would have a huge impact on digital cryptography as it is used today. We cannot pursue these rumors as they come from a paper to be published at the $EUROCRYPT$ in May 2005 – two months after the completion of this thesis. In any case, the validity of these rumors would not effect our proposal since, up to now, solely the property of collision resistance seems to be affected. In Chapter 8.2.2 we already mentioned that this attribute is not relevant in our use.
References


[3] M.O. Rabin: Digitalized signatures and public key functions as intractable as factorization, 1979


A Installation Instructions

Several non-trivial issues have to be considered to get our implementation running. Therefore, in the following, we will give an installation introduction for Linux. Firstly, we deal with the compilation of the sensors’ code and the porting to the hardware. Secondly, we outline a way how Java gains access to the serial interface.

Beside the TinyOS-system files (latest version available at http://sourceforge.net/projects/tinyos/), the nesC-compiler ncc as well as the cross-compiler-kit CDK4AVR need to be installed (rpm-files are available under http://www.tinyos.net/tinyos-1.x/doc/install.html#linux). During our installation some problems occurred concerning the usage of the Java Native Interface (JNI). Therefore the following files have to be altered as described.

- $TINYOS_HOME/tools/java/jni/Makefile: replace the line specifying the JNI-environment variable with the path to the JNI-header file. Probably in the following way,

  JNI=$JAVA_HOME/include/

- $TINYOS_HOME/tools/java/jni/Makefile.Linux: add the following parameters to the gcc-command

  -I$JAVA_HOME/include/ -I$JAVA_HOME/include/linux

After this installation and modification, place the code within its directory to the TinyOS application directory ($TINYOS/apps/). This directory should already include the Makefile depicted in Listing A.1.

```
COMPONENT=PacketDifferentiator
PFLAGS=-I%T/lib/TinySec
include ../Makerules
```

Listing A.1: Makefile

The script listed in Listing A.2 uses this Makefile to install the sensors’ code on the motes. Some variables have to be set denoting the paths to the executables of the nesC-compiler and the cross-compiler (line 1-2). Additionally, the path to a script is required that assigns an identifier to a mote (l. 3).

```
NCC_PATH=/usr/local/bin
AVR_PATH=/opt/cdk4avr/bin
SET_MOTE_ID_PATH=$TINYOS_HOME/tools/scripts
5 # only allow root to run this script
```
TEST_USER='whoami'
if [ $TEST_USER != 'root' ]; then
        echo "You must be root to run this script"
        exit 10
fi

# check if all paths are included in $PATH
TEST_PATH='echo $PATH | grep $NCC
PATH'
if test -n $TEST_PATH ; then
        export PATH=$PATH:$NCC
        echo $PATH
fi
TEST_PATH='echo $PATH | grep $AVR
PATH'
if test -n $TEST_PATH ; then
        export PATH=$PATH:$AVR
fi
TEST_PATH='echo $PATH | grep $SET
MOTE
PATH'
if test -n $TEST_PATH ; then
        export PATH=$PATH:$SET
fi

# check whether serial interface is specified
TEST_MIB510='echo $MIB510'
if test -n $TEST_MIB510 ; then
        export MIB510=/dev/ttyS0
fi

# install on mote and assign identifier '1'
mica2 install.1
make mica2 install.1

Listing A.2: Sensor Installation-Script

The script only allows the root-user to execute as access to the serial interface /dev/ttyS0 is mostly refused for all other users (1. 6-10). After having included all necessary paths to the environment variable PATH (1. 13-25), the script sets an environment variable that specifies the used serial interface (1. 28-31, we used the first serial interface /dev/ttyS0, respectively COM1 under Windows). Finally the code gets compiled and transmitted to the mote. In this case, we additionally configure the sensor with the identifier '1' (1. 34).

The difficulty of running the Java-program lies in the problematic of getting access to the serial interface as a driver for Linux is not provided by the Java Software Development Kit. We use an open source porting of the original Sun Java Comm API to the Linux OS. You find an installation instruction as well as the files under http://wass.homelinux.net/howtos/Comm_How-To.shtml.

Listing A.3 lists a script that runs the Java-application. Firstly, the script checks whether the Java-classpath includes the tools provided by TinyOS (line 2-5, e.g. necessary for the MoteIF-class).

# check whether classpath includes TinyOS Java tools
CLASSPATH_SET='echo $CLASSPATH | grep $TINYOS_HOME/tools/java'
if test -n $CLASSPATH_SET ; then
        export CLASSPATH=$CLASSPATH:$TINYOS_HOME/tools/java/
After the compilation (l. 8) an environment variable needs to be set that is used by some TinyOS Java tools (l. 12-15). The variable specifies the interface used for the communication (l. 14, serial, /dev/ttyS0) as well as the baud rate of the connected mote (l. 14, mica2 is a synonym for 57600 baud).

