Secure and Reliable Remote Programming in Wireless Sensor Networks

Dissertation
zur Erlangung des akademischen Grades

DOKTOR RER. NAT.

der Fakultät für
Mathematik und Informatik

der FernUniversität
in Hagen

von
Dipl.-Inf. Osman Ugus

geboren in Ovacık - Türkei

Hagen 2012

1. Reviewer: Prof. Dr. Jörg Keller

2. Reviewer: Prof. Dr. rer. nat. habil. Dirk Westhoff

Day of the defense: 26. February 2013

Bu çalışmayı annem Meryem ve babam İsmail'e adıyorum
Dedicated to my parents Meryem and İsmail

Abstract

Remote programming mechanisms are one of the most essential tools in managing a wireless sensor network (WSN). They allow to program the sensor nodes of a WSN with a new software over-the-air. This is necessary for example to remove bugs or security vulnerabilities from the software running on the sensor nodes. Remote programming mechanisms need to be properly designed and implemented for the use on sensor nodes. Firstly, sensor nodes are typically equipped with very limited hardware resources. The remote programming mechanism must share those limited resources with the code of the actual application. In particular, it must be efficient in terms of the code size. Otherwise, remote programming code does not fit in the flash memory of the sensor nodes. Secondly, sensor nodes are often deployed in public and hostile environments. Hence, remote programming mechanisms must be secured against adversarial interferences. The authenticity, the integrity, and the confidentiality of the software updates must be protected. Moreover, DoS attacks that aim to deplete the resources of the sensor nodes must be mitigated, too. Finally, several remote programming mechanisms use advanced encoding techniques such as Fountain Codes to ensure an efficient software dissemination in unreliable environments. Hence, the security mechanisms must be applicable to the remote programming protocols using advanced encoding techniques such as Fountain Codes. This thesis presents and analyzes security mechanisms for remote software updates in WSNs that have all required properties: they protect the authenticity, integrity, and confidentiality of the software updates, they are DoS-resilient, and they have a small memory footprint.

Kurzfassung

Mechanismen, die eine Programmierung von Sensorknoten over-the-air erlauben, gehören zu den wichtigsten Hilfsmitteln beim Management und Betrieb von Drahtlosen Sensornetzen (WSN). Eine derartige Neuprogrammierung von Sensorknoten ist bspw. notwendig, um Softwarefehler oder Sicherheitslücken aus der Software zu entfernen. Beim Design und der Implementierung derartiger Programmierung Mechanismen sind jedoch zahlreiche Hürden zu überwinden. Zuvorderst verfügen Sensorknoten idR. nur über äußert eingeschränkte Hardwareressourcen. Die Programmierung Mechanismen müssen sich dabei die limitierten Ressourcen mit dem tatsächlich zu betreibenden Programm teilen. Jede Implementation für Sensorknoten muss daher äußert effizient bzgl. der Größe des Codes ausfallen, da der Code zur entferten Programmierung andernfalls nicht in den Flashspeicher eines Knotens passt. Darüber hinaus kommen Sensorknoten häufig gerade in öffentlichen oder sogar feindseligen Umgebungen zum Einsatz. Eine entfernte Programmierung ist daher speziell gegen Angriffe abzusichern und die Authentizität, die Integrität und Vertraulichkeit der Softwareupdates muss geschützt werden. Hierbei sind auch DoS Angriffe, die darauf abzielen, die limitierten Ressourcen eines Knotens aufzubrauchen, zu verhindern. Um die Softwareupdates auch in unzuverlässigen Umgebungen effizient zu verteilen, werden idR. spezielle Techniken, wie bspw. Fountain Codes verwendet. Daher müssen die verwendeten Sicherheitsmechanismen auch bei Verwendung derartiger Techniken sicher anwendbar sein. Diese Arbeit präsentiert, analysiert und evaluiert Sicherheitsmechanismen zur entfernten Programmierung von Sensorknoten, die alle geforderten Vorraussetzungen erfüllen: Sie schützen die Authentizität, die Integrität und Vertraulichkeit der Softwareupdates, sie sind resistent gegenüber DoS Angriffen und weisen nur eine geringe Codegröße auf.

Assistance of third parties and the material used in this thesis

Some chapters of this thesis contain contributions and assistance from the co-authors and the previous works. They are listed below:

- Section 2.3.1.1 relies on the Diploma thesis of the author of this thesis [1].
- Section 2.3.2 is taken from [2]. The taken part, i.e. Section 2.3.2, is entirely written by the author of this thesis.
- Chapter 6 and Section 4.4 are based on a paper [3] which was a joint work with the co-authors. The author of this thesis contributed to all parts of [3]. All material used in Chapter 6 is produced by the author of this thesis. Chapter 6 and Section 4.4 are entirely written by the author of this thesis.
- Chapter 7 is based on a paper [4] which was a joint work with the co-authors. The author of this thesis contributed to all parts of [4] related to security design. Simulations for the overhead analysis (Section 7.5) were not done by the author of this thesis. Chapter 7 is entirely written by the author of this thesis.
- Chapter 8 is based on a paper [5] which was a joint work with the co-authors. The author of this thesis contributed to all parts of [5] related to security design. The prototype implementation and the performance evaluation (Section 8.5) were not done by the author of this thesis. Chapter 8 is entirely written by the author of this thesis.

Declaration

I herewith declare that I have produced this thesis without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This thesis has not previously been presented in identical or similar form to any other German or foreign examination board.

Osman Ugus

Acknowledgements

First of all, I would like to thank my first advisor, Prof. Dr. Jörg Keller. I am deeply grateful to him for dedicating the time and patience to advising me. His comments and ideas have been a great help in the compilation of this thesis. I am also deeply grateful to Prof. Dr. rer. nat. habil. Dirk Westhoff for his guidance throughout my thesis and especially for his support in developing the ideas presented in this thesis.

My special thanks go to Dr. Jens-Matthias Bohli for various joint works and for his comments on several chapters of my thesis. I learned a lot from him.

I would like to thank Alban Hessler and Prof. Dr. Christoph Sorge for their helpful ideas and discussions on different aspects of wireless sensor networks. I also owe sincere and earnest thankfulness to all my co-authors, Nicola Bui, Moreno Dissegna, Dr. Michele Rossi, and Prof. Dr. Michele Zorzi.

I would like to thank Prof. Dr. Thomas Schmidt, head of Internet Technologies Group at HAW Hamburg, for providing me a great working atmosphere while I was writing my thesis.

I also thank the SKIMS project, sponsored by the Federal Ministry of Education and Research (BMBF), for the financial support during my PhD studies at HAW Hamburg. My special thanks also go to EU projects Ubisec&Sens and SENSEI for the financial support during my PhD studies at NEC Laboratories Europe in Heidelberg. Furthermore, I thank NEC Laboratories Europe, where I started my PhD studies, and all of its members.

Finally, my special thanks go to Bojana Avramovic for proofreading of my thesis and for her endless support.

Contents

List of Figures

List of Tables

Glossary

ACK Acknowledge character.

Adv Adversary.

AES Advanced Encryption Standard.

CBC Cipher Block Chaining.

CGIMN Canetti, Garay, Itkis, Micciancio, Naor, Pinkas.

CRC Cyclic Redundancy Check.

DES Data Encryption Standard.

DFY Desmedt, Frankel, Yung.

DoS Denial of Service.

DSA Digital Signature Algorithm.

ECC Elliptic Curve Cryptography.

ECDSA Elliptic Curve DSA.

EEPROM Electrically erasable programmable ROM.

EMSS Efficient Multi-chained Stream Signature.

ESA External Strong Adversary.

FEC Forward Error Correction Code.

FIFO First In First out.

GPS Global Positioning System.

GR Gennaro, Rohatgi.

GW Gateway Device.

HMAC Hash-based MAC.

ISA Internal Strong Adversary.

IV Initialization Vector.

LT Code Luby Transform Code.

MAC Message authentication code.

MD Monitoring Device.

MNP Multihop Network Programming.

MOAP Multihop Over-the-Air Programming.

NACK Negative-acknowledge character.

nesC network embedded systems C.

NMAC Nested MAC.

OFB Output Feedback.

OS Operating System.

RAM Random Access Memory.

ROM Read Only Memory.

RP Remote Programming.

RSA Rivest, Shamir, Adleman.

SHA Secure Hash Algorithm.

SKN Sink Node.

SN Sensor Node.

SRP Secure Remote Programming.

SU Software Update.

TDMA Time Division Multiple Access.

WA Weak Adversary.

WAN Wide Area Network.

WSN Wireless Sensor Network.

1

Introduction

A Wireless Sensor Network (WSN) is a wireless network consisting of a large number of small devices called sensor nodes [6]. Sensor nodes differ from typical computers mainly in two ways. Firstly, they are typically small in size. Hence, they are often equipped with hardware components with limited storage and computation capacity. Secondly, they are powered by a limited power source like batteries. Hence, programs and algorithms to be run on sensor nodes should be designed to cope with the limited power and resources available to them. In fact, once properly solved, these challenges become the advantages of sensor nodes. Firstly, their small size allows for cheap productions. This permits large scale WSN deployments at a reasonable cost. Secondly, due to their tiny size, they can be attached to humans, animals and machines without disturbing them or they can be easily distributed in the field. This makes them applicable in many important scenarios including environmental monitoring, industrial automation, home automation, and health monitoring [6].

As it is the case in every computer system, WSNs need to be maintained after their deployment. Typical maintenance works include updating the software running on the sensor nodes. This might be necessary due to changing application requirements as well as for removing software bugs and security vulnerabilities. Manual software updates would not be practical, in particular when the number of the sensor nodes is large or when they are deployed in difficult-to-reach fields. Hence, software updates must be performed ideally over-the-air. The process of updating the software of sensor nodes is referred to as remote programming.

1. INTRODUCTION

The design of a remote programming mechanism for WSNs is a challenging task. This is due to the limited resources available to sensor nodes and the need of security. The challenges can be divided into two main groups: implementation challenges and security challenges.

The remote programming mechanism must share the program memory (Flash memory ROM) with the code of the actual application. Its implementation may only occupy a very small amount of the program memory. Otherwise, they do not fit on the sensor nodes. Hence, the main implementation challenge is the small code size. Moreover, the limited data memory (RAM) and the power source available to sensor nodes require the remote programming mechanism to be efficient in terms of RAM size and execution performance.

WSNs are often deployed in public and hostile environments. Hence, remote programming mechanisms must be protected against adversarial interferences. The authenticity, integrity, and confidentiality of the software updates must be protected. Moreover, all security mechanisms must be designed in a DoS resilient way. The authenticity protection ensures that the sensor nodes accept only the software updates delivered by authorized entities. The integrity protection ensures that the sensor nodes accept only the software updates which are integer. Finally, the confidentiality protection ensures that the software updates themselves as well as the sensitive information carried by them are prevented from being learned by unauthorized entities. DoS resilience is required to prevent the adversaries from being able to exhaust the resources of the sensor nodes (energy or storage).

This thesis presents and analyzes security mechanisms for remote programming in WSNs. They protect the authenticity, integrity, and confidentiality of the software updates. They are DoS-resilient and they have a small memory footprint.

The rest of this chapter introduces basics of WSNs and remote programming mechanisms and details the contributions of this thesis. A short introduction to WSNs is provided in Section 1.1. Section 1.2 gives a short overview of remote programming mechanisms for WSNs. Secure remote programming and requirements on it are presented in Section 1.3. Contributions of this thesis are given in Section 1.4.

The characteristics of the hardware platform and the operating system used in the implementations of the solutions developed in this thesis are described in Chapter 2.

Chapter 3 presents the security models used for identifying the security requirements for remote programming in WSNs.

Implementation challenges are addressed by using efficient building blocks in implementing the required security mechanisms. This is done by proposing a code-size optimized toolbox in Chapter 4. The toolbox comprises a hash function, which is the fundamental primitive for almost all security solutions, a MAC function for protecting message authenticity and integrity, and finally a signature scheme, for providing non-repudiation in addition to authenticity and integrity protection. The signature scheme is a dedicated design for the secure remote programming mechanism. The entire toolbox is based on a block cipher. High performance is achieved by using only symmetric algorithms. The code size is reduced due to the code sharing.

Security challenges are addressed by proposing several DoS resilient security mechanisms for protecting the authenticity, integrity, and confidentiality of the software updates.

Software updates are broadcast from a single source to a large number of receivers. Hence, security mechanisms required are closely related to the broadcast security. Broadcast security has been addressed by many researches. Hence, there are several broadcast authentication schemes available in the literature. Chapter 5 analyzes the existing broadcast security mechanisms in terms of their applicability to the remote programming scenario. Based on this analysis, an improved version of the off-line signing [12] has been used for securing the software updates.

Chapter 6 presents a code-size optimized and DoS resilient approach for protecting the authenticity and integrity of the software updates. It relies on the improvement of the off-line signing [12]. The implementation results show that the proposed approach occupies only 1% of the available flash memory on a TelosB platform. This is a large improvement compared to the approaches using elliptic curve based signature schemes occupying 28% of the available flash memory on the same platform. Such a great improvement is mainly possible due to the code-size optimized toolbox in Chapter 5.

Fountain codes allow for an efficient and reliable software dissemination even in highly unreliable environments. However, DoS attacks become more powerful due to the natural error-propagating property of the fountain codes. Chapter 7 presents security mechanisms for a remote programing protocol using fountain codes (i.e., rateless erasure codes).

Chapter 8 presents an efficient approach for protecting the confidentiality of the software updates. The proposed approach is based on the hardware implementation of AES available on most modern sensor platforms. A secret key is shared between the sender (i.e., monitoring device) and the receivers (i.e., sensor nodes). Software updates are then encrypted and decrypted using the shared secret key.

Finally, Chapter 9 concludes this thesis.

1.1 Wireless sensor networks

A Wireless Sensor Network (WSN) is a wireless network consisting of a large number of small devices called sensor nodes. Sensor nodes cooperatively monitor their environments in a self-organized way. Monitored data is transmitted to a central data collection point for further evaluation and processing. A typical WSN is composed of a monitoring device, a gateway device, a sink node, and many sensor nodes. A sensor node is equipped with a microprocessor for processing, a memory for code and data storage, a wireless communication module, and some sensors as well as possibly some actuators. Sensor nodes are in general powered by a battery pack and have very limited resources. The sink node is a special sensor node used for connecting a WSN with the external world. It is usually a more powerful device than a sensor node. Data gathered by the sensor nodes is collected at the sink node before leaving the WSN. Likewise, data and commands sent from the remote are delivered to the sensor nodes via a device called the sink node. The sink node is connected to the backbone with a gateway node. The gateway node is therefore typically a PC-class device with the Internet connection. Finally, the monitoring device is a device from where one (or multiple) WSN is managed. Data received from the sensor nodes are stored, analyzed, and processed at the monitoring device. The monitoring device pulls/pushes data from/to a WSN via the gateway node.

1.1.1 Execution model

The communication between the sensor nodes can be one-to-one, one-to-many, many-to-one, and many-to-many. One-to-one communication represents the individual interaction between two sensor nodes. This communication type is rather seldom compared to other ones. It is typically used during the execution of specific protocols such as a key agreement between two sensor nodes. One-to-many communication is used to deliver the same data to multiple receivers. Broadcast of commands, configuration changes, and software updates from the sink node to the sensor nodes is a typical example of a one-to-many communication. It is also applied in data replication protocols. In a data replication protocol, sensor data is replicated at multiple nodes to increase the robustness against e.g. node failures. In a many-to-one communication, multiple senders communicate with a unique receiver. It can be found in protocols used for delivering data from sensor nodes to the sink node. Finally, many-to-many communication is typically applied in scenarios where there are multiple receivers and broadcasters are active at the same time. Propagation of software updates in a large WSN is a typical example of this communication pattern.

Data communication in a small WSN, where all sensor nodes are in the coverage of the sink node, is a relatively easy task. Data is simply broadcast to the sink node. However, it is challenging in large WSNs where sources and destinations are located at multi-hop distances. Hence, in large sensor networks, some sensor nodes must act as a routing node for other nodes in their vicinity. Depending on the size of the network, several number of sensor nodes may need to act as a routing node. A general architecture for a WSN is depicted in Figure 1.1.

1.1.2 Characteristics

WSNs are different from traditional computer networks in many ways. Characteristics unique to them offer great benefits in many application fields. However, they bring also new challenges that need to be solved.

Foremost and most important sensor nodes are typically small in size. This results in an application platform with a constrained energy, storage, and computation capacity. These constraints possibly prohibit the use of algorithms and protocols developed for traditional computer networks in WSNs. Therefore, new protocols specific to WSNs need to be developed. They should be designed to cope with the limited resources available on the sensor nodes. Once properly solved, this challenge brings two most significant advantages of WSNs. Firstly, tiny size of sensor nodes allows for cheap productions. This permits large scale deployments at a reasonable cost. Secondly, due to their tiny size, sensors can be attached to humans or animals without

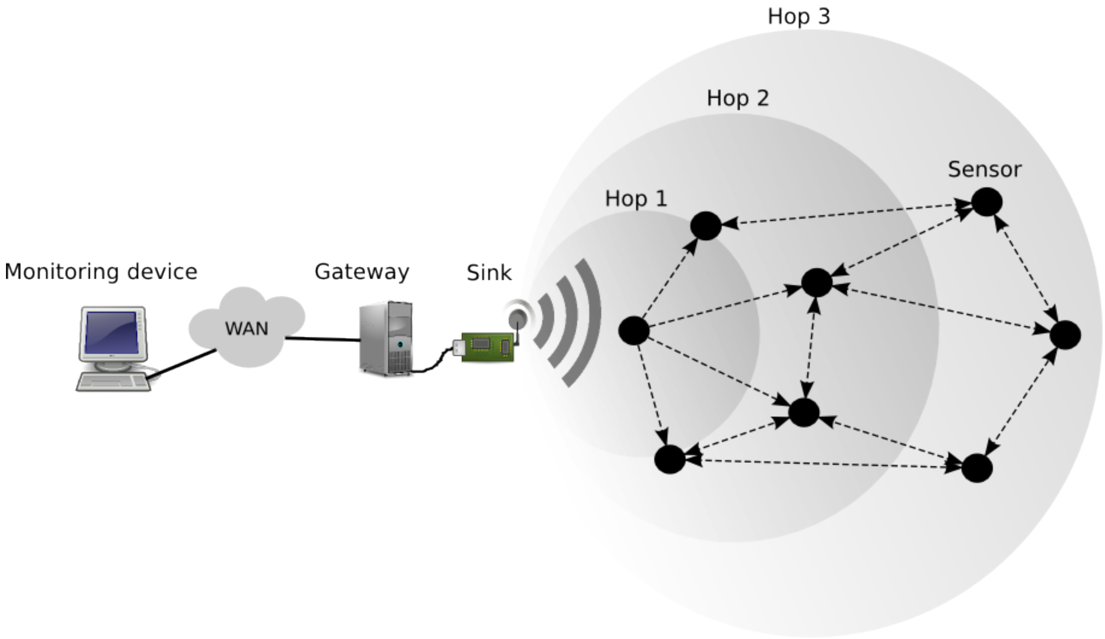


Figure 1.1: Generic WSN architecture - The monitoring device is connected with the gateway node over a WAN. The gateway acts as an interface between the monitoring device and the sink node. The sink node connects the WSN with the gateway node. Sensor nodes communicate with each other and the sink node using wireless links.

disturbing them. This makes them applicable in many important scenarios including body networks for health monitoring.

WSNs are usually deployed in remote areas without infrastructure support [13]. For this reason, sensor nodes are equipped with algorithms and protocols that enable self-management and unattended operation. Self-management refers to the capability of adaptation to environmental or local changes. For example, a self-managing sensor can change its sensing frequency according to its remaining energy budget. Self-management is also an essential requirement for random deployments. Consider a disaster monitoring scenario like the forest fire application. Sensor nodes need to be dropped from an aircraft into the forest randomly instead of their one-by-one placement. Self-managing sensor nodes can configure themselves to execute their mission independent of their locations. This makes WSNs applicable in many important scenarios with inaccessible and dangerous deployment locations.

All these economical and technical advantages resulted in a wide range of WSN applications.

1.1.3 Classification

Depending on the purpose and sensors' capability, WSNs can be classified as sense-only or sense-and-react [14]. Measurement of physical events in the vicinity of sensor nodes and their transmission to a central monitoring device for further analysis is the main characteristic of a sense-only setting. In such a setting, sensor nodes are merely responsible for information gathering. Depending on the connectivity level, gathered data is stored locally or reported to the monitoring device in real-time. Sensor nodes have no means to react against the happenings and the sensed data. Actions that need to be taken are decided and triggered by the monitoring device. An example of a sense-only deployment is the early forest fire detection system. Sensor nodes notify the forest ranger about the temperature in their surroundings. A fire alarm is arisen by the forest ranger if the temperature crosses a critical value. Typically, sense-only WSNs are used in scenarios where the purpose is only information gathering or actions that need to be taken as a response to the measured phenomenon cannot be automatized and the involvement of human beings is necessary. Typical scenarios of such applications are area monitoring such as border controlling for intrusion detection or geo-fencing for observing wildlife.

In a sense-and-react setting, sensor networks are equipped with additional nodes called actuators, typically a mechanical device actuated by a controller. The controller activates/deactivates the mechanical system upon receiving a command to change the physical conditions of the environment in accordance with a predefined rule. Commands can be issued by other sensor nodes or by a central monitoring device. In a typical setting, sensor nodes monitor their environment and send the measured data to a central monitoring device. The monitoring device issues a proper command when a condition is met and sends it to the actuator. An example of a sense-and-react deployment is a building automation system where the air conditioning actuator is controlled by the temperature values received from sensors distributed around the building. Sense-and-react WSNs are used in scenarios where the purpose is to react to the sensed data immediately. Typical scenarios of such applications are industrial automation systems, smart living and tunnel monitoring.

1.1.4 Applications

This subsection presents a set of sense-only and sense-and-react WSN applications. The selection of examples is done to cover a wide range of deployment fields such as wildlife monitoring, environmental monitoring, agriculture monitoring, traffic monitoring, structural monitoring, health monitoring, and home automation. The reader is referred to [13, 14] for further examples.

Wildlife monitoring (sense-only): One of the earliest and most famous applications for wildlife monitoring is the ZebraNet [15]. It was deployed at the Mpala Research Center in Kenya for tracking the movements of Zebras. ZebraNet was composed of 30 tracking nodes and a sink node. Tracking nodes were equipped with a GPS module and attached to Zebras to be monitored. The position of Zebras was recorded every three minutes. Detailed movement information was used by the researchers to determine the lifestyle of Zebras (i.e., eating, sleeping, and resting behaviors). Whenever two tracking nodes met, the collected data was exchanged to ease the data collection. The ranger needed to find only one Zebra every few days or weeks for retrieving the entire data collected by all tracking nodes. The mobility of the tracking nodes and the sink node is a noteworthy property of the ZebraNet.

Environmental monitoring (sense-only): FireWxNet [16] is an example application for monitoring of environmental conditions at forest fires. It was deployed in the Selway-Salmon Complex fires in 2005. FireWxNet was composed of three WSNs and two web cameras. WSNs were deployed in three different locations spanned over 160 square kilometers in total. WSNs were composed of two sensor nodes, five sensor nodes and six sensor nodes. Sensor nodes were used to measure the weather conditions such as humidity, temperature and wind direction to predict fire's behavior. Cameras were used to get visual data from the fire. It was used for example to estimate the required altitude for a safe flight over the fire. Sensor nodes reported their measurements to a sink node via the Chipcon CC1000 radio [17] operating at 900MHz. The sink node was connected to the base station (gateway) running Gentoo Linux via a USB-to-serial cable. The cameras and the base station were connected to the backbone through a switch. Finally, the backbone was connected to the base camp through a directional antenna operating at 924MHz and providing an access range of approximately 50 kilometers. The base camp was connected to the Internet via a satellite. A notable challenge being faced was the bad connectivity of the sensor nodes. The packet loss rate within the WSNs deployed was over 50%. This is indeed a surprising result. One would expect a better connectivity in such a deployment where no radio noise source such as wireless access points around the sensor nodes exists.

Agriculture monitoring (sense-only): A WSN application for vineyard monitoring is demonstrated in [18]. It was deployed for 6 months in a vineyard in the Okanagan Valley in British Columbia. The mission was to measure the rapid temperature variances in a vineyard to predict fruit maturity and crop damage. The WSN was composed of 65 sensor nodes and a monitoring device. Sensor data was transmitted to the monitoring device using a multi-hop network established among the sensor nodes. The reporting frequency for the sensed data was 5 minutes. The data was used to evaluate the impact of temperature on the growth as well as on the damage of grapes. Moreover, it was shown that live sensor data could help with preventing crop loss due to the frost. A quite interesting observation was the difference of the heat summation units within the vineyard. The difference in some points was over 35% in a 100 meters distance. The packet loss rate was around 33%. This was rather an expected behavior due to the harsh vineyard environment.

Another WSN prototype for vineyard monitoring was developed in the European research project UbiSec&Sens [19]. It was deployed for two weeks in a vineyard in Neustadt in Germany. The prototype was composed of 60 sensor nodes. Sensor nodes were equipped with an external humidity sensor to estimate the need of water in the vineyard. Sensor readings were reported to a gateway via the Chipcon CC2420 radio [20]. Security was one of the important design goal. Hence, the prototype was secured with the reliability and security components developed within the UbiSec&Sens project.

Traffic monitoring (sense-and-react): Manzie et al. [21] demonstrated a use case for WSNs in vehicular networks for optimizing the fuel consumption in urban environments. The fuel consumption of a vehicle equipped with a WSN was studied in a simulation environment. The WSN was assumed to provide the current traffic situation by communicating with other vehicles and the roadside units. This information was used to determine the optimum speed for the vehicles for some look-ahead distance. The fuel consumption was impacted badly by the traffic information from smaller look-ahead distances. However, the improvement for a look-ahead distance of over 50s was larger than 20% compared to a vehicle without traffic information. This was even better than the performance of a hybrid-vehicle which provided only 10% improvement in the same simulation setting. The fuel consumption for a look-head distance of 180s was improved up to 38%. It was possible to improve the results even further by combining the hybrid technology with a WSN in case of large look-ahead distances.

Industrial monitoring (sense-only): Krishnamurthy et al. [22] deployed a WSN for industrial monitoring at a semiconductor fabrication plant as well as at an oil tanker in the North Sea. The WSN setting was the same at both application scenarios. It was composed of 150 accelerometers, 26 sensor nodes, 4 gateway nodes and a central monitoring device. Accelerometers were used to measure the vibration level of machines to predict their health status. The vibration information collected by the sensors was reported to the central monitoring device for long-term storage and off-line processing. For failure diagnosis, the gathered data was compared with specific defect characteristics to predict bearing failures, gearbox defects, electrical and other mechanical issues. The robustness of connectivity in both settings was very good. The WSN connectivity was not affected by the radar of the ship and the noisy environment in the engine room. Finally, although the initial design was a multi-hop network, the authors found out that a single hop network would be sufficient for the needed coverage due to the good connectivity.

Home automation (sense-and-react): A building automation system using a WSN to control home appliances over the Internet is demonstrated in [23]. It was based on Sizzle which is an implementation of the https protocol for sensor nodes. In a demo setting, several home appliances such as light switches and thermostats attached with a sensor node running the Sizzle could be controlled via a web browser from a remote location. An interesting observation was that RSA turned out to be not suitable for the implementation. Its implementation occupied almost all of the RAM available on the sensor platform. Hence, authors proposed to apply ECC based alternatives. Moreover, the data transfer was the most time consuming operation in a https connection. The main reason for this was the capacity of the wireless channel rather than computation speed. Hence, a tailored version of the SSL protocol was proposed to reduce the data overhead.

Health monitoring (sense-only): Mercury [24] is a body WSN for monitoring patients with Parkinsons Disease, epilepsy, and stroke. It was composed of 8 wearable sensor nodes and a base station, a laptop class device. The base station was equipped with an IEEE 802.15.4 capable wireless module to be able to communicate with sensor nodes. Sensor nodes were used to record the movement of patients and the physiological conditions. The data monitored by sensor nodes was delivered to the base station upon the request received from it. Moreover, the base station was used to update configuration of the sensor nodes such as sensing and storage policies. A microSD flash memory with 2GB capacity was used to store sensed data locally on the sensor nodes. To test Mercury in a realistic setting, it was deployed on one of the coauthors for over 5 hours. Data from accelerometers and gyroscopes were sampled during the test person performing daily activities such as walking, sitting, and typing at a frequency of 100Hz. The main challenge faced during the test was the limited capacity of the radio channel. The base station was unable to download all samples stored by a sensor node at the speed of sensing. There was a plan to use Mercury for monitoring of patients remotely while residing at home.

1.2 Remote programming

As shown in the previous Section, WSNs are used in many applications. The location of a possible deployment ranges from hard-to-access wildlands such as a forest to the body of a patient as in the health monitoring scenario. The network size of a deployment depends on the application scenario and the purpose. Typically, several tens of sensor nodes are deployed in environmental and industrial monitoring applications [15, 16, 18, 22]. A few tens of sensor nodes are typical for home automation and health monitoring applications [23, 24]. Sensor nodes can be located at physically easily accessible places [18, 22, 23, 24] as well as at hard-to-reach regions [15, 16].

As it is the case in every computer system and network, WSNs need to be maintained after their deployment. For example, software running on the sensor nodes may need to be adapted to changing physical conditions and application requirements. Hence, at some point in time after the deployment of a WSN, it may be necessary to replace the software running on the sensor nodes. The replacement can be incremental requiring to update some parts of the software or complete requiring to update the entire software. The type of the replacement is of interest with respect to performance issues and impacts the design of the protocol to be used for updating the software. However, the main challenge to cope with is the same in both settings. That is the medium used for bringing an update to the sensor nodes. Traditionally, a software running on a sensor node can be replaced over a wired connection. For this, the sensor node is connected to a host computer over a parallel serial port. Subsequently, the host computer transfers the new software into the sensor node using the port connection. However, this method is not practical for real-world WSNs due to many reasons.

Firstly, sensor nodes can be located at a place which is difficult to access. For example, FireWxNet [16] was deployed at inaccessible wildlands. Hence, the researchers were transported with a helicopter to the application field. Obviously, it would not be payable to fly to the application field whenever a software update is needed.

Secondly, sensor nodes can be mobile. Hence, their exact position can be unknown or spread over a large area at the time when a physical access is needed. ZebraNet [15] is a good example for the latter case. It would be expensive in terms of time and money to access all sensor nodes one by one at different locations whenever a software update is required.

Thirdly, a manual software update requires some level of expertise. Consider the application for health monitoring [24] where sensor nodes are attached to the body of a patient. The application domain in such a scenario is typically elder people with limited or no technical know-how. It would be a cumbersome task for a patient to perform a software update.

Finally, it is a time consuming work to manually update the software of several tens of sensor nodes, even if the nodes are easily accessible. To give an example, consider a lab environment where a WSN application is developed. During a software development, the programmer needs to test the software many times. Obviously, plugging all sensor nodes to a computer for updating their software one by one, whenever a test needs to be performed, would consume a large amount of time. This would result in loss of motivation and productivity.