Before running the program, the script removes any lock that might still be held on the serial interface from preceeding runs (l. 18). The call of the class requires two parameters (l. 21): the first one is required to bypass a problem that comes along with the serial interface driver. The latter (−Djava.library.path=.) is necessary to include the used crypto-library (libjcrypto.so) that you can find in the application’s directory.
B Source Code of PacketDifferentiatorM

```c
#include "parameters.h"
#include <authentication>

module PacketDifferentiatorM {
  provides {
    interface StdControl as Control;
    command NBOR_t *findAcquaintance(uint16_t id);
    command NBOR_t *getAggregator();
    command uint8_t getHoplevel();
    command uint8_t *getClusterKey();
    command uint8_t *getKeyAll();
    command void setHoplevel(uint8_t new_hoplevel);
    command uint8_t *getMasterKey();
  }
  uses {
    command uint8_t *one_hash(uint8_t *data, uint8_t size, uint8_t *result);
    interface SendMsg as Send;
    interface ReceiveMsg as Receive;
    interface Timer as Timer_Bstr;
    interface Timer as Timer_Nbor_Disc;
    interface Timer as Timer_No_Hop;
    interface Timer as Timer_Leds;
    interface Timer as Timer_Hop_Msgs;
    interface SREQ_I;
    interface SRES_I;
    interface CLS_REQ_I;
    interface CLS_RES_I;
    interface BSTR_HOP_I;
    interface BSTR_NN_I;
  }
  implementation {
    /* internal state */
    state_t state;
    /* current aggregator in use */
    NBOR_t *aggregator = NULL;
    /* hop distance to sink node */
    uint8_t hoplevel;
    /* keys stored by this node */
    uint8_t master_key[HASH_OUTPUT_SIZE];
    uint8_t cluster_key[KEY_SIZE];
    uint8_t key_all[] = KEY_ALL;
    /* all known one-hop neighbors of this node */
    NBOR_t acquaintances[MAX_ACQUAINTANCES];
    bool acquaintances_used[MAX_ACQUAINTANCES];
  }
```
74  B  SOURCE CODE OF PACKETDIFFERENTIATORM

/* buffer for all messages currently working on */
uint8_t stored_msg[MAX_STORED_MESSAGES][TOSHDATA_LENGTH];
bool occupied[MAX_STORED_MESSAGES];
/* set to true if node currently sends a message */
bool busy_sending;
/* message to be sent away */
STORED_Msg_t sent_msg;

/* message for SRES */
/* TEST-MODE: just for temporary use */
uint8_t sres_msg[20];
uint8_t sres_used;

uint8_t *addStoreMessage(void *m, uint8_t size);
result_t removeMessage(void *m);
NBOR_t *chooseAggregator();
void estimateHopLevel();
void informOnHopLevel();
result_t send_(uint16_t addr, uint8_t type, void *data,
uint8_t size);

command result_t Control.init() {
  uint8_t tmp[2 + KEY_SIZE];
  state = BSTR_PHASE;
  busy_sending = FALSE;
  hoplevel = INFINITY;
  /* set every buffer to "unoccupied" */
  memset(occupied, 0, MAX_STORED_MESSAGES);
  /* set every acquaintance to "unused" */
  memset(acquaintances_used, 0, MAX_ACQUAINTANCES);
  /* initialize the leds */
  call Leds.init();
  /* computation of master key */
  memcpy(tmp, &TOSH_LOCAL_ADDRESS, 2);
  memcpy(tmp + 2, key_all, KEY_SIZE);
  call one_hash(tmp, sizeof(tmp), master_key);
  /* TEST-MODE: should be replaced later */
  sres_used = 0;
  return SUCCESS;
}
command result_t Control.start() {
  call Timer_Bstr.start(TIMER_ONE_SHOT, BSTR_DURATION);
  call Timer_NoHop.start(TIMER_ONE_SHOT, BSTR_NO_HOP);
  return SUCCESS;
}
command result_t Control.stop() {
  return SUCCESS;
}

event result_t Send.sendDone(TOS_MsgPtr m, result_t success) {
  removeMessage(sent_msg.stored_data);
atomic busy_sending = FALSE;
return SUCCESS;
}

/* used to determine end of the bootstrapping phase */
event result_t Timer_Bstr_fired() {
    CLS_REQ_t m, *m_ptr;
    bool success = TRUE;

    /* TEST-MODE: visualization of end of BSTR-phase */
call Leds.redOn();
call Leds.greenOn();
call Timer_Leds.start(TIMER_ONE_SHOT, LEDS_DURATION);

atomic {
    /* delete key all */
    memset(key_all, 0, KEY_SIZE);

    /* set internal phase to non-bootstrapping phase */
    state = SERVICE_PHASE;

    /* turn off neighbor discovery timer */
call Timer_Nbor_Disc.stop();
}

/* try to estimate hop level if it is still unknown –
  * mostly only necessary if sensor got spread out at a
  * later point in time
  */
if (hplevel == INFINITY) {
    state = HOP_INFO_PHASE;
    estimateHoplevel();
    informOnHopLevel();

    return SUCCESS;
}

/* choose aggregator */
aggregator = chooseAggregator();

/* generate cluster request message */
m_ptr = (CLS_REQ_t*) addStoreMessage(&m, sizeof(CLS_REQ_t));
if (m_ptr != NULL)
    success = call CLS_REQ_I.generate(m_ptr);
if (m_ptr != NULL && !success)
    removeMessage(m_ptr);

return SUCCESS;
}

/* used in bootstrapping phase – periodically sends out
 * neighbor discovery messages
*/
event result_t Timer_Nbor_Disc_fired() {
    BSTR_NN_t *m_ptr = NULL, m;
    bool success = SUCCESS;

    m_ptr = (BSTR_NN_t*) addStoreMessage(&m, sizeof(BSTR_NN_t));
if (m_ptr != NULL) {
    if (hplevel != INFINITY)
        success = call BSTR_NNI.generate_req(m_ptr);
    else
        success = call BSTR_NNI.generate_ext(m_ptr);
}

/* if acceptance was denied, delete msg from buffer */
if (m_ptr != NULL && !success)
    removeMessage(m_ptr);

return SUCCESS;

/* starts neighbor negotiation without knowing one's hoplevel */
event result_t Timer_NoHop_fired() {
    call Timer_Nbor_Disc.start(TIMER_REPEAT, BSTR_NBOR_DETECT);
    return SUCCESS;
}