It is therefore desired to update the software of sensor nodes over a wireless channel and to automatize the update process.

1.2.1 Execution model

The operation of a software update in a WSN via wireless communication that requires no physical access to sensor nodes is called *remote programming*[1]. The steps that need to be executed for a software update in a WSN are depicted in Figure 1.2. The software update is prepared and transferred to the gateway device using a wired connection such as the Internet (Steps 1 and 2). The gateway node passes the update to the sink node through a parallel or serial connection (Step 3). The sink node distributes the software update to the sensor nodes using a dissemination protocol via a wireless channel (Step 4). The distribution of the software update starts typically with the broadcast of an advertisement packet followed by the packets of the actual update. Sensor nodes decide to accept or reject the software update by checking e.g., the version number of the update received in the advertisement packet. In a multi-hop setting, sensor nodes at each hop forwards the received update to the sensor nodes at the next hop according to a dissemination protocol (Step 5). This is repeated recursively until the last hop is reached to ensure that all sensor nodes receive the update.

[1] It is also referred to as *over-the-air* programming or *code update*. In this thesis, the term ”remote programming” is used.

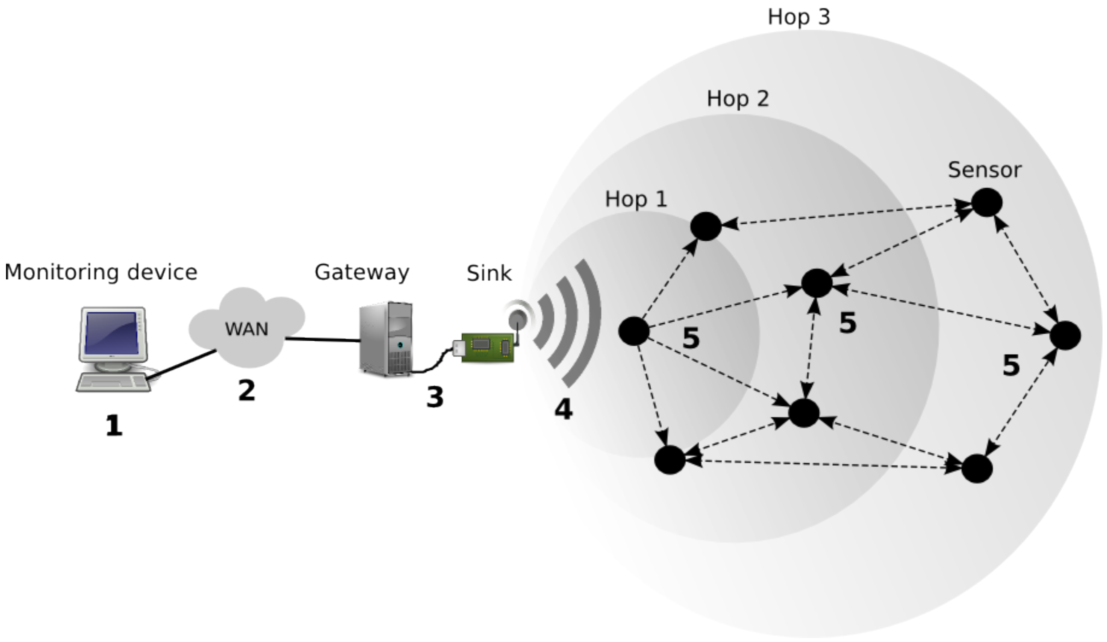


Figure 1.2: Generic remote programming architecture - Steps being performed to update the software in a WSN from a remote device: The new version of the software is delivered to the gateway of the WSN via WAN (1,2). The gateway node disseminates the update to the sensor nodes via the sink node using wireless links (3,4). Sensor nodes at each hop propagate the received update to the sensors at other hops (5).

1.2.2 Classification

Many remote programming protocols for WSNs have been proposed. All of them follow the execution model described above. All of them also have the same goal. That is the dissemination of a software update to all sensor nodes of a WSN in a reliable and an efficient way using unreliable wireless links. They differ mainly with respect to the methods used for improving the performance and efficiency. The improvements mainly target at reducing the communication cost and increasing the dissemination performance in multi-hop settings. In the following subsections, existing remote programming protocols are classified according to the methods applied to achieve these goals. Moreover, for each method, several protocols are shortly introduced as an example.

1.2.2.1 Methods for reducing communication cost

Energy spent for sending data is expensive. Transmitting a single bit of data is more expensive than execution of thousands of instructions [25]. For this reason, several methods have been proposed to reduce the amount of data that needs to be transmitted for a software update. They are largely based on the use of virtual machines, incremental software replacements, custom compilers, and software compressions. Figure 1.3 provides an overview of the commonly used approaches.

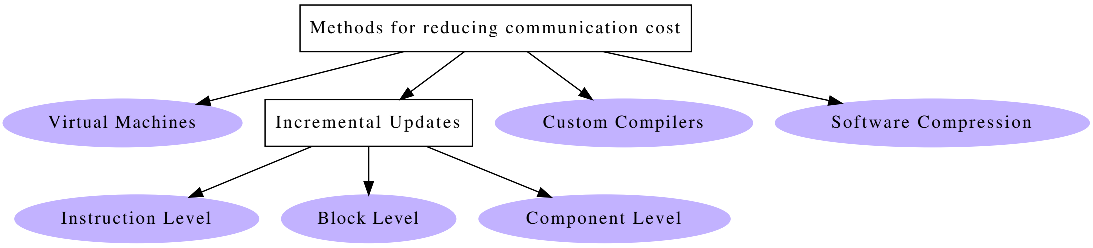


Figure 1.3: Methods for reducing communication cost - Common methods used in remote programming protocols for reducing the size of the software updates to be disseminated.

Virtual machines Virtual machines allow for writing programs in a language that produces much compacter code than the native code. This is achieved by using a specific set of functions and instructions allowing to simplify the representation of complex programs. However, the code produced for such a representation is not native. Therefore, it cannot be executed directly on the actual platform. Virtual machines are used for their execution.

Mate [26] is a virtual machine implementation for sensor platforms. Its design goal is to reduce the communication cost of remote programming by reducing the size of programs. To achieve this goal, it provides custom instructions and functions that can be executed by the virtual machine. Programs are written by using these special functions. This leads to much compacter executables with a smaller size. The smallest piece of executable program, called capsule, is composed of at most 24 instructions. Hence, it fits into a single radio packet. Large programs are composed of multiple capsules. Each sensor node forwards a received capsule to its neighbors. Mate also supports the multi-hop programming of large WSNs. The main drawback of Mate is its low performance in program execution. Other virtual machine examples for WSNs include [27, 28, 29, 30, 31].

Incremental updates Another well known method used for reducing the communication cost in a software update is the incremental update. A remote programming without incremental update requires the transmission of the entire code for a new version. Instead of a full transmission, an incremental update requires the transmission of the differences from the previous version only. Hence, the amount of data that needs to be transmitted for an incremental update is often smaller than that is for an entire update. Due to its energy efficiency, this approach has been used in many remote programming protocols. They mainly vary in the granularity of code segments allowed to be updated. They can be at instruction level, block level, and component level.

- **Instruction level:** An incremental remote programming protocol of this category is proposed in [32]. It works by distributing a diff script instead of the entire software. The diff script contains the update (differences) as well as the patch operations necessary for applying the update. Patching is performed using architecture-specific instructions. Hence, it is a platform-dependent solution. Multi-hop programming is not supported as well. A multi-hop programming protocol based on a similar approach was proposed by Reijers et al. [33]. However, this protocol is also platform-dependent. A platform independent incremental programming based on differential comparison is presented in [34]. It uses a greedy algorithm to generate a minimal update patch between the new and old version of the software. Other incremental programming protocols using byte-level comparison with an improved version of Rsync [35] to generate update patches include [36, 37].

- **Block level:** Jeong et al. [38] proposed an incremental programming protocol based on the Rsync algorithm. The software update is generated by computing the differences between the old and the new versions of the software. This is done by using a derivative of the original Rsync algorithm [35] which is tailored for the resource limited sensor devices. Compared to [32], this approach is hardware independent and can be applied to any platform. Platform independence is achieved by generating the difference between software versions at the block level. No multi-hop support is provided by this solution.

- **Component level:** Diff-like approaches above, generating an update patch by comparing two software versions, ignore the application structure. Hence, even a tiny change in the code may result in a large update patch. This is due to the natural changes in the address space of the code. For example, the address of a function in an update may change from x to $x \pm y$. This happens when the size of the function changes. Using a block-level approach therefore would require to transfer a block for each address change to refresh the pointers referencing them. Moreover, the pointer updates increase the number of the copy and write operations within the program memory. Hence, instruction and block-level approaches are not optimal in terms of the performance and energy consumption.

  This problem is addressed in [39] by providing a slop space after each function. A slop is a piece of empty memory space. Preserving an empty space after each function enables them to shrink or grow within the reserved memory space. Hence, no pointer needs to be updated to reference a function even if its size changes. Only the functions and components that are changed need to be transmitted in a software update. However, in order to manage the slop spaces and the pointers, the linker needs to be modified. The need for a customized linker is the main drawback of this method. Moreover, the flexibility of this method is limited. The reason is that the linking needs to be performed at a remote host where the address map of applications is known.

FlexCup [40, 41] is another programming protocol based on incremental linking. Similar to the previous approach, it allows to update only the parts of a program that have actually changed. Moreover, it is possible to update only a certain component in an application without changing the rest. The linking of the changed components with the code is done on the sensor nodes. This is achieved by sending the new symbol and allocation table along the update patch. Hence, the flexibility is the main advantage of this protocol over [39] that requires the linking to be done at a remote host.

Custom compilers Another method used for reducing the size of an update is the compiler modification [42]. A custom compiler is applied to increase the similarity between two versions of the software. This is achieved by minimizing the instruction differences. The idea is to keep the register allocations as much as possible the same in the new version of the software. However, the resulting code is less efficient in terms of execution. Hence, a trade-off between the introduced runtime overhead and the energy conservation gained from the reduced communication cost needs to be done. For this purpose, the choice of register allocations is modeled as a mixed integer non-linear programming problem. Registers are allocated by solving this problem towards the optimal performance improvement and energy minimization.

The concept of replaceable components is introduced in [43]. A replaceable component is a part of a program that needs to be updated in the future. A compiler extension is used for declaring variables and functions in a program as replaceable. The software update is then performed by transmitting only those parts of the code declared as replaceable in the previous version. However, the flexibility of this approach is limited. It cannot be applied to architectures such as Harvard which are not supporting code execution in RAM.

Software compression A natural method used for reducing the size of software updates is the software compression. Compression algorithms are used to encode the data such that encoded output is smaller than the source. A well-known technique to achieve this goal is to remove the redundancies from the input data using an encoding algorithm. Such a compression looses no information and is referred as a lossless com-

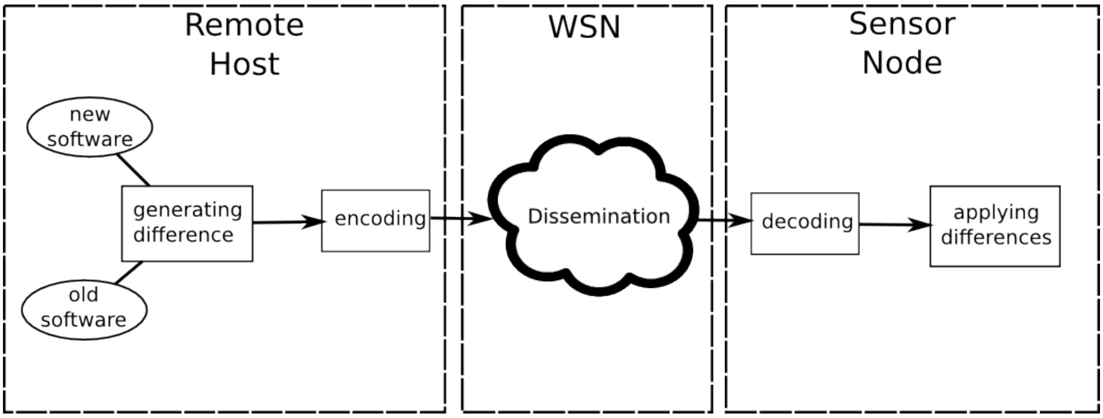


Figure 1.4: Remote programming with software compression - Schemata of an incremental programming using software compression.

pression. The original data is restored from the compression entirely using a decoding algorithm.

A rough schema for an incremental programming improved with software compression is sketched in Figure 1.4. The overhead introduced by the compression is the encoding cost at the remote host as well as the decoding cost at the sensor nodes. Since the remote host is typically a powerful device, the overhead of encoding can be ignored. However, sensor nodes are typically resource constrained. Hence, the selection of the compression algorithm should be done carefully. In particular, the following points need to be evaluated carefully in the selection:

- **Compression ratio:** The amount of energy conserved due to the compression must be larger than the amount of energy required for decoding the compressed data.

- **Code size:** Implementation of a decoding algorithm occupies program memory on the sensor nodes. The program memory available on sensor nodes is typically very limited. Therefore, a compression algorithm with a small code size must be selected.

- **Performance overhead:** Sensor nodes are equipped with a CPU with limited computing power. Thus, a compression algorithm with a small execution overhead must be selected.

Tsiftes et al. [44] evaluated seven compression algorithms with respect to compression ratio and performance overhead . It was found out that GZIP compression could reduce the update size by 44%. The dissemination speed was improved by 67% and the energy consumption was reduced by 69% in a multi-hop setting.

1.2.2.2 Methods for improving dissemination efficiency

A dissemination protocol is the top-level communication component of a remote programming mechanism. It is responsible for the delivery of software updates within the WSN. Besides a rapid distribution, its main task is to ensure the software update to reach all sensor nodes. Dissemination protocols need to operate over the wireless links provided by sensor nodes. Wireless sensor links are typically unreliable and provide only limited transfer capacities. Hence, dissemination protocols must implement specific methods to ensure an efficient and robust operation despite these limitations. Methods broadly used to achieve dissemination goals can be divided into three main categories: reliability measures against packet losses, dissemination policies for reducing packet collisions, and, finally, methods for improving the dissemination speed. Figure 1.5 provides an overview of the commonly used approaches that are described in the following.

1.2.2.2.1 Reliability mechanisms Software updates are prone to packet losses or corruptions due to the unreliable wireless channels. Hence, many packets may get lost or corrupted before reaching the sensor nodes. Several reliability mechanisms have been proposed to cope with this problem. Apart from the implementation details, they are mainly based on packet retransmissions or forward error correction codes (also known as channel coding).

Packet retransmission: The only way to obtain a missing packet in the absence of a mechanism, that allows its reconstruction from the already received ones, is its retransmission. This requires an efficient mechanism to detect packet losses and corruptions. Once a missing or an invalid packet is detected, the receiver notifies the sender for its retransmission by sending a notification message. The notification messages can be ACK-based or NACK-based.