/* disables leds */
event result_t Timer_Leds_fired() {
    call Leds.redOff();
    call Leds.greenOff();
    call Leds.yellowOff();
    return SUCCESS;
}

event result_t Timer_Hop_Msgs_fired() {
    informOnHopLevel();
    return SUCCESS;
}

event result_t Timer_SRES_fired() {
    uint16_t tmp = TOS_LOCAL_ADDRESS;
    SRES_t m, *m_ptr;
    m_ptr = (SRES_t *) addStoreMessage(&m, sizeof(SRES_t));
    memcpy(sres_msg + sres_used++ * 2, &tmp, 2);
    call SRES_L.generate(sres_msg, sres_used * 2, m_ptr);
    sres_used = 0;
    return SUCCESS;
}

event TOS_MsgPtr Receive.receive(TOS_MsgPtr m) {
    uint8_t *m_ptr;
    m_ptr = addStoreMessage(m->data + sizeof(AUTH_HD_t),
                            m->length - sizeof(AUTH_HD_t));
    if (m_ptr == NULL) return m;
    switch (((AUTH_HD_t *) (m->data))->type) {

case SREQ_TYPE:
    if (call SREQ_I.receive((SREQ_t*) m_ptr))
        removeMessage(m_ptr);
    return m;

case SRES_TYPE:
    call SRES_I.receive((SRES_t*) m_ptr);
    return m;

case CLS_REQ_TYPE:
    call CLS_REQ_I.receive((CLS_REQ_t*) m_ptr);
    return m;

case CLS_RES_TYPE:
    call CLS_RES_I.receive((CLS_RES_t*) m_ptr);
    return m;

case BSTR_HOP_TYPE:
    /* TEST-MODE: delete in final version */
    if (((BSTR_HOP_t*) m_ptr)->hlevel == 0x00 &&
        TOS_LOCAL_ADDRESS == 0x0003)
        goto remove;
    call BSTR_HOP_I.receive((BSTR_HOP_t*) m_ptr);
    return m;

case BSTR_NN_REQ_TYPE:
    /* neighbor detection request is not used in SERVICE_PHASE */
    if (state == SERVICE_PHASE) {
        removeMessage(m_ptr);
        return m;
    }
    call BSTR_NN_I.receive_req((BSTR_NN_t*) m_ptr);
    return m;

case BSTR_NN_RES_TYPE:
    /* TEST-MODE: delete in final version */
    if (((BSTR_NN_t*) m_ptr)->senderID == 0x0000 &&
        TOS_LOCAL_ADDRESS == 0x0003)
        goto remove;
    /* neighbor detection request is not used in SERVICE_PHASE */
    if (state == SERVICE_PHASE) {
        removeMessage(m_ptr);
        return m;
    }
    call BSTR_NN_I.receive_res((BSTR_NN_t*) m_ptr);
    return m;

case BSTR_NN_EXT_TYPE:
    call BSTR_NN_I.receive_ext((BSTR_NN_t*) m_ptr);
    return m;

default:
    return m;
}

remove:
    removeMessage(m_ptr);
    return m;

uint8_t* addStoreMessage(void* sm, uint8_t size)
{
    uint8_t* buffer = NULL;
}
register bool found = FALSE;
uint8_t i = 0;

for (; !found && i < MAX_STORED_MESSAGES; i++) {
    atomic {
        if (!occupied[i]) {
            occupied[i] = TRUE;
            buffer = stored_msg[i];
            found = TRUE;
            memcpy(buffer, m, size);
        }
    }
}

return buffer;
}

result_t removeMessage(void *m) {
    register uint8_t i = 0;

    for (; i < MAX_STORED_MESSAGES; i++) {
        if (stored_msg[i] == ((uint8_t *) m)) {
            occupied[i] = FALSE;
            return SUCCESS;
        }
    }

    return FAIL;
}

/* dummy function for choosing an aggregator – just pick the * first possible node – should be replaced by LEACH e.g. */
NBOR_t *chooseAggregator() {
    register uint8_t i = 0;

    for (; i < MAX_ACQUAINTANCES; i++) {
        if (acquaintances_used[i] &&
            acquaintances[i].hoplevel + 1 == hoplevel)
            return &(acquaintances[i]);
    }

    return NULL;
}

/* according to the hoplevels of known neighbors it tries * to find the node's own hoplevel */
void estimateHoplevel() {
    register uint8_t i = 0;
    register uint8_t max = 0, min = INFINITY;

    for (; i < MAX_ACQUAINTANCES; i++) {
        if (acquaintances_used[i]) {
            max = acquaintances[i].hoplevel;
            min = acquaintances[i].hoplevel;
        }
        break;
    }
for (; i < MAX_ACQUAINTANCES; i++) {
    if (acquaintances_used[i]) {
        min = (acquaintances[i].hoplevel < min) ? acquaintances[i].hoplevel : min;
        max = (acquaintances[i].hoplevel > max) ? acquaintances[i].hoplevel : max;
    }
}

/* node is probably placed at the edge of the network */
if (max == min)
    hoplevel = min + 1;
/* node probably found neighbors around itself */
else if (min + 1 == max || min + 2 == max)
    hoplevel = min + 1;