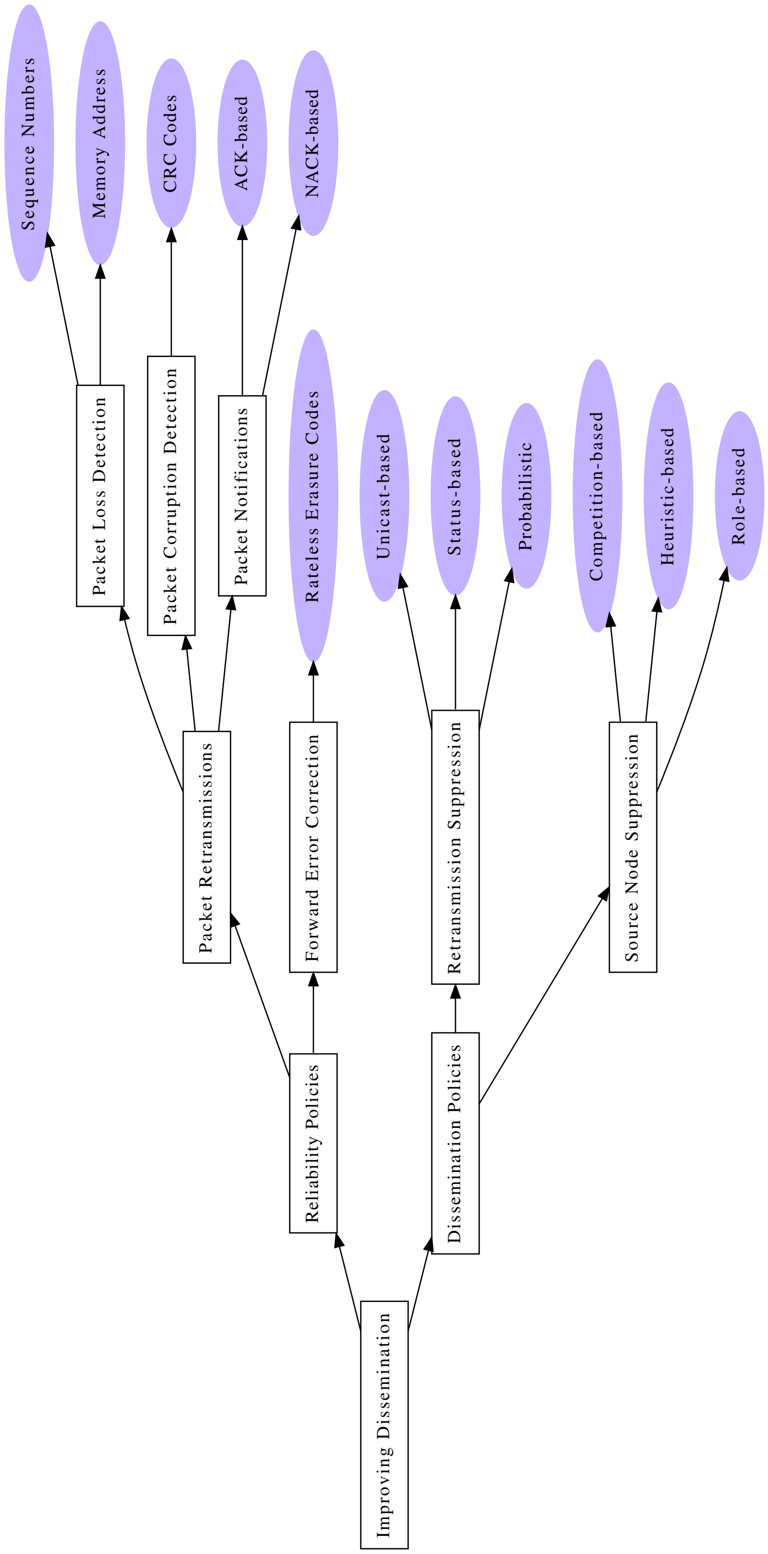


Figure 1.5: Methods for improving dissemination - Common methods used in remote programming protocols for improving the efficiency of the code dissemination.

- **Packet loss detection:** The receivers detect a packet loss using the information sent along with the packets. This information can be e.g., a sequence number or an address range. For instance, the update scripts for the incremental programming protocols [32, 33, 38] specify the addresses where the new software should be written in the memory. The receivers detect the missing packets by checking the address information sent in the update scripts. Some programming protocols append each packet a unique sequence number [45, 46, 47]. The receivers keep track of the missing packets by checking the sequence numbers of the received packets.

- **Packet corruption detection:** A common approach used for detecting a corrupt packet is Cyclic Redundancy Check (CRC). Each packet carries a fixed checksum value that is computed using a CRC algortihm. The receivers detect any error during the transmission by verifying those checksum values. CRCs are computed using a bit vector represented as a binary polynomial [7]. For producing k-bit checksums, a bit vector of length of $k + 1$ needs to be used [7]. It is often called generator polynomial and denoted by $g(x)$ [7]. A k-checksum of an arbitrary-length data is computed as follows: suppose that data is t bits long. Then, it is represented as a binary polynomial $d(x)$ of degree $t - 1$. The k-bit checksum $c(x)$ is the reminder of the polynomial division, $(x^k \cdot d(x))/g(x)$. Deluge [46] uses 16-bit CRC checksums for detecting packet corruptions.

- **Packet notifications:** Some packets may get lost or corrupted during the dissemination. In both cases, the receivers need to send a notification message to notify the senders about missing packets. The notifications can be ACK-based or NACK-based. In ACK-based methods, receivers send an ACK message for every packet that they received correctly to the senders. Senders keep track of non-ACKed packets and retransmit them at some point. In NACK-based methods, receivers are themselves responsible for keeping track of missing packets. They send a NACK message to the senders for every packet that they did not receive correctly. Senders retransmit the requested packets. The time of the retransmission depends on the design in both methods.

  The large number of ACK messages that need to be sent is the main drawback of ACK-based methods. Their main benefit is that acknowledged packets can be removed from the sender's buffer. Sender nodes place the packet buffers on the RAM. As RAM is typically very limited, the packet buffer size is also limited. Dynamically replacing the acknowledged packets with the new ones allows to apply e.g., pipelining more effectively to speed up the programming process.

The main advantage of the NACK-based methods is that the number of notification messages that need to be sent is smaller than the number of ACKs. Its main disadvantage is that senders need to keep all packets in their buffer until the NACKs from all receivers are obtained. This limits their flexibility in applying e.g. pipelining for improving the dissemination performance. Moreover, any NACK received for a packet which is not available on the packet buffer anymore requires to retrieve that packet from the external memory. Access to an external memory on sensor nodes is slower than RAM and much more expensive in terms of energy consumption.

Depending on the application specific requirements, one method can be superior to an other one. A careful analysis needs to be done for selecting the right method for the application scenario. In general, NACK-based methods scale better in remote programming applications than ACK-based methods due to the smaller number of notification messages that need to be sent [45, 46, 47].

Forward error correction: Forward error correction codes (FECs) [48], which are also known as channel coding, allow to reconstruct the missing packets from the already received ones. The main idea of a FEC is to encode the packets using an error-correction code and send them in a redundant way. The amount of redundancy required depends on the channel quality, e.g. the packet loss rate. Any transmission error that occurred within the assumed loss rate can be corrected with a high probability if a sufficient number of redundant packet is received. However, FEC becomes impractical if the channel quality varies and the amount of the redundancy needs to be increased. Once the redundancy rate is fixed, FECs do not allow for further encodings. This is the main limitation of the error-correction codes for using in lossy environments with varying channel quality.

Rateless erasure codes [49, 50, 51], which are also known as Fountain codes, were introduced to solve this problem. They allow for producing an unlimited number of packet encodings. The number of encoded packets required for a successful decoding is an asymptotically fixed ratio of the source packets [49]. Hence, given a sufficient number of encoded packets, the receiver can always recover any packet that was lost.

Fountain codes are particularly beneficial in situations, like remote programming, where the same data needs to be broadcast to multiple receivers. It is not relevant in which order the packets are received and which packets are lost. Moreover, no acknowledgment message needs to be sent to the sender for requesting packet retransmissions. Receivers only need to collect a sufficient number of encoded packets to recover the software update.

Due to their advantages, fountain codes are used in many recent remote programming protocols. Synapse [11] is based on rateless LT-codes [49]. Packets are encoded using XOR operations. It applies a Gaussian elimination method for decoding. An extension of Synapse with multihop support is presented in [52]. Rateless Deluge [53] applies random linear codes for encoding. Decoding is performed using Gaussian elimination method with back substitution. Adapcode [54] considers the link quality in encoding process for load balancing. Encoding is performed by linearly combining the source packets. Decoding relies on the Gaussian elimination method. Another approach based on online-codes is proposed in [55].

1.2.2.2.2 Dissemination policies The dissemination model being used is of a great importance for the efficiency of a remote programming. Consider a programming protocol based on the classical flooding: all sensor nodes are the senders and receivers at the same time. Receivers forward data they obtained to other sensor nodes by broadcast. Retransmission requests are flooded without addressing a specific receiver. Such a broadcast storm would result in many redundant and concurrent transmissions. This would possibly cause a large amount of packet collisions and contentions that waste energy and resource [56].

Remote programming protocols mitigate the broadcast storm problem by applying suppression mechanisms. Suppression mechanisms are implemented on the senders and can be broadly classified into two domains: suppression of the retransmissions of the missing packets and suppression of the advertisements. The basic idea behind any suppression mechanism is to reduce packet collisions by preventing concurrent and duplicate transmissions.

Retransmission suppression When a receiver node detects a packet loss, it requests for its retransmission. Retransmission requests can be broadcast or unicast. Unicast requests are addressed to specific senders only. Senders reply to the requests, that are sent to them, by retransmitting the desired packets. The main drawback of a unicast-based request policy is its limited reliability. Requests must reach the senders that they are sent to. Otherwise, they are simply ignored.

Broadcast of requests naturally increases the chance of getting the desired packets, since any sender possessing the requested packets can potentially retransmit them. However, this may also lead to the broadcast storm problem. Multiple senders retransmit the same packet at the same time when a broadcast request is received. This increases the number of duplicated replies which in turn increases the number of the packet collisions. Hence, a suppression method is required to stop multiple replies to the same request.

Suppression mechanisms aim to reduce the number of senders that reply the same retransmission request. Some of the methods proposed to achieve this goal are described in the following [45]:

- **Unicast retransmission requests:** As unicast requests are replied by a single source only, they are not subject to concurrent and duplicated retransmissions.
- **Status based retransmissions:** Senders reply only the retransmission requests which are not replied by anyone else [57].
- **Probabilistic retransmissions:** Senders reply a request according to a probability distribution. The probability distribution can be static or adaptive. In the latter case, senders adopt their probability for replying a request according to some metrics like neighborhood size and the number of the senders.

Source node suppression Remote programming protocols start the dissemination of a new software with an advertisement message. The advertisement message contains the summary of the new software like the version number and the size of the software. Depending on the information received in the advertisement packets, sensor nodes decide to accept the new software or not. In a multi-hop WSN, some of the sensor nodes at each hop become the source for the nodes of the following hops to ensure the propagation of the update to the entire network. For this reason, some sensor nodes make the update they received available for download for other nodes by sending advertisement messages. Allowing all sensor nodes to become a source node would result in the broadcast storm problem. Therefore, a suppression mechanism is required to allow only a selective set of nodes to become a source node.

Several suppression mechanisms have been proposed to reduce the number of source nodes. They are mainly based on the publish-subscribe principle. That is, once a sensor node has the complete software update, it advertises the update by sending a publish message. Sensor nodes receiving the advertisement check their version number and send a subscribe message to the publishing node to register their interest. Publishers become the source node for their subscribers. The mechanisms differ mainly in determining the publishers and the source nodes:

- **Competition-based methods:** Sensor nodes compete to become a source for forwarding the update to other sensor nodes.

  Ripple [45] is a dissemination protocol belonging to this category. All sensor nodes that have the entire software update try to become a sender. For this, they advertise the availability of the new version by broadcasting a publish message. Sensor nodes which received an advertisement message check their version number and send a subscribe message to the publishing node to register their interest. If there are multiple publishers, sensor nodes choose the one with the best link quality as the source to download the software update from it. Source nodes disseminate the update to their subscribers. Subscribers send their packet retransmission requests using unicast and wait for the response. If the response is not received after a certain amount of time, they request the missing packets again by using broadcast this time. All sources that received a broadcast request reply with the requested packet. Subscribers choose a new source based on the link quality.

  Multihop Network Programming (MNP) [47] is another dissemination protocol in which the sensor nodes compete to become a source node. It can be seen as a simple improvement of Ripple [45]. The improvement is achieved by considering the number of subscribers while selecting a source node. Publishers with the highest number of subscribers are elected as the source node. Firecracker [58] is another example that belongs to this category. It differs from the previous approaches in two ways. Firstly, the source nodes are selected by the base station. Secondly, this approach uses a routing protocol to transfer the update to the source nodes. To become a source node, sensor nodes report the number of neighbors they have to the base station. The base station selects the nodes with most neighbors as a source node. For the dissemination, the base station delivers the software update to the source nodes using a routing protocol. Subsequently, source nodes disseminate the received data to their neighbors by broadcast.

- **Heuristic-based methods:** Heuristic approaches are used to optimize the number of source nodes.

  Trickle [46, 59] is a dissemination protocol that belongs to this category. In this approach, all sensor nodes can become a source node. The probability of becoming a source node is theoretically equal for all nodes. This is achieved by using the following heuristic. All sensor nodes send an advertisement message to announce the update available on them. However, the number of advertisements that a sensor node can send is limited. Firstly, sensor nodes send an advertisement only if they hear no identical advertisement. Secondly, the advertisements are sent at random intervals. These two requirements ensure that a software update is advertised by only one sensor node in each interval even if there are multiple candidates. Since there will be only one advertiser in each interval, there will be only one source node. Receivers send their packet retransmission requests to the source node using unicast.

- **Role-based methods:** In role-based approaches, sensor nodes become a source node or a receiver node depending on the role assigned to them.

  Time Division Multiple Access (TDMA) [60] introduces the concept of predecessors and successors to suppress the number of source nodes. Each sensor node downloads an update from its predecessors only. Similarly, each sensor node forwards an update to its successors only. For this reason, each sensor node selects some of the senders as its successors and some of the receivers as its predecessors. This classification depends on the interaction frequency and it is done as follows: each node sets a sender node, from which it received many packets, as its predecessor. Similarly, each node sets a receiver node, to which it sends many packets, as its successor. Every node may have multiple successors and predecessors. Once sensor nodes have classified their neighbors, they listen only to their predecessors as the source node for downloading the updates. Each node forwards the received updates to its successors only. Allowing only the predecessors to become a source node reduces the number of the senders.

Another role-based approach is Sprinkler [61]. Sprinkler introduces two new roles: cluster heads and cluster members. For assigning a role to each sensor node, the WSN is firstly divided into clusters. For each cluster, a sensor node is selected as the cluster head using a connected dominating set (CNS) algorithm. Remaining sensor nodes become the cluster members. Clustering is done such that every sensor node (member) can connect to at least one cluster head. Once the cluster heads are computed, each member node listens to the update from its cluster head only. Software updates are disseminated only by the Cluster heads. As Sprinkler allows only the cluster heads to disseminate the updates, the number of source nodes is suppressed.

1.2.2.3 Methods for improving performance

Figure 1.6 provides an overview of the approaches commonly used for improving the dissemination speed of a remote programming.

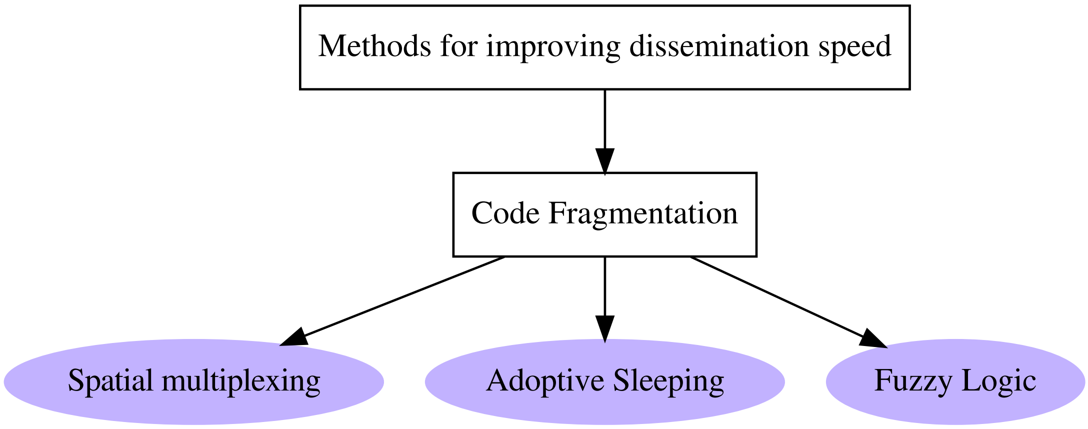


Figure 1.6: Methods for improving dissemination speed - Common methods used in remote programming protocols for improving the dissemination speed.