/* informs all neighbors on this node's hoplevel --
 * needed if node didn’t receive hop-message or was
 * distributed
 * at a later point in time */
void informOnHopLevel() {
    BSTR NN_t m, *m_ptr = NULL;
    CLS_REQ_t c, *c_ptr = NULL;
    static uint8_t i = 0;
    bool success = FALSE;
    while (i < MAX_ACQUAINTANCES && !acquaintances_used[i]) i++;
    if (i == MAX_ACQUAINTANCES) {
        state = SERVICE_PHASE;
        /* hoplevel of neighbors is now known -- choose
         * aggregator */
        aggregator = chooseAggregator();
        /* generate cluster request message */
        c_ptr = (CLS_REQ_t *) addStoreMessage(&c, sizeof(CLS_REQ_t));
        if (c_ptr != NULL) {
            success = call CLS_REQ_I.generate(c_ptr);
        }
        if (c_ptr != NULL && !success)
            removeMessage(c_ptr);
        return;
    }
    m_ptr = (BSTR NN_t *) addStoreMessage(&m, sizeof(BSTR NN_t));
    if (m_ptr == NULL) return;
    m_ptr->senderID = acquaintances[i++].id;
    if (!call BSTR NN_I.generate_ext(m_ptr)) {
        removeMessage(m_ptr);
    }
    call Timer_Hop_Msgs.start(TIMER_ONE_SHOT, BSTR_HOP_MSGS);
`findAcquaintance(uint16_t id)`

```c
425 register uint8_t i = 0;

for (; i < MAX_ACQUAINTANCES; i++) {
    if (acquaintances_used[i] && acquaintances[i].id == id)
        return &(acquaintances[i]);
}
```

`getAggregator()`

```c
435 return aggregator;
```

`getHoplevel()`

```c
440 return hoplevel;
```

`setHoplevel(uint8_t new_hoplevel)`

```c
445 if (hoplevel == INFINITY)
    call Timer_No_Hop.stop();
    call Timer_Nbor_Disc.start(TIMER_REPEAT, BSTR_NBOR_DETECT);

hoplevel = new_hoplevel;
```

`getMasterKey()`

```c
460 return master_key;
```

`getClusterKey()`

```c
465 return cluster_key;
```

`getKeyAll()`

```c
470 return key_all;
```

`send(uint16_t addr, uint8_t type, void *data, uint8_t size)`

```c
475 atomic busy_sending = TRUE;

sent_msg.msg.addr = addr;
sent_msg.msg.length = size + sizeof(AUTH_HDR_t);
memcpy(sent_msg.msg.data, &type, sizeof(AUTH_HDR_t));
memcpy(sent_msg.msg.data + sizeof(AUTH_HDR_t), data, size);
```

`register_u8_t`
return call Send.send(sent_msg.msg.addr, sent_msg.msg.length, &sent_msg.msg);
}

event result_t SREQ_I.sink_msg_inspected(SREQ_t *m, result_t result) {
  bool success = SUCCESS;
  if (result) {
    /* TEST-MODE: signaling of message received still required */
    call Leds.greenOn();
    call Timer.Leds.start(TIMER_ONE_SHOT, 1000 / hoplevel);
    success = send_(TOS_BCAST_ADDR, SREQ_TYPE, m, sizeof(SREQ_t));
    /* TEST-MODE: send arbitrary message back */
    call Timer.SRES.start(TIMER_ONE_SHOT, 2000);
  } else {
    call Leds.redOn();
    call Timer.Leds.start(TIMER_ONE_SHOT, 1000);
    removeMessage(m);
  }
  return success;
}

event result_t SRES_I.ready(SRES_t *m, result_t result) {
  bool success = SUCCESS;
  if (result) {
    /* check whether SRES message was generated by yourself */
    if (m->senderID == TOS_LOCAL_ADDRESS) {
      send_((aggregator->id), SRES_TYPE, m, m->msg_size + 3 + HASH_OUTPUT_SIZE);
    } else {
      /* you received a SRES from a mote with higher hop level */
      /* TODO send signal to store message - wait to aggregate */
      /* this implementation just for testing */
      memcpy(sres_msg + sres_used++, &m->senderID, 2);
    }
  } else {
    removeMessage(m);
  }
  return success;
}

event result_t CLS_REQ_I.ready(CLS_REQ_t *m, result_t result) {
  bool success = SUCCESS;
}
if (!result) {
    removeMessage(m);
    return SUCCESS;
}

/* our own package — we generated it */
if (m->senderID == TOS_LOCAL_ADDRESS) {
    success = send((aggregator->id, CLS_REQTYPE, m,
                    sizeof(CLS_REQT)));
} else {
    if (call findAcquaintance(m->senderID) != NULL)
        success = call CLS_RESI.generate(
                        call findAcquaintance(m->senderID), (CLS_RESI *) m);
    else
        removeMessage(m);
}
return success;

event result_t CLS_RESI.ready(CLS_RESI *m, result_t result,
                                NBORI *partner) {
    bool success = SUCCESS;
    if (partner != NULL)
        success = send((partner->id, CLS_REQTYPE, m, sizeof(CLS_RESI)));
    else
        removeMessage(m);
    return success;
}

event result_t CLS_RESI.new_cluster_key(uint8_t *ckey) {
    call Leds.yellowOn();
    call Timer.Leds.start(TIMER_ONE_SHOT, 1000);
    memcpy(cluster_key, ckey, KEY_SIZE);
    return SUCCESS;
}

event result_t BSTR_HOPI.ready(BSTR_HOPI *m, result_t result,
                                bool generated) {
    bool success = SUCCESS;
    if (generated)
        success = send((TOS_BCAST_ADDR, BSTR_HOPTYPE, m,
                        sizeof(BSTR_HOPI)));
    else {
        if (result == SUCCESS)
            call BSTR_HOPI.generate(m);
        else
            removeMessage(m);
    }
    return success;
}

event result_t BSTR_NNI.ready(BSTR_NNI *m, result_t result,
                                NBORI *partner, bool
bool success = SUCCESS;

uint8_t type;

/* a message was generated */
if (generated) {
    /* it's a request - broadcast it */
    if (partner == NULL)
        /* send extension message if hop level equals INFINITY */
        success = send_ext((TOS_BCAST_ADDR,
            (m->hlevel != INFINITY) ?
                BSTR_NN_REQ_TYPE :
                BSTR_NN_EXT_TYPE,
                m, sizeof(BSTR_NN_t));
    /* unicast is necessary */
    else {
        switch (state) {
            case HOP_INFO_PHASE:
                type = BSTR_NN_EXT_TYPE;
                break;
            default:
                type = BSTR_NN_RES_TYPE;
                break;
        }
        /* decide whether to send a response or an extension */
        success = send_ext(partner->id, type,
            m, sizeof(BSTR_NN_t));
    }
    /* a received message was processed */
} else {
    /* remove message if it was a response or the verification failed */
    if (partner == NULL || result == FAIL)
        removeMessage(m);
    /* generate a response if verified message was a request */
    else
        success = call BSTR_NN_I.generate_res(m);
}

return success;

event void BSTR_NN_I.new_acquaintance(NBOR_t *new_nbor) {
    register uint8_t i = 0;

    /* TEST-MODE: visualization of "new acquaintance found" */
call Leds.greenOn();
call Leds.yellowOn();
call Timer_Leds.start(TIMER_ONE_SHOT, LEDS_DURATION);

    for (; i < MAX_ACQUAINTANCES; i++) {
        if (!acquaintances_used[i]) {
            atomic acquaintances_used[i] = TRUE;
            memcpy(acquaintances + i, new_nbor, sizeof(NBOR_t));
            break;
        }
    }
}
}  

Listing B.4: PacketDifferentiatorM.nc
C Source Code of SREQM

includes authentication;

module SREQM {  
  provides {  
      interface SREQ_I;  
  }  
  uses {  
      interface Random;  
      interface Hash_I;  
      interface Timer;  
      command void setHChain(uint16_t number, uint8_t *value);  
      command uint8_t *lastHChainValue();  
      command uint16_t lastHChainNumber();  
  }  
}  

implementation {  
  uint8_t busy;  
  /* 'just_worked_on' avoids the re-working on an hash value  
     * that is again received by a broadcast-message sent  
     * out by a child mote  
     */  
  bool just_worked_on = FALSE;  
  SREQ_I *msg;  
  uint8_t tmp[HASH_OUTPUT_SIZE];  
  task void process() {  
    command result_t SREQ_I.receive(SREQ_I* packet) {  
      if (busy || just_worked_on) return FAIL;  
      atomic {  
        busy = TRUE;  
        msg = packet;  
        just_worked_on = TRUE;  
        call Timer.start(TIMER_ONE_SHOT, 1000);  
        post process();  
      }  
      return SUCCESS;  
    }  
    task void process() {  
      /* return FALSE if value is already resolved */  
      if (msg->hchain_no > call lastHChainNumber())  
        goto return_fail;  
      /* if received value is equal to already stored one,  
         * answer with probability RES_PROBABILITY  
         */  
      if (msg->hchain_no == call lastHChainNumber()) {  
        if ((uint8_t) call Random.rand() < RES_PROBABILITY) {  
          if (!memcmp(msg->hchain_value, call  
                     lastHChainValue(),  
                     HASH_OUTPUT_SIZE))  
            goto return_success;  
        }  
        goto return_fail;  
    }  
}
apply hash function to received value

```c
  call Hash_I.hash(msg->hchain_value, HASH_OUTPUT_SIZE, tmp, 
                 lastHChainNumber() - msg->hchain_no);
  return;
```

```c
return fail:
  signal SREQ_I.sink_msg_inspected(msg, FAIL);
  busy = FALSE;
  return;
```

```c
return success:
  signal SREQ_I.sink_msg_inspected(msg, SUCCESS);
  busy = FALSE;
  return;
```

```c
  just_worked_on = FALSE;
  return SUCCESS;
}
```

```c
  if (memcmp(call lastHChainValue(), tmp, HASH_OUTPUT_SIZE)) {
    // received value is valid - store as new verified one
    call setHChain(msg->hchain_no, msg->hchain_value);
  }
  goto return success;
}
```

```c
  signal SREQ_I.sink_msg_inspected(msg, FAIL);
  busy = FALSE;
  return SUCCESS;
```

```c
return success:
  signal SREQ_I.sink_msg_inspected(msg, SUCCESS);
  busy = FALSE;
  return SUCCESS;
}
```

Listing C.5: SREQM.nc
D  Source Code of Sink

D.1  Java Implementation

```java
import net.tinyos.message.*;
import net.tinyos.util.*;
import net.tinyos.packet.*;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Sink {
    private static final int HASH_OUTPUT_SIZE = 8;
    private static final int HASH_NUM_ROUNDS = 4;
    private static final byte[] LOCAL_ADDRESS = {0x00, 0x00};
    private static final byte[] key_all = {0x10, 0x21, 0x32, 0x43, 0x54, 0x65, (byte) 0x76};
    private static final byte[] first_element = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07};
    private int currentChainNo = 999;
    private boolean hop_msg_sent;
    private MoteIF mote;
    private JCrypto crypto;
    private byte[] cluster_key = {0x34, (byte) 0xa9, 0x4b, (byte) 0xea, (byte) 0xf0, (byte) 0xce, 0x11, 0x0a};

    public static int groupId() {
        String gid = Env.getenv("GROUPID");
        if (gid == null)
            return -1;
        return Integer.parseInt(gid);
    }

    public static void main(String[] args) {
        new Sink();
    }

    public Sink() {
        mote = new MoteIF(PrintStreamMessenger.err, -1);
        mote.registerListener(new SRES(), new MessageListener() {
            public void messageReceived(int to, Message m) {
                SRES sres = (SRES) m;
                System.out.println("Received SRES from " + sres.get_senderID());
                short[] tmp_whole = sres.get_msg();
                short[] tmp_part = new short[]
                    {tmp_whole[tmp_whole.length - 1] - 12};
                for (int i = 0; i < tmp_part.length; i++)
                    tmp_part[i] = tmp_whole[i];

                proceed_sres(
                    getByteValue(sres.get_senderID()),
                    getByteArray(sres.get_mac()),
```
getByteValue(tmp_part.length),
getByteArray(tmp_part));
}
}