Software updates usually do not fit the RAM available on the sensor nodes. Hence, they need to be fragmented into small pieces before the transmission. This allows to store only the part of the update being disseminated in the RAM. Other parts of the update are stored in the external memory and picked up when needed for the transmission. The code part stored in the RAM is disseminated packet by packet to the other nodes in the network. This process is called code fragmentation and is applied in all modern remote programming protocols.

A simple approach used for disseminating a fragmented update is the sequential propagation. That is, each node disseminates an update further to other nodes after receiving all fragments completely. For example, Multihop Over-the-Air Programming (MOAP) [45] follows this approach. However, this approach performs badly in multi-hop networks since it prohibits the use of spatial multiplexing. Spatial multiplexing [46] allows the dissemination of the received segments immediately before receiving the entire image. This speeds up the dissemination process significantly by increasing the throughput of the network. However, since any sensor node can be the source of the dissemination, all nodes need to be awake during a software update. This increases the overall energy consumption within the network. In order to mitigate this problem, MNP [47] introduced an adaptive sleeping approach. This approach puts sensor nodes into the sleeping mode during the dissemination of code segments which are out of interest.

A different approach based on fuzzy logic for improving the dissemination efficiency is proposed in [62]. In this approach, all sensor nodes start to broadcast a software update with full speed at the beginning of a dissemination process. To mitigate the broadcast storm problem, means of fuzzy control are used. That is, each sensor node snoops the communication in its vicinity to estimate the packet collision rate. Using this information and a fuzzy control logic, sensor nodes dynamically optimize its transmission rate in accordance with the local congestion of the radio channel.

1.2.3 Applications

Remote programming is an important tool needed for maintaining and managing WSNs. It is required for rapid software development, prototyping and testing. Furthermore, it is required by the WSN owners for rapid application replacements and feature upgrades. A nice overview of the possible use cases is given in [63].

Software development and testing Software development is an iterative process. Before a software application is deployed in a real WSN, its quality needs to be ensured. This requires to upload the developed software on a sensor testbed and check its behavior under multiple test cases. The bugs found in the software are removed and the debugged software is deployed on the testbed again for running further test cases. This process is repeated until the application passes all tests. Instead of reprogramming each sensor node one by one, remote programming allows to upload the new version of the software to the entire testbed via the wireless channel. This obviously shortens the development and testing phase of WSN applications significantly.

Bug fixing The behavior of a software in a real WSN deployment differs from its behavior on a testbed deployment. This is mainly due to the events caused by indeterministic environmental or physical conditions that can not be simulated in a laboratory ambiance. Hence, some bugs are likely to remain still undiscovered in the software despite of the intensive software testing performed before its deployment. Remote programming speeds up the dissemination of the new software versions for bug fixing in real WSN deployments.

Configuration updates Varying environmental conditions such as the large temperature difference between the summer and winter may cause the software application to stop working properly. For example, the wireless communication in a WSN deployed on the Swiss Alps worked well during the day but failed during the nights [64]. The reason was the failure in the clock drift that was caused by the big temperature difference between the day and the night time at the Swiss Alps. Remote programming can be used to transmit configuration updates to sensor nodes to tolerate such failures in extreme environmental conditions.

Application fine-tuning During the life time of a WSN, the requirements on the software application may change. The required changes can range from parameter updates for sensing frequency to adding new functions to the existing software. Remote programming can ease the application fine-tunings.

Application replacement Remote reprogramming is crucial in scenarios where multiple applications need to be run on the same WSN. Sensor nodes are typically equipped with limited hardware resources such as memory and computation power. Pre-installation of multiple applications on the resource limited scarce sensor nodes is usually not possible. In such cases, new applications can be uploaded to sensor nodes via remote programming.

Security updates WSNs are ideally deployed for long periods of time. During the lifetime of a WSN, new security weaknesses can be discovered in the software deployed on the sensor nodes. Security patches via software updates need to be applied to the existing software to keep the WSN secure against adversaries in such cases. Hence, remote programming is crucial for fixing security problems in WSNs.

1.3 Secure remote programming

WSNs are often deployed for long-term monitoring in environments which are possibly public and even hostile. Hence, adversaries can easily access the WSNs. The form of access ranges from eavesdropping and tampering with the wireless communication to physically possession of sensor nodes. Software updates may contain key material to be used for securing the operations and communication within the WSN. An adversary eavesdropping the remote programming can easily obtain such a confidential key material stored in the software update and compromise the security of the entire WSN. Similarly, adversaries can modify the software being disseminated or even replace it with a malicious one by tampering with the wireless communication. Hence, remote reprogramming must be protected against the adversarial interferences. This requires the development of security mechanisms which make remote programming remain secure in open and even hostile environments. However, this is a challenging task in particular due to the following reasons.

First of all, the open nature of WSNs provides adversaries with a rich set of means for implementing their attacks. In general, attacks on remote programming can be implemented merely by snooping the sensor communication or tampering with it. This is easily possible using only off-the-shelf tools and readily available free software. The low investment costs required for performing the attacks and the high impact in case of success make remote programming an attractive target for adversaries.

Secondly, the means available on the sensor nodes for implementing possible counter measures are typically very limited. Sensor nodes provide a very scarce computation, communication, and storage capacity. For example, a Telos node [25], a widely used sensor platform in WSNs, has only 10KB of data memory (RAM), 48KB of program memory (Flash memory), and 16KB of non-volatile memory (EEPROM). Hence, security measures need to be realizable with very limited resources. This requires a careful design that trades off the security and the resource consumption properly.

Finally, resource limitation of sensor nodes prevents the use of protocols developed for traditional computer networks in remote programming. New protocols, that are tailored for remote programming, need to be developed. Considering the multitude of possible attacks and platforms, this is obviously a non-trivial task.

By taking the observations above into account, requirements for an ideal secure remote programming can be considered in two groups: security requirements and implementation requirements.

1.3.1 Security requirements

Security mechanisms are required for protecting remote programming protocols against adversarial interferences. Requirements that an ideal security mechanism must satisfy depend on the concrete attack vector it is designed for. According to their characteristics, attacks for remote programming can be divided into two main categories: attacks on the communication medium and attacks on the sensor node.

Depending on the adversary's goal, attacks on the communication medium can be further divided into two subcategories: attacks on the software update itself and attacks on the sensor resources. The goal of the adversary in the former case is to modify the software being disseminated. The modification can be a complete replacement of the software with a malicious one as well as a partial replacement of some components or even some program lines. The adversary is assumed to be successful if it can change the behavior of the application running in a WSN by exploiting a remote programming mechanism. The goal of the adversary in the latter case is to put a WSN out of service. This can be achieved by depleting the energy of sensor nodes or blocking their communication. Jamming [65, 66] is a conventional mean for blocking the sensor communication. A more fine-grained kind of jamming is called selective jamming. Selective jamming attacks allow the adversary to target at specific packets of high importance [67]. In general, jamming attacks are applicable to any WSN communication. They are not specific to the remote programming protocols. This thesis considers the attacks that make use of the remote programming protocol for wasting resources of sensor nodes. The adversary is assumed to be successful if it can make sensor nodes to waste their resources by exploiting a remote programming mechanism.

Finally, node compromise attacks [68] are another attack category that needs to be considered. The goal of the adversary is to obtain the cryptographic material stored on the sensor nodes. Therefore, node compromise attacks can be seen as a preparation step for other attacks. Once the cryptographic material is revealed, the adversary uses it for performing malicious activities typically without being detected. Contrary to the previous attacks, node compromise attacks require physical access to the sensor nodes. This presents a certain level of challenge to the adversary. However, WSNs may be deployed in public environments where the adversary can have a free access to the sensor nodes. As demonstrated in [68], the key material stored on a standard sensor platform can be extracted in less than a minute. Hence, node compromise attacks need to be considered particularly in public WSN deployments. The adversary is assumed to be successful if it can break the security of a remote programming mechanism using the cryptographic material obtained from node compromises.

Figure 1.7 gives an overview of the attacks identified and their countermeasures that are described in the following:

1.3.1.1 Attacks on the communication channel

The adversary tampers with the sensor communication to achieve its goal. The attack target can be the software update itself or the network resources.

Attacks on the software The goal of the adversary is to change the application logic. It achieves this by replacing the software being disseminated either completely or partly. Moreover, attacks violating the privacy are considered in this category. Security mechanisms mitigating this kind of attack should have the following properties:

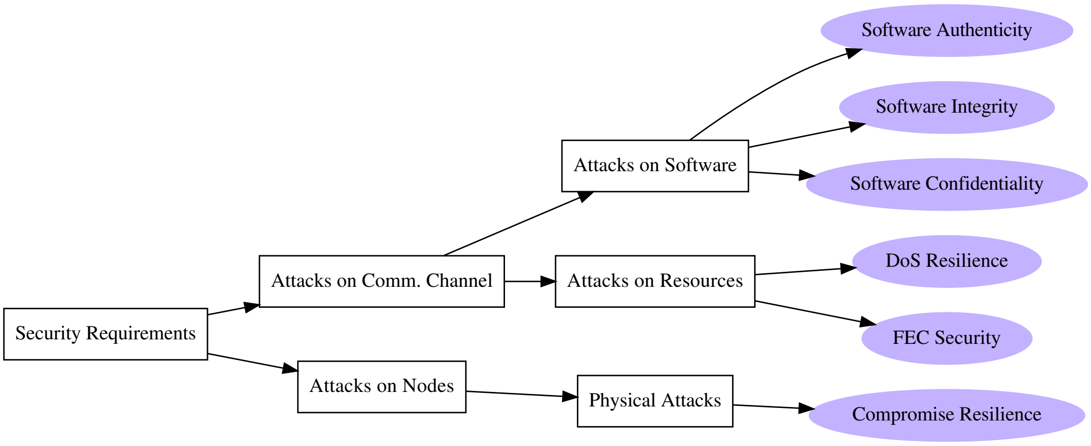


Figure 1.7: Security requirements on secure remote programming - Classification of the security requirements according to high-level attack vectors, their relations to each other and the possible counter measures.

- **Software authenticity** In order to mitigate a complete or partial software replacement, sensor nodes must be protected from running the code from unauthorized sources. This can be achieved via software authentication. Hence, an efficient security mechanism for authenticating the source of the update is required. This allows sensor nodes to verify the authenticity of the updates and accept them only from a trusted origin, e.g. from the gateway node.

- **Software integrity** In order to mitigate modifications on the software during its dissemination, the integrity of the software must be protected. Even though the software integrity is implied by the software authenticity, a separate mechanism for integrity protection is required due to efficiency reasons. Software authenticity would ensure the integrity of a complete update. Thus, flipping even a single bit in an update would require the retransmission of the complete update. This would be very costly for WSNs. For this reason, an efficient mechanism for detecting the integrity violations at a finer granularity (e.g., at packet level) is desired. This allows sensor nodes to accept only the integer packets and requires the retransmission of non-integer packets only.

- **Software confidentiality** Software itself or the information that it carries might be sensitive. A commercial software might be an example of the former case. Software updates carrying cryptographic material might be an example of the latter case. In both cases, confidentiality of the software update must be protected against unauthorized entities. Hence, an efficient security mechanism that prevents eavesdroppers from gaining information on the update being disseminated is desired.

Attacks on the sensor resources The goal of the adversary is to waste sensor resources, for example to shorten the lifetime of a WSN. The adversary achieves its goal by tampering with the communication. It tries to bring the dissemination process in an endless packet send/discard loop by modifying the packets randomly or selectively. Security mechanisms for mitigating such attacks should have the following properties:

- **Denial of Service protection** Remote programming is an energy intensive process requiring dissemination of a large amount of data through a multihop WSN. As the update is transmitted via wireless links, the adversary may corrupt packets during their propagation. In such a situation, the corrupted packets need to be retransmitted to fix the update. However, this is possible only if they are known. Otherwise, the complete image needs to be retransmitted. The adversary can make use of exactly this behavior to perform its attack. It can bring a remote programming process in an endless send/discard loop by tampering with some packets during their dissemination. As the data transmission is a very energy consuming operation, sensor nodes' energy is depleted sooner. This type of attacks, aiming to put a WSN out of service sooner than planned, are referred as Denial of Service (DoS). In order to mitigate DoS attacks, an efficient security mechanism[1] for detecting exactly those packets which have been corrupted is required. This allows the sensor nodes to repair only corrupted packets and avoids unnecessary energy consumption due to their propagation over multiple hops.

- **FEC security** In order to increase the reliability of a remote programming, advanced coding techniques such as Fountain codes are used (see Section 1.2.2.2.1). Such coding techniques are particularly beneficial in environments where the packet loss rate is high and a large data stream needs to be disseminated to a large number of receivers. The main idea is to enable the sensor nodes to

[1] A mechanism whose cost for providing DoS security is lower than the cost of the attack it mitigates.

deduce missing packets from the packets received without explicitly asking for their retransmission. However, such coding techniques cause new threats, namely poisoning attacks, a kind of DoS attack. That is, corruption of a single packet spreads through the entire image due to the encoding. For this reason, remote programming protocols based on coding are particularly prone to DoS attacks. In order to mitigate this kind of attacks, efficient security mechanisms that support coding techniques are required. They prevent the software updates from poisoning attacks and avoid unnecessary energy consumption due to the propagation of invalid packets over multiple hops.

1.3.1.2 Attacks on the sensor nodes

- **Physical attacks** Adversaries can tamper with the sensor nodes physically to obtain the cryptographic material stored on them in case of the deployments in public and hostile environments. Hence, an ideal security mechanism must prevent the adversaries from being able to update non-compromised sensor nodes with a malicious software even if a large number of nodes get compromised.

1.3.2 Implementation requirements

A secure remote programming needs to be applicable on sensor nodes. Hence, security mechanisms must demand as little as possible from the available resources so that sensor nodes can still perform their main tasks. As depicted in Figure 1.8, the following implementation requirements are identified:

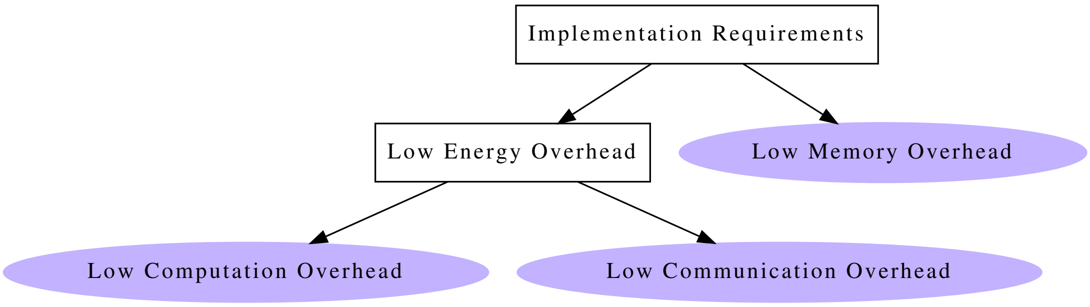


Figure 1.8: Implementation requirements on secure remote programming - Classification of the implementation requirements according to the sensor specific resources and their relations to each other.