mote.registerListener(new CLS_REQ(), new MessageListener()
{
    public void messageReceived(int to, Message m)
    {
        CLS_REQ cls = (CLS_REQ) m;
        System.out.println("Received CLS_REQ from " +
                           cls.get_senderID());
        proceed_cls_req(
            getByteValue(cls.get_senderID()),
            getByteArray(cls.get_mac()));
    }
});

mote.registerListener(new BSTR_NN(), new MessageListener()
{
    public void messageReceived(int to, Message m)
    {
        BSTR_NN bstr = (BSTR_NN) m;
        System.out.println("Received BSTR_NN from " +
                           bstr.get_senderID() +
                           ", with hoplevel " +
                           bstr.get_hlevel());
        proceed_nn_req(
            getByteValue(bstr.get_senderID()),
            getByteValue(bstr.get_hlevel()),
            getByteArray(bstr.get_mac()));
    }
});

this.crypto = new JCrypto();
this.hop_msg_sent = false;
this.run();

private void run()
{
    BufferedReader in = new BufferedReader(
        new InputStreamReader(System.in));

    while (true)
    {
        System.out.print("> ");
        String s = " ";
        try
        {
            s = in.readLine();
        }
        catch (Exception e)
        {
            System.err.println("Error while reading from command line");
            System.exit(1);
        }

        if (s == null || s.length() == 0) continue;

        switch (s.charAt(0))
        {
            case 's':
                this.send_sreq();
                System.out.println("Send_Service_Request");
                break;
            case 'o':
```java
public void send(SREQ sreq) {
    try {
        this.mote.send(MoteIF.TOS_BCAST_ADDR, sreq);
    } catch (Exception e) {
        System.err.println("Sending of SREQ failed!");
        e.printStackTrace();
    }
}

public void send(CLS_RES cls_res) {
    try {
        this.mote.send(MoteIF.TOS_BCAST_ADDR, cls_res);
    } catch (Exception e) {
        System.err.println("Sending of CLS_RES failed!");
        e.printStackTrace();
    }
}

public void send(BSTR_HOP bstr_hop) {
    try {
        this.mote.send(MoteIF.TOS_BCAST_ADDR, bstr_hop);
    } catch (Exception e) {
        System.err.println("Sending of BSTR_HOP failed!");
        e.printStackTrace();
    }
}

public void send(BSTR_NN with bstr_nn) {
    try {
        this.mote.send(MoteIF.TOS_BCAST_ADDR, bstr_nn);
    } catch (Exception e) {
        System.err.println("Sending of BSTR_NN failed!");
    }
}

private static byte[] subarray(byte[] array, int start, int size) {
    byte[] sub_array = new byte[size];
    System.arraycopy(array, start, sub_array, 0, size);
    return sub_array;
}

private byte[] hash(byte[] data, int howoften) {
    byte[] result = new byte[HASH_OUTPUT_SIZE];
    // Implementation of hash function
    return result;
}
```
System.arraycopy(data, 0, result, 0, HASH_OUTPUT_SIZE);

for (int i = 0; i < howoften; i++) {
    byte[] tmp = new byte[HASH_OUTPUT_SIZE];
    byte[] iv = {0x00, 0x00, 0x00, 0x00,
                 0x00, 0x00, 0x00, 0x00};
    this.crypto.init(HASH_OUTPUT_SIZE, HASH_OUTPUT_SIZE,
                    (i == 0) ? data : result);
    for (int j = 0; j < HASH_NUM_ROUNDS; j++) {
        tmp = this.crypto.encrypt(iv);
        /* store in both arrays their xor-results*/
        for (int k = 0; k < tmp.length; k++)
            tmp[k] = iv[k] = (byte) (tmp[k] ^ iv[k]);
    }
    /* copy result of this hash computation to final results*/
    for (int j = 0; j < HASH_OUTPUT_SIZE; j++)
        result[j] = tmp[j];
}
return result;

/* computes symmetric key shared with mote 'nodeID' */
private byte[] getMasterKey(byte[] nodeID) {
    int i = 0;
    byte[] tmp = new byte[nodeID.length + key_all.length];
    System.arraycopy(toBigEndian(nodeID), 0,
                    tmp, 0, nodeID.length);
    System.arraycopy(key_all, 0, tmp,
                    nodeID.length, key_all.length);
    return this.hash(tmp, 1);
}

private byte[] getSymmetricKey(byte[] moteID) {
    byte[] tmp = new byte[moteID.length + HASH_OUTPUT_SIZE];
    System.arraycopy(toBigEndian(moteID), 0,
                    tmp, 0, moteID.length);
    System.arraycopy(this.getMasterKey(LOCAL_ADDRESS), 0, tmp,
                    moteID.length, HASH_OUTPUT_SIZE);
    return this.hash(tmp, 1);
}

/* converts byte to int */
private static int getIntValue(byte[] number) {
    int int_value = 0;
    for (int i = 0; i < number.length; i++)
        int_value |= (number[i] & 0xff) << (number.length - i - 1) * 8;
    return int_value;
}
/* converts int to byte */
private static byte[] getByteValue(int i) {
    byte[] result = new byte[2];
    result[0] = (byte) ((i >> 8) & 0xff);
    result[1] = (byte) (i & 0xff);
    return result;
}