Low communication overhead

WSNs are desired to be functional for a long time after their deployment. The length of this duration depends on how long sensor nodes' storage and energy resources remain available without being depleted. The most energy consuming operation on a sensor node is the radio communication (sending as well as receiving). For example, sending a single bit of data consumes the energy of thousands of CPU cycles [25]. Hence, security mechanisms for remote programming protocols must not introduce too much communication overhead. This mandates that cryptographic primitives to be used in security mechanisms have to be chosen carefully. They need to be light-weight and the communication overhead caused by them must constitute only a small percentage of the total communication. Generally speaking, asymmetric cryptography requires more bits to be transmitted than symmetric cryptography for the same level of security. Therefore, an ideal security solution for a remote programming should mainly rely on symmetric primitives. Asymmetric cryptography has to be used only if the required security cannot be realized by using symmetric cryptography.

Low computational overhead

In order to keep the energy consumption low, security mechanisms must demand only a moderate amount of CPU cycles. In general, asymmetric cryptography is much more CPU expensive than symmetric cryptography for the same level of security. Therefore, an ideal security solution for a remote programming should mainly rely on symmetric primitives. Asymmetric cryptography has to be used only if the required security primitive cannot be realized by using symmetric cryptography.

Low memory overhead

During a secure remote programming, cryptographic material need to be processed on RAM. Sensor nodes have only a limited amount of RAM. Therefore, cryptographic primitives chosen for security must be efficiently executable even in a very limited RAM. Additionally, the available code space (ROM) must be shared among the actual application and the security mechanisms. Hence, security implementations must fit on the sensor node's program memory (ROM) and occupy only a moderate amount of it. For this reason, symmetric primitives are typically more suitable than asymmetric ones.

1.4 Contributions of this thesis

This thesis introduces several mechanisms for meeting the security requirements identified in the previous section. They are briefly as follows:

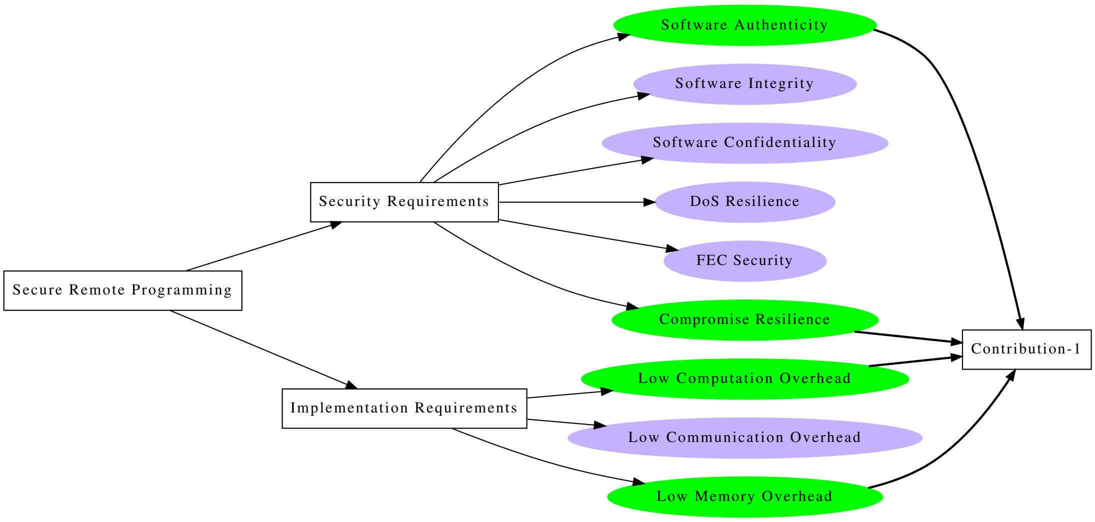


Figure 1.9: Contributions for software authenticity - $\mathcal{T}$-time Signature is a hash based signature scheme. It is resilient against node compromise attacks and demands only a moderate level of computation power and storage.

Contribution 1: A signature mechanism for software authentication The contribution of this thesis in terms of software authenticity is summarized in Figure 1.9. Software authentication in a remote programming is achieved by signing the software update using a signature mechanism. However, due to the limited program memory (ROM) available on the sensor nodes, implementation of a conventional public-key base signature mechanism is a challenging task. For instance, the implementation of the elliptic curve digital signature scheme occupies 28.1% (13.5KB) of the available ROM [69] on a typical sensor platform. Even more efficient signature schemes such as RSA, NTRUSign, and XTR-DSA still introduce 7.4KB, 11.3KB, and 24.3KB ROM overhead [70]. Such a large memory footprint for a signature scheme prohibits a shared implementation of the application and the software update mechanism without cutting back on functionality or security.

This thesis presents a *stateful-verifier* $\mathcal{T}$-time signature scheme to address this problem. It is built on Merkle's one-time signature scheme [71] and requires only a collision-resistant hash function for the implementation. The implementation code occupies only 2KB of ROM. This is a tremendous improvement compared to the memory requirements of conventional signature mechanisms. As the signature computations requires only hash operations, it is computationally more efficient than the conventional signature mechanisms as well. Finally, it is secure against node compromise attacks. The adversary needs to break the underlying hash function for a signature forgery. However, this is computationally not feasible if a hash function like SHA-256 is used.

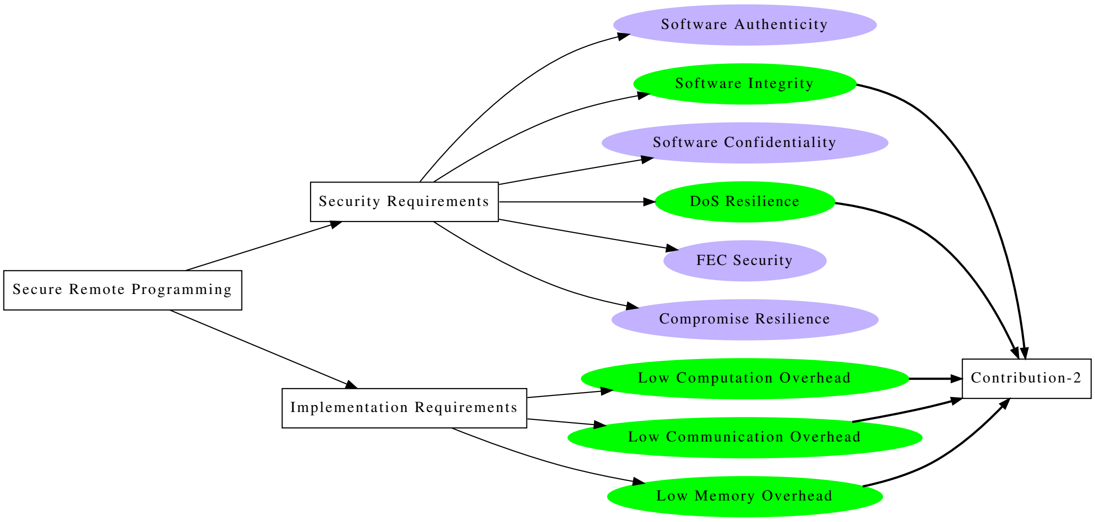


Figure 1.10: Contributions for software integrity - Security of the $\mathcal{T}$-time signature is extended via multiple-hash chains through the entire software update. This ensures a computation and memory efficient integrity protection. Integrity verifications at multiple granularities allow for an optimal DoS protection.

Contribution 2: An efficient mechanism for software integrity The contribution of this thesis in terms of software integrity is summarized in Figure 1.10. Software integrity is protected using an approach similar to the one that was originally proposed by Gennaro et al. [12]. The basic idea is to use only one digital signature over the very first element of a hash chain that includes all packets of the software update. Hence, this approach amortizes the computation and communication overhead of the digital signature by combining it with cheap hash computations over several packets. All previous secure remote programming protocols [72, 73, 74, 75] are also based on this idea. They differ mainly in terms of the number of hash chains used, the granularity of data considered in the hash chains, and the amount of flexibility in verifying packets when they arrive out-of-order.

The approach described in this thesis differs from the previous solutions in three aspects. Firstly, it exploits the stateful-verifier $\mathcal{T}$-time signature scheme to minimize the memory usage and computation overhead. The program memory overhead is only 480B. Secondly, it defeats DoS attacks more efficiently than the previous solutions. This is achieved by applying multiple hash chains at multiple granularities (at packet and page levels). Finally, using multiple-hash chains enables a fine-grained control over the security degree applied at packet and page level. This allows for a better trade-off between the communication overhead and the security level obtained.

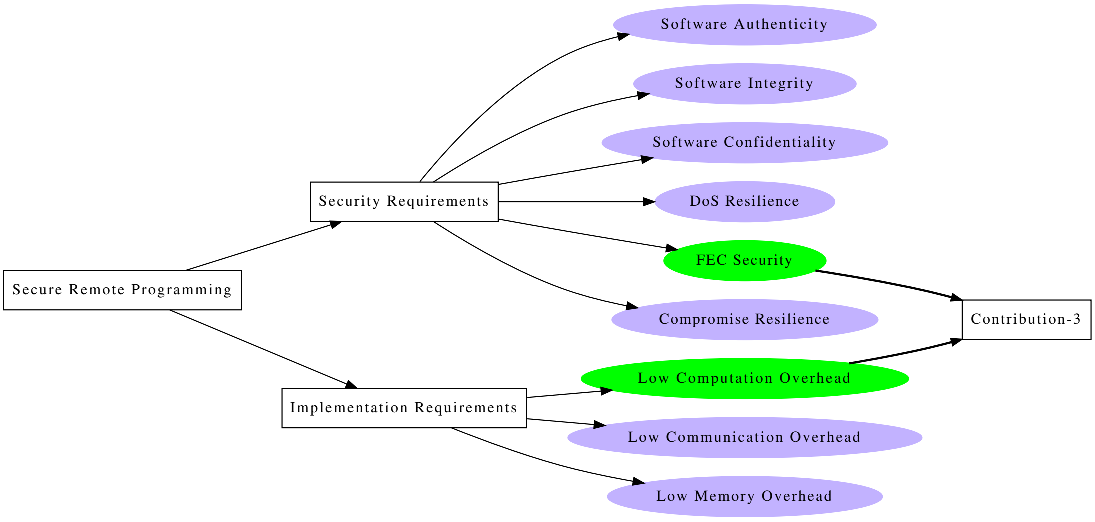


Figure 1.11: Contributions for FEC security - Integrity of the encoded packets is checked nearly on-the-fly. This allows to detect a corrupted packet immediately after reception and prevents it from further propagations.

Contribution 3: An efficient mechanism for FEC security The contribution of this thesis in terms of rateless erasure codes is summarized in Figure 1.11. Remote programming protocols based on rateless erasure codes, like Fountain codes, are vulnerable to poisoning attacks. That is, the modification of a single packet leads to the corruption of the whole programming process. This thesis describes a security extension for Fountain code-based remote programming protocols. It mitigates the poisoning (DoS) attacks by verifying the encoded packets (nearly) on-the-fly. The computation overhead is very low as it requires only XOR operations and hash computations.

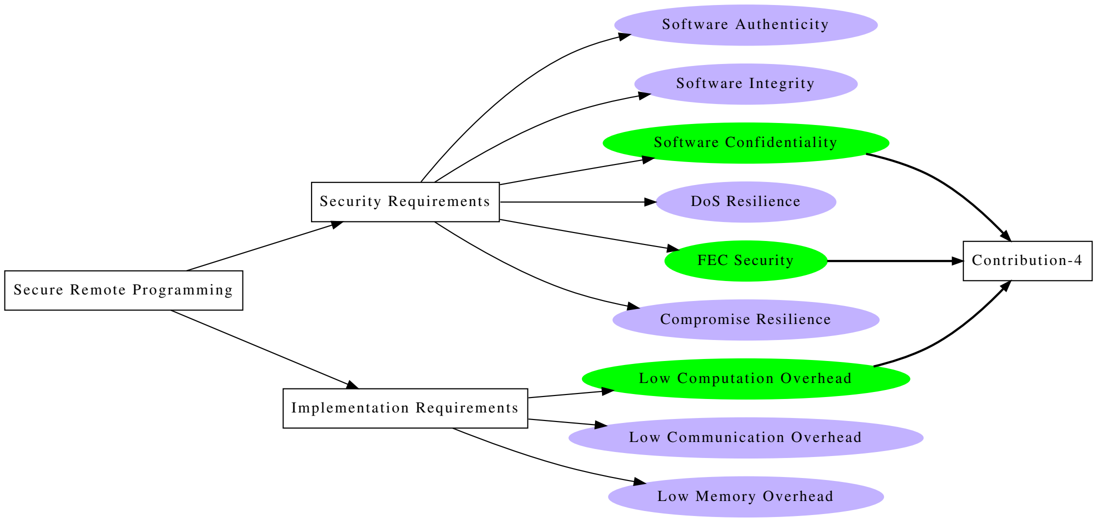


Figure 1.12: Contributions for software confidentiality - Software confidentiality is achieved by encrypting the packets using a symmetric encryption. Encryptions and decryptions are performed using a hardware implementation provided by the radio module. The memory and computation overhead of the confidentiality protection is kept at minimum.

Contribution - 4: An efficient mechanism for software confidentiality The contribution of this thesis in terms of software confidentiality is summarized in Figure 1.12. A security extension for Fountain-code based remote programming protocols for confidentiality protection is described. It is based on the end-to-end software encryption. The memory and computation overheads are kept at minimum by using the encryption component provided in the radio module.

2

Materials & Methods and Impacts to Design

This section firstly provides a short overview of the general characteristics of the hardware and software platforms used in WSNs. Subsequently, the hardware and software platforms used for the implementation of the security mechanisms introduced in this thesis are described. After that the methods, programming styles and algorithm choices that can be used for optimizing the resource consumptions are described. Finally, design choices made in this thesis for optimizing the resources consumed by security mechanisms for remote programming are described.

2.1 Hardware platform

WSNs are composed of multiple sensor nodes which are used for performing sensing, storing, processing, and communication tasks. Sensor nodes are desired to be cheap for allowing large scale deployments at low costs. This key requirement prohibits the use of expensive technologies and rich resources on them. Hence, sensor nodes are usually equipped with very scarce resources for performing pretty complex tasks.

In the following, firstly, the typical hardware components of a sensor node and their general characteristics in terms of resources are described. Subsequently, the sensor platform used for the implementations in this thesis is introduced.

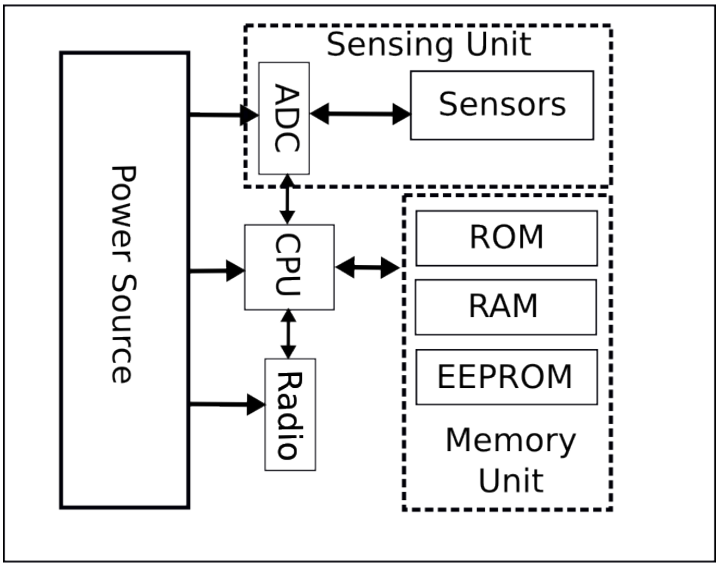


Figure 2.1: Generic hardware architecture for a sensor node (adapted from [6]) - A typical sensor node is composed of a power source, a sensing unit, a microcontroller, a radio transceiver, and a memory unit.

2.1.1 General characteristics

As shown in Figure 2.1, a typical sensor node is composed of a sensing unit, a memory unit, a radio module, and a microcontroller (CPU) [6].

Memory unit As shown in Figure 2.2, sensor nodes are usually equipped with three types of memory: RAM, flash memory (ROM) and EEPROM.

RAM (Random-access Memory) is used as the temporary data memory and working space for the applications and operating system being run on the sensor node. A RAM can be static or dynamic. Compared to a dynamic RAM (DRAM), a static RAM (SRAM) consumes less energy during its operation. For this reason, sensor nodes are typically equipped with an SRAM. Another major advantage of an SRAM is its execution performance. Data in an SRAM is stored on flip-flops. The memory content (i.e., data) is updated by changing the states of the flip-flops. As the states of flip-flops can change very fast, an SRAM is faster than a DRAM where data is stored on transistor-capacitor pairs. The main disadvantage of SRAM is its cost. SRAM is expensive to produce. Therefore, it is usually kept very small on sensor nodes (4KB $\sim$ 10KB).

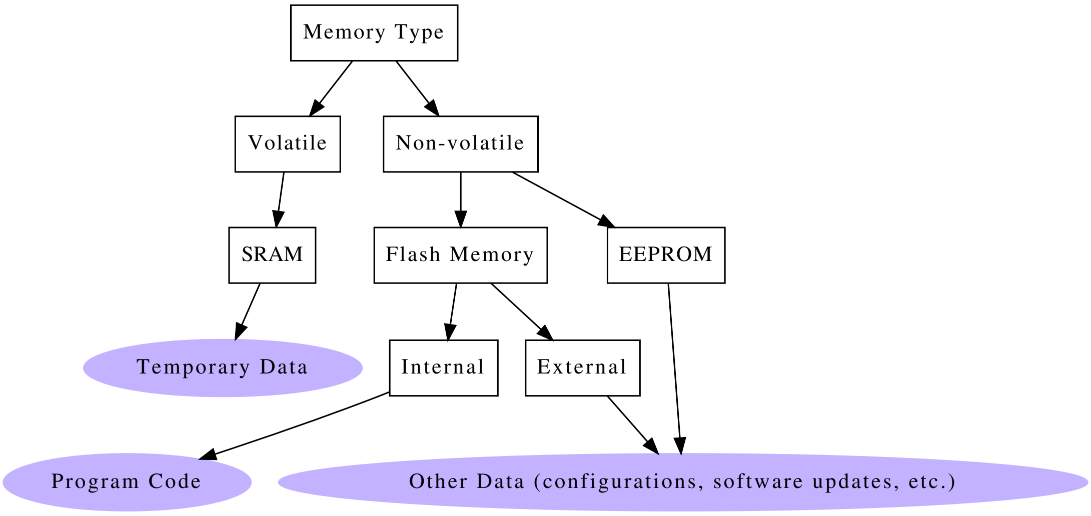


Figure 2.2: Memory components of a sensor node - Typical memory components of a sensor node and their use: SRAM is a temporary data and working space for programs. Internal flash memory is used for storing program code. External flash memory and EEPROM are used for persistent data storage such as calibration data and other data including software updates.

Flash memory is used for the permanent data storage. A sensor node is required to have at least one flash memory for storing the program code of the applications and operating system. Flash memory of this purpose is usually referred as internal flash memory. Typically, an extra flash memory is provided for storing other application data including software updates. Flash memory of this purpose is usually referred as external flash memory. Flash memory is a writable variant of ROM (Read-only Memory). Hence, the memory content can be updated by write and erase operations. However, these operations can only be done in large blocks varying from 256B to 128KB. As a result, compared to SRAM, flash memory is typically very slow in writing and erasing of data. On the other hand, flash memory is less expensive to produce. Sensor nodes are usually equipped with a relatively large flash memory (48KB $\sim$ 1MB).

EEPROM (Electrically Erasable Programmable Read-only Memory) is also a writable variant of ROM. It is generally slower than a flash memory in terms of erasing of data. EEPROM is used for the same purpose as external flash memory. That is the persistent data storage such as calibration information or other data including software updates that needs to be available when the power is removed. Due to the cost reasons, sensor nodes are typically equipped with either an EEPROM or an external flash memory.

Sensing unit A sensing unit is composed of one or multiple sensors and possibly an analog-to-digital converter (ADC) [13]. Sensors are used to measure the physical events being monitored like temperature, movement, pressure, light, etc. A sensor can deliver its measurement in the form of a continuous analog or digital signal. As analog signals cannot be processed by a digital CPU, they are first converted into digital signals by the ADC. Subsequently, the digital signals representing the sensor measurements are fed into the CPU for further processing. If sensors produce digital outputs, no ADC is required. The calibration information of sensors is typically stored on the sensors' onboard EEPROM and can be updated on demand. Depending on how they are powered, sensors are mainly classified into two categories: passive sensors and active sensors [13]. Active sensors require a continuous external power source while performing their sensing tasks. Passive sensors are self powered and obtain the required energy from the phenomena being measured. The use of passive sensors is more often than the use of active sensors in WSNs.

Microcontroller All tasks and computations on a sensor node are executed by a digital processor. Hence, the choice of the processor is of a great importance for the overall performance, energy consumption, and flexibility. Microcontrollers, digital signal processors, application-specific integrated circuits, and field programmable gate arrays are the possible options that can be used as a processing unit on a sensor node [13]. Due to their low energy consumption, easy programming, and flexibility in connecting with external peripherals, microcontrollers are superior to other alternatives. For this reason, microcontrollers are the main choice for many existing sensor nodes today. Microcontrollers for sensor nodes usually operate with 8- or 16-Bit words. They are typically clocked at very low frequencies (4Mhz $\sim$ 8MHz) to minimize the power consumption.

Radio transceiver Radio communication is the most energy consuming operation on a sensor node. Hence, the choice of the radio technology is critical for the overall energy efficiency. The IEEE 802.15.4 radio standard was proposed for wireless personal area networks. The main goal was to standardize a protocol stack that enables short-range communications at a very low cost. Due to its energy efficiency, most of the new generation sensor nodes at present are equipped with an IEEE 802.15.4 compliant radio transceiver. A prominent example of such a transceiver is the Chipcon CC2420 [20]. It uses the 2.4GHz band. Due to its reliability and low power operation, the CC2420 radio is the dominating choice for the sensor nodes. Another IEEE 802.15.4 compliant radio transceiver which was widely used on earlier sensor node generations is the Chipcon CC1000. It operates at the 915MHz band [17].

2.1.2 Implementation hardware

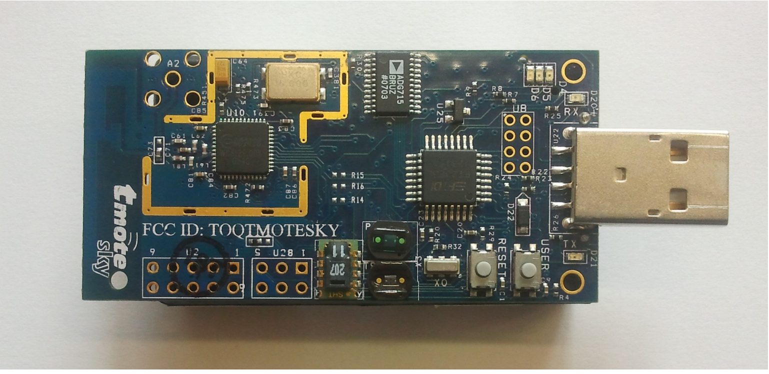


Figure 2.3: TelosB sensor node - A prominent sensor node platform using a 16-bit microcontroller, an IEEE 802.15.4/ZigBee compliant radio, and integrated sensors (temperature, humidity, light).

The sensor node used for the implementations in this thesis is the Telos revision B (TelosB) [76], depicted in Figure 2.3. It uses an ultra low power microcontroller and communication module and is equipped with a humidity, temperature, and light sensor. Due to its comparability with industry standards like USB and IEEE 802.15.4, TelosB is one of the widely used sensor platforms at present. It can be easily programmed using open source software and tools via an USB port. Hence, there is no need for a special board for programming.

The TelosB uses the Chipcon CC2420 [20], IEEE 802.15.4 compliant, ZigBee ready radio transceiver and is powered by 2 AA batteries. It is equipped with a 16-bit TI MSP430 microcontroller which can be clocked at most at 8MHz. The microcontroller is based on the RISC architecture. Thus, the data and program code need to be stored separately in memory (RAM) and flash memory (ROM), respectively. The TelosB node has the following key features [77]:

- Powered by 2 AA batteries
- 8MHz, 16-bit Texas Instruments MSP430 microcontroller
- 48KB in-system programmable flash memory (ROM), 10KB data memory (SRAM), 1024KB external flash memory
- IEEE 802.15.4/ZigBee compliant Chipcon CC2420 Radio Transceiver
- 250kbps data rate and 2400MHz to 2483.5MHz frequency band
- Integrated ADC, DAC, Supply Voltage Supervisor, and DMA Controller
- Integrated onboard antenna with 50m range indoors / 125m range outdoors
- Integrated humidity, temperature, and light sensors
- Ultra low power consumption
- Hardware link-layer encryption (AES-128) and authentication (CBC-MAC)
- Programming and data collection via USB

2.2 Software platform

A sensor node is composed of multiple hardware components that are used by programs for providing the required functionality. Programs cannot use the hardware components directly. For this, an intermediary software layer providing interfaces that enable programs to access the hardware components is required. Such a software layer residing between programs and hardware resources is called operating system (OS) [13]. Apart from providing access to the hardware components, OSs are responsible for managing the resources available on the sensor nodes. This includes the scheduling of programs for execution as well as allocating the available resources among them when they are executed. OSs are therefore a central element for ensuring the optimal use of sensor resources.

There are several OSs available for sensor nodes [78, 79, 80, 81, 82]. All of them consider the scarce resources available to sensor nodes in their design. Differences are the methods they use for achieving the optimal performance as well as the features they provide. In the following, firstly, an overview of the typical characteristics of OSs for sensor nodes is provided. Subsequently, the OS that was chosen for the implementations in this thesis is introduced.

2.2.1 General characteristics

Operating Systems for sensor nodes are required to suffice with the limited resource budget available for them. This mandates the design of a general-purpose OS that suits all application scenarios. In the following, some of the essential design decisions that impact the characteristics of an OS are discussed[1].

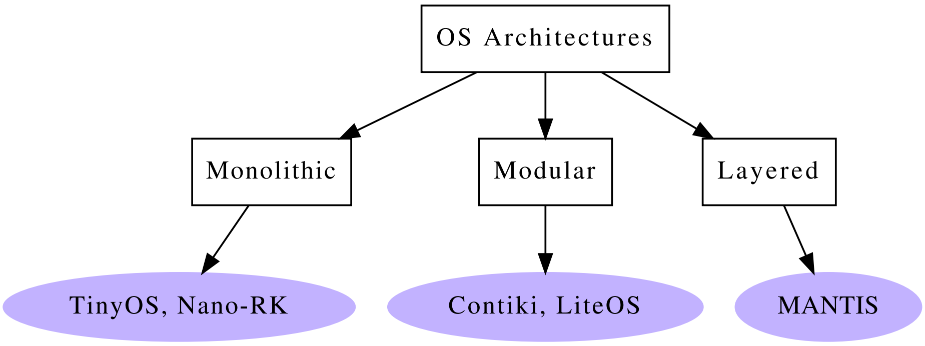


Figure 2.4: Typical architectures for WSN OSs - TinyOs [78] and Nano-RK [81] are the examples of the monolithic architectures. Contiki [79] and LiteOs [82] implement the modular approach. MANTIS [80] is the example of the less common layered architecture.

Architecture In traditional computers, programs and the OS are clearly separated. Programs run on top of the OS and communicate with it using interfaces. Independent of the programs to be run, the core OS remains the same. However, such a design is not suitable for sensor nodes, since an ideal OS for sensor nodes is required to be small in size due to the memory constraints (RAM, ROM). For this reason, OSs for sensor nodes are based on a specific architecture that enables a flexible configuration according

[1] The design aspects are categorized according to [13] and [83].

to programs that need to be run. Typical OS architectures for sensor nodes are the monolithic architecture, the modular architecture, and finally the layered architecture.

In a monolithic architecture, the OS is divided into services and each service is implemented in a separate module. Applications use only those modules implementing the services required for the desired functionality in the program code. The required operating system modules and the application code are then compiled into a single image. This is why this kind of OSs are called monolithic. Since only the required services need to be incorporated in the code, the monolithic OSs allow for reducing the code size of the applications significantly. For example, an application that is responsible for sensing is required to use only the OS module for sensing service. Other components e.g., providing the permanent data storage service on the external memory are excluded from the code image. TinyOS [78] and Nano-RK [81] are the examples of the monolithic OSs.

The size of the OS in a modular architecture is minimized by separating the core services and the services for programs. The core services implement the minimum functionality that is required for running programs on a sensor node such as the kernel, the program loader, the communication stack, and the device drivers. The core services are compiled into a single image and stored on the sensor nodes. The programs are loaded by the program loader and run on top of the core services. The tiny size of the core services makes modular OSs very suitable for sensor nodes. Contiki [79] and LiteOS [82] are the prominent OSs belonging to this category.

In a layered architecture, the OS services are divided into layers. Each layer runs the services from the layers above it. Layers can be implemented in separated threads or multiple layers can share a single thread. This allows for a fine-grained control over the trade-off between the performance and flexibility. MANTIS [80] is a layered OS designed for sensor nodes. Figure 2.4 provides the overview of the mentioned architectures.

Programming model Concurrency is an essential requirement for WSN applications. Event-based programming and multithreading are the common methods used for achieving concurrency in OSs for sensor nodes.

In event-based programming, concurrency is implemented through events and event handlers. Event handlers register themselves to the scheduler to be notified when certain events happen. The scheduler notifies the event handler when an event, that the handler is registered for, occurs and passes the control to it for execution. The event handler passes the control back to the scheduler upon the completion of the execution. Although multithreading is supported by TinyOS after version 2.1, TinyOS [78] primarily uses the event-based programming for concurrency. Concurrency in the kernel of Contiki [79] is also implemented using the event-based programming approach.

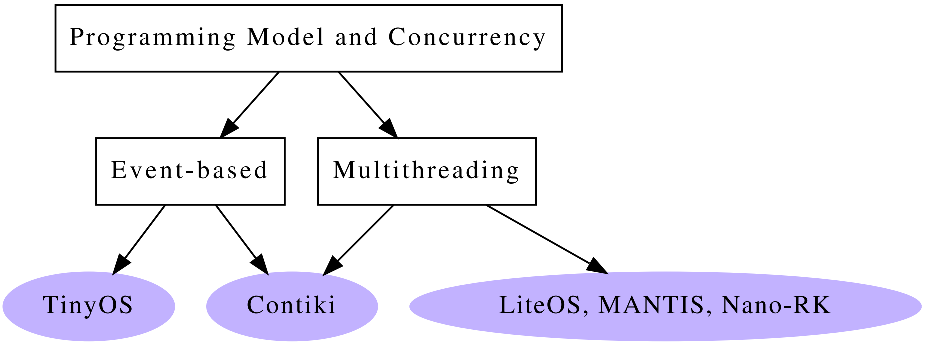


Figure 2.5: Typical programming models for WSN OSs - Contiki [79], MANTIS [80], Nano-RK [81], and LiteOS [82] follow the multithreading approach, whereas TinyOS [78] uses the event-based programming.