/* converts short to byte array */
private static byte[] getByteValue(short s) {
    byte[] result = new byte[1];
    result[0] = (byte) (s & 0xff);
    return result;
}

/* converts short array to byte array */
private static byte[] getByteArray(short[] array) {
    byte[] result = new byte[array.length];
    for (int i = 0; i < array.length; i++)
        result[i] = (byte) (array[i] & 0xff);
    return result;
}

/* converts byte array to big endian (used on the motes) */
private static byte[] toBigEndian(byte[] array) {
    byte[] result = new byte[array.length];
    for (int i = 0; i < array.length; i++)
        result[result.length - i - 1] = array[i];
    return result;
}

private void send_hop() {
    BSTR_HOP bstr_hop = new BSTR_HOP();
    byte[][] tmp_byte, tmp_data;
    short[][] tmp_short;

    /* insert type to packet */
    bstr_hop.set_type((short) 0x10);
    /* insert hop level to packet */
    bstr_hop.set_hlevel((short) 0x00);

    /* insert mac to packet */
    tmp_data = new byte[1 + HASH_OUTPUT_SIZE];
    tmp_data[0] = (byte) 0x00;
    System.arraycopy(key_all, 0, tmp_data, 1, HASH_OUTPUT_SIZE);
    tmp_byte = this.hash(tmp_data, 1);
    tmp_short = new short[tmp_byte.length];
    for (int i = 0; i < tmp_byte.length; i++)
        tmp_short[i] = (short) (tmp_byte[i] & 0xff);
    bstr_hop.set_mac(tmp_short);

    this.send(bstr_hop);
}
private void send_sreq() {
    SREQ sreq = new SREQ();
    byte[] hash_value;
    short[] hash_value_short;

    /* no unknown hash chain value available - return */
    if (currentChainNo < 0) return;

    /* compute hash value and convert it to short array */
    hash_value = this.hash(first_element, currentChainNo);
    hash_value_short = new short[hash_value.length];
    for (int i = 0; i < hash_value.length; i++)
        hash_value_short[i] = (short)(hash_value[i] & 0xff);

    /* build packet together */
    sreq.setType((short)0x00);
    sreq.setHchainValue(hash_value_short);
    sreq.setHchainNo(currentChainNo--);
    this.send(sreq);
}

private void proceed_sres(byte[] senderID, byte[] msg, 
                           byte[] tmp, byte[] mac, 
                           int msg_size, byte[] msg) {
    key = getSymmetricKey(senderID);

    /* compute mac yourself */
    tmp = new byte[senderID.length + msg.length + 
                  HASHOUTPUT_SIZE * 2];
    System.arraycopy(toBigEndian(senderID), 0, 
                    tmp, 0, senderID.length);
    System.arraycopy(msg, 0, tmp, senderID.length, msg.length);
    System.arraycopy(this.hash(first_element, 
                      currentChainNo + 1), 
                    0, tmp, senderID.length + msg.length, 
                    HASHOUTPUT_SIZE);
    System.arraycopy(key, 0, tmp, senderID.length + msg.length + 
                    HASHOUTPUT_SIZE, HASHOUTPUT_SIZE);
    tmp_mac = this.hash(tmp, 1);

    /* if macs vary from each other, return */
    for (int i = 0; i < HASHOUTPUT_SIZE; i++)
        if (mac[i] != tmp_mac[i]) return;

    System.out.print("Received verified message from mote " + 
                     getIntValue(senderID) + ": ");
    for (int i = 0; i < msg.length; i++)
        System.out.print("", + 
                        Integer.toHexString(
                        getIntValue(new byte[]{msg[i]}))
                     );

    System.out.println();
}

private void proceed_cls_req(byte[] senderID, byte[] mac) {
    byte[] tmp = new byte[2 * HASHOUTPUT_SIZE +
D.1 Java Implementation

byte[] key = this.getSymmetricKey(senderID);
System.arraycopy(toBigEndian(senderID), 0, tmp, 0, senderID.length);
System.arraycopy(this.hash(first_element, currentChainNo + 1),
    0, tmp, senderID.length, HASH_OUTPUT_SIZE);
System.arraycopy(key, 0, tmp, senderID.length + HASH_OUTPUT_SIZE, HASH_OUTPUT_SIZE);

/* compute mac */
tmp_mac = this.hash(tmp, 1);
/* return if macs vary from each other */
for (int i = 0; i < tmp_mac.length; i++)
    if (tmp_mac[i] != mac[i]) {
        System.out.println("error");
        return;
    }
/* mac is correct -> send response */
this.send_cls_res(senderID);
private void send_cls_res(byte[] senderID) {
    CLS_RES cls_res = new CLS_RES();
    short[] tmp_short;
    byte[] tmp_byte;
    /* set type to message */
    cls_res.setType((short)(0x03 & 0xff));
    /* insert mac of cluster key */
    tmp_short = new short[tmp_byte.length];
    for (int i = 0; i < tmp_byte.length; i++)
        tmp_short[i] = (short)(tmp_byte[i] & 0xff);
    cls_res.setMac(tmp_short);
    /* insert encrypted cluster key */
    this.crypto.init(HASH_OUTPUT_SIZE, HASH_OUTPUT_SIZE);
    tmp_byte = this.crypto.encrypt(this.cluster_key);
    tmp_short = new short[tmp_byte.length];
    for (int i = 0; i < tmp_byte.length; i++)
        tmp_short[i] = (short)(tmp_byte[i] & 0xff);
    cls_res.setMsg(tmp_short);
    /* insert size of encrypted key */
    cls_res.setMsgSize((short)(tmp_short.length & 0xffff)) ;
    /* insert 'real' receiver to packet */
    cls_res.setTmpReceiver(getIntValue(senderID));
    this.send(cls_res);
}
/* Neighbor Detection Request and Response */
private void proceed_nn_req(byte[] senderID, byte[] hoplevel,
    byte[] mac) {

byte[] tmp = new byte[senderID.length + 1 + HASHOUTPUT_SIZE];
byte[] tmp_mac;

// prepare to compute mac yourself
System.arraycopy(toBigEndian(senderID), 0, tmp, 0, senderID.length);
System.arraycopy(hoplevel, 0, tmp, senderID.length, hoplevel.length);
System.arraycopy(key_all, 0, tmp, senderID.length + hoplevel.length, key_all.length);

// compute mac
tmp_mac = this.hash(tmp, 1);

// return if macs vary from each other
for (int i = 0; i < mac.length; i++)
if (mac[i] != tmp_mac[i])
return;

// mac is valid - send a response
this.send_nn_res(senderID);
}

private void send_nn_res(byte[] senderID) {
BSTR_NN with bstr_nn = new BSTR_NN_with();
byte[] tmp_data, tmp_byte;
short[] tmp_short;

// set type
bstr_nn.setType((short) 0x12);

// set sender address
bstr_nn.set_senderID(getIntValue(LOCAL_ADDRESS));

// set hop level
bstr_nn.set_hlevel((short) 0x00);

// set mac
tmp_data = new byte[LOCAL_ADDRESS.length + 1 + HASHOUTPUT_SIZE];
System.arraycopy(LOCAL_ADDRESS, 0, tmp_data, 0, LOCAL_ADDRESS.length);
tmp_data[LOCAL_ADDRESS.length] = 0x00;
System.arraycopy(this.getMasterKey(LOCAL_ADDRESS), 0, tmp_data, LOCAL_ADDRESS.length + 1, HASHOUTPUT_SIZE);
tmp_byte = this.hash(tmp_data, 1);
tmp_short = new short[tmp_byte.length];
for (int i = 0; i < tmp_short.length; i++)
tmp_short[i] = (short)(tmp_byte[i] & 0xff);
bstr_nn.set_mac(tmp_short);

// set 'real' receiver to packet
bstr_nn.set_tmp_receiver(getIntValue(senderID));
this.send(bstr_nn);
}

private void proceed_nn_ext(byte[] senderID) {
this.send_nn_res(senderID);
}
D.2 nesC Implementation

```nesC
static {
    System.loadLibrary("jcrypto");
}
```

Listing D.6: Sink.java

D.2 nesC Implementation

```nesC
includes sink;

module SinkM {
    provides {
        interface StdControl as Control;
    }
    uses {
        interface SendMsg as SendJava_SRES;
        interface SendMsg as SendJava_CLS;
        interface SendMsg as SendJava_BSTR;
        interface ReceiveMsg as ReceiveJava;
        interface SendMsg as SendSensors;
        interface ReceiveMsg as ReceiveSensors;
        interface Leds;
        interface Timer as Timer_Led;
    }
    implementation {
        TOS_Msg msg_to_sensors, msg_to_java;

        command result_t Control.init() {
            call Leds.init();
            return SUCCESS;
        }

        command result_t Control.start() {
            return SUCCESS;
        }

        command result_t Control.stop() {
            return SUCCESS;
        }

        event result_t SendJava_SRES.sendDone(TOS_MsgPtr sent, result_t success) {
            return SUCCESS;
        }

        event result_t SendJava_CLS.sendDone(TOS_MsgPtr sent, result_t success) {
            return SUCCESS;
        }

        event result_t SendJava_BSTR.sendDone(TOS_MsgPtr sent, result_t success) {
            return SUCCESS;
        }

        event result_t SendSensors.sendDone(TOS_MsgPtr sent, result_t success) {
            return SUCCESS;
        }
```
event result_t Timer_Led.fired() {
    call Leds.redOff();
    call Leds.greenOff();
    call Leds.yellowOff();
    return SUCCESS;
}

event TOS_MsgPtr ReceiveJava.receive(TOS_MsgPtr m) {
    uint16_t addr = m->addr;
    if (*((uint8_t*) (m->data)) == 0x03 ||
        *((uint8_t*) (m->data)) == 0x12) {
        addr = *((uint16_t*) (m->data + m->length - 2));
        m->length = m->length - 2;
    }
    memcpy(msg_to_sensors.data, m->data, m->length);
    call SendSensors.send(addr, m->length, &msg_to_sensors);
    call Leds.redOn();
    call Timer_Led.start(TIMER_ONE_SHOT, 1000);
    return m;
}

event TOS_MsgPtr ReceiveSensors.receive(TOS_MsgPtr m) {
    call Leds.yellowOn();
    call Timer_Led.start(TIMER_ONE_SHOT, 1000);
    memcpy(msg_to_java.data, m->data, m->length);
    /* Received BSTR_REQ message */
    if (*((uint8_t*) (m->data)) == 0x11) {
        msg_to_java.type = 32;
        call SendJava_BSTR.send(TOS_UART_ADDR, m->length, &msg_to_java);
        return m;
    }
    /* Received CLS_REQ message */
    if (*((uint8_t*) (m->data)) == 0x02) {
        msg_to_java.type = 31;
        call SendJava_CLS.send(TOS_UART_ADDR, m->length, &msg_to_java);
        return m;
    }
    /* Received SRES message */
    if (*((uint8_t*) (m->data)) == 0x01) {
        msg_to_java.length = 29;
        msg_to_java.data[28] = m->length;
        msg_to_java.type = 30;
        call SendJava_SRES.send(TOS_UART_ADDR, 29, &msg_to_java);
        return m;
    }
    return m;
}
Listing D.7: SinkM.nc