In multithreading approach, concurrency is achieved by allowing the execution of multiple tasks at the same time. The number of the tasks which can run concurrently is typically limited due to the resource constraints of sensor nodes. For this reason, the OS implements a pool where the allowed number of threads are stored. Each task that needs to be executed concurrently is assigned to a free thread available in the pool for execution. Upon the completion of the task, the thread is freed and stored back in the pool to be assigned for other tasks. Contiki [79], MANTIS [80], Nano-RK [81], and LiteOS [82] support the multithreading. Classification of the mentioned OSs according to the used programming model is depicted in Figure 2.4.

Scheduling Scheduling determines the order in which the tasks are executed by the microcontroller. Hence, scheduling is of a critical importance for the performance of the OS. Scheduling mechanisms for WSN OSs are mainly considered in two groups: queuing-based scheduling and round-robin scheduling [13]. Most of the OSs implement one of these mechanisms or use no scheduler at all. Contiki [79] is an example of an

OS without a specific scheduler. The scheduling is achieved through the hardware interrupts and the kernel events.

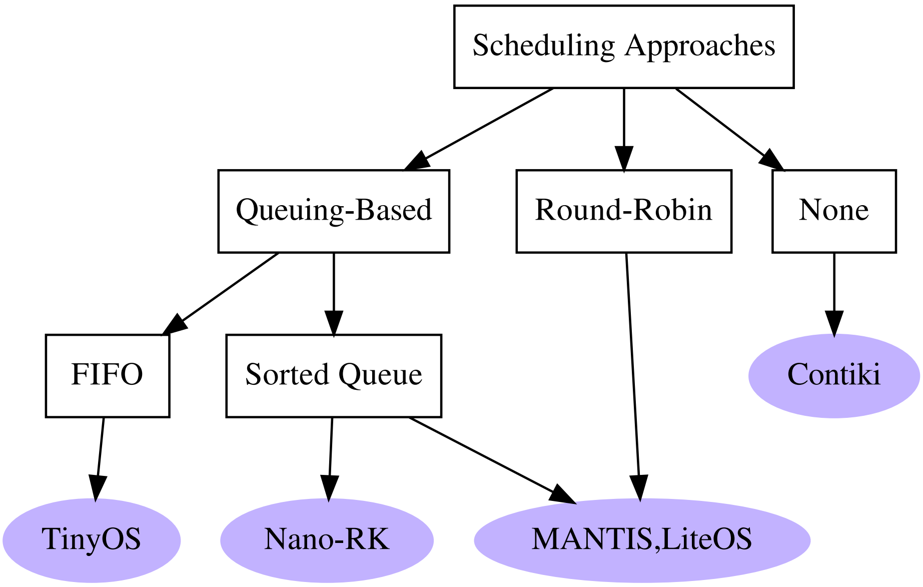


Figure 2.6: Typical scheduling approaches for WSN OSs - TinyOs [78] and Nano-RK [81] impelement a queuing based scheduling mechanism, while MANTIS [80] and LiteOS [82] additionally incorporate the round-robin scheduling mechanism. Contiki [79] uses no specific scheduler.

Depending on the policy used for selecting which task to execute, queuing-based schedulers can be first-in-first-out (FIFO) or sorted queue. In a FIFO scheduler, tasks are stored in a pool according to their arrival time. Upon the completion of a task, the oldest task in the pool is executed next. TinyOS [78] uses a FIFO based scheduler. In a sorted queue, tasks are executed according to a priority level assigned to them. Upon the completion of a task, the task with the highest priority is executed next as soon as the microcontroller is free. MANTIS [80], Nano-RK [81], and LiteOS [82] implement the sorted queue scheduling.

In a Round-robin based scheduling, tasks are executed only within the time slots assigned to them. Each task is stored in a pool as in the FIFO scheduler. Upon the completion of a task, the scheduler assigns a time slot for the next task in the queue. As soon as the microcontroller is free, the task is executed for so long as the time slot assigned to it. Its execution is paused when the time is over. In case the execution of the task is not fully completed, it is re-queued to the end of the task pool and starts to wait for the assignment of a new time slot to resume the execution. This operation is repeated for all tasks until the pool is empty. MANTIS [80] and LiteOS [82] use the round-robin scheduling within sorted queues. Figure 2.6 summarizes the scheduling approaches used in the OS of sensor nodes.

Programming language and data types Applications for OSs designed for sensor nodes are programmed in C or in its derivatives. TinyOS [78] programs are written in nesC (network embedded systems C) [84] which is a dialect of the C language. Contiki [79], MANTIS [80], and Nano-RK [81] are programmed in native C language. Programs for LiteOS [82] are written in LiteC++. Therefore, the native data types of C language including some complex data types such as struct and enum are supported by all of the OSs.

2.2.2 Implementation software

Security mechanisms provided in this thesis are independent of the OS. They can be implemented in any OS designed for sensor nodes ([78, 79, 80, 81, 82, 85]). However, TinyOS [78] is the mostly used one due to its excellent documentation and a large number of components available for programmers. This is the reason for choosing TinyOS as the implementation platform in this thesis.

TinyOS [83] is a component-based monolithic OS. The core OS is only 400B big. Hence, it can be used on sensor nodes even with a very limited memory. TinyOS is an application-specific OS. That is, the OS is composed of interfaces and components which implement the services required by applications such as sensing, communication, storage, and timers. Programs are built by connecting the components that provide services required by the application with each other through interfaces. Thus, TinyOS conserves the memory of sensor nodes by excluding the unnecessary OS services from the application code. TinyOS applications are written in nesC [84]. It is a derivative of the C language which supports the concepts and execution model of TinyOS. TinyOS has the following attractive features [83]:

- Provides database support for permanent data storage with TinyDB [86]
- Provides a framework for communication security with TinySec [87]

- Provides simulation support with TOSSIM [88]
- TinyOS can be emulated with Avrora [89]
- Supports almost all of the off-the-shelf sensor node platforms
- Programs are written in nesC (network embedded systems C), a dialect of the C language

2.3 Impact of resource constraints to design

As shown in Section 2.1 above, sensor nodes are equipped with very limited hardware resources. Data memory available to a typical sensor platform is mostly no more than a few tens of kilobytes. Similarly, program memory is usually limited to a few hundred kilobytes. The microcontroller is clocked at less than 10MHz for keeping the energy consumption at minimum. These resource constraints need to be considered during the design and implementation of WSN applications and addressed by applying appropriate methods. There are two main challenges in dealing with this problem. The first one is the identification of the appropriate means to be used for minimizing the resource demands of the applications. The second challenge is the identification of the most critical resource to protect for a specific application scenario.

In the following, first of all, general methods for dealing with resource limitations of sensor nodes are identified. Subsequently, the design decisions made in this thesis, explicitly for dealing with the resource constraints in the secure remote programming scenario, are introduced.

2.3.1 General approaches for dealing with resource constraints

There are four types of hardware resources on a sensor node: power supply, microcontroller, memory, and radio transceiver. To be able to propose an appropriate method for optimizing the consumption of a resource, fist of all the consumers need to be identified.

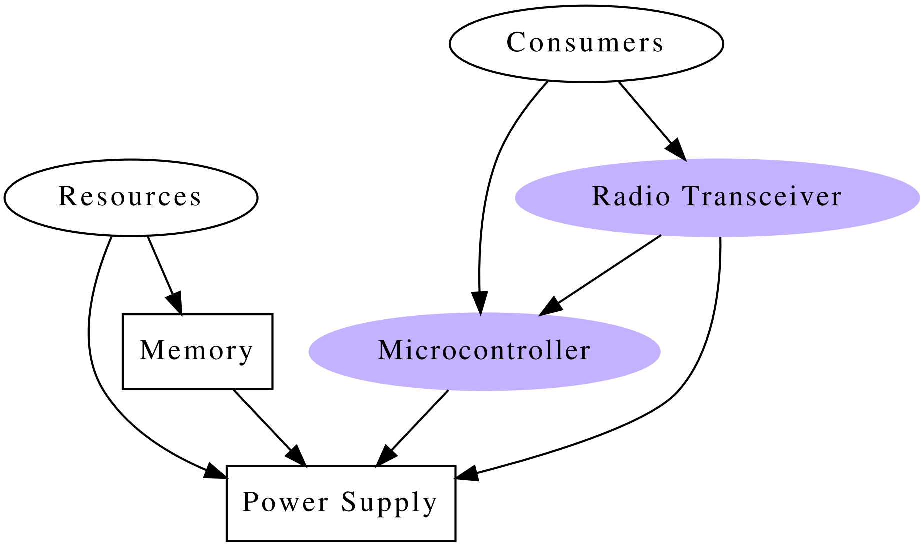


Figure 2.7: Resources and consumers on a sensor node - Power supply and memory are the resources while microcontroller and radio transceiver are the consumers.

Resource-consumer model The microcontroller is a hardware module providing processing service which is virtually endless. However, it is available only if the microcontroller is supplied with power. Hence, the microcontroller is a consumer rather than a resource, since it depends on the power source to operate. Similarly, the radio transceiver is a consumer. It depends on the power supply and the microcontroller to perform its task. The power supply and the memory unit (ROM and RAM) are the resources. The power supply can provide its service as long as its energy resource is not depleted. In a similar sense, the memory unit (ROM and RAM) can provide its service as long as the resource (storage) capacity is not exceeded. The resource-consumer model considered in this thesis is depicted in Figure 2.7.

Identifying means for optimizing resource consumptions According to the resource-consumer model above, the critical resources to be conserved on a sensor node are the power and memory. Their consumers are the microcontroller and the radio transceiver, as their services depend on the availability of the resources.

The power consumption of the microcontroller depends on the time in which it is active. Its activity time is based on the execution performance which is determined by two factors. The first one is the hardware characteristics of the microcontroller as well as the OS. These are out of the scope, since the application designer has typically no direct impact on them. The second factor is the quality of the design an the implementation. It depends on the choice of algorithms as well as the programming style used in their implementation.

Another major power consumer is the radio transceiver. Likewise the microcontroller, the amount of energy consumed by the radio depends on its activity. That is determined by the amount of data being communicated. The larger the communication overhead, the higher is the energy consumption. For this reason, methods targeting at optimizing the resource consumptions caused by the radio transceiver are required to focus on the reduction of communication overhead. This requires the choice of proper algorithms during the design as well as a decent programming style leading to protocols and implementations with a small communication overhead.

The methods for optimizing the memory consumption need to be considered in two groups: methods for optimizing the power consumption caused by the memory unit and methods for reducing the use of memory itself. The power consumption of the memory depends on its performance. That is, the less the number of memory operations (shifts, writes, deletes, etc.) are performed, the faster is the memory and hence the less is the power consumption. The performance of the memory is determined by the construction choices and the quality of implementation. The use of memory can be optimized using an appropriate programming style and appropriate algorithms that lead to implementations with small code size and RAM requirement. Often, there is a trade-off between the code performance and the code size. Obviously, the size of the programs is determined mainly by the algorithms to be implemented.

The observations from the discussions above are summarized in Figure 2.8. It implies that there are two essential means for optimizing the resource consumption on sensor nodes: the choice of appropriate programming style and the choice of appropriate algorithms. These are discussed in the next subsections in more detail.

2.3.1.1 Choice of programming style

The code performance as well as the consumption of data (RAM) and program memory (flash) can be optimized by applying appropriate programming techniques. In the following, general guidelines for efficient programming are introduced [1, 90, 91].

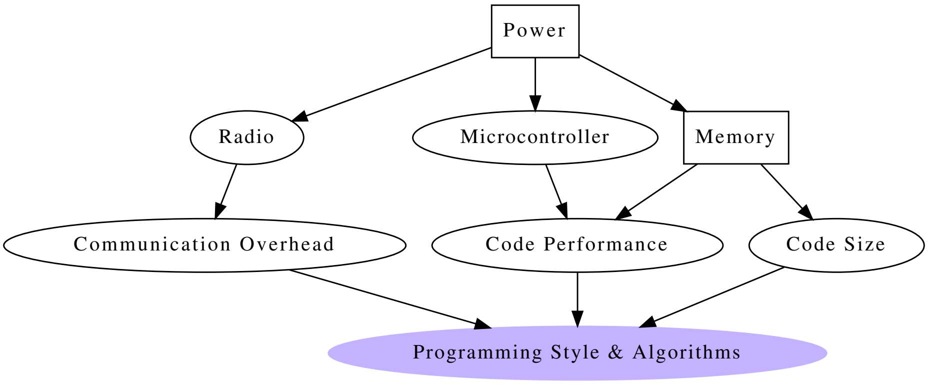


Figure 2.8: General means to optimize resource consumptions - The choice of algorithms and programming style impacts the code size, code performance and data amount which in turn influence the amount of the power and memory consumptions.

Reducing RAM requirement RAM is the temporary data storage and working space for programs. There are two main consumers of RAM: variables and stack structures. A variable is a memory location which stores data required by programs during their execution. Data can be the output of a program or the intermediate results of some functions which are in turn the inputs of other functions. Variables declared inside functions are called local variables, while variables declared outside functions are called global variables. A stack structure is a memory space that keeps the track of the code being executed. It stores the return addresses to functions needed when executions of code at different memory spaces are completed. RAM consumption can be significantly reduced by optimizing the use of variables and stack structures by following the implementation rules below:

- Store global constant data in program memory: RAM consumption can be reduced by storing the variables for global constant data in the program memory instead of RAM. This can be achieved typically by using the FLASH keyword in the declaration of a variable. However, the penalty for this is the performance loss and the increased code size. The reason of the performance loss is that read/write operations from/to flash memory are slower than they are in RAM. The reason of the large code size is that variables are coded directly into the flash memory with the program code.

- Use of registers: Registers are similar to variables. They can be used for storing data during program executions. The difference is that registers are internal to microcontrollers and do not reside in RAM. The microcontroller can access registers faster than variables. Hence, storing the frequently accessed variables in registers increases not only the execution performance, but also decreases the RAM consumption. Compilers can be ordered to store a variable into a register by adding the REGISTER keyword in the variable declaration.

- Declare variables as local instead of global whenever possible: Global variables are stored in RAM. They are loaded into registers to be processed and stored back into the RAM after processing. Hence, the use of global variables increases the number of memory accesses. This not only decreases the code performance, but also increases the RAM consumption and the code size. Therefore, use of global variables should be avoided in the code whenever possible. Local variables are assigned to registers whenever there are free registers available to the microcontroller. For this reason, use of local variables increases the execution performance while reducing the RAM consumption and the code size.

- Avoid recursive programs: Recursive programs increase the size of the software stack stored in RAM. Hence, use of recursive algorithms should be avoided for minimizing the RAM consumption.

Reducing code size Following techniques can be used for reducing the code size of applications:

- Declare variables as local instead of global whenever possible: As mentioned above, global variables are stored in RAM and loaded into registers whenever they need to be processed. This not only decreases the code performance, but also increases the RAM consumption and the code size. Therefore, global variables should be replaced with local variables whenever it is possible to minimize the code size.

- Pack global data into structures: Accesses to consecutive memory addresses are efficient, since a constant pointer needs to be incremented and decremented only to point to the required memory location. Hence, data global to a program should be collected in structures instead of global variables. This makes it possible to access data with more efficient instructions that result in smaller code size.

- Use correct data types: Variables should be assigned to the smallest possible data type supported by the microcontroller. For example, if the microcontroller in use is 16-bit, use of 32- and 64-bit variables should be avoided. This may not only reduce the code size significantly, but also increase the code performance, as fewer instructions are needed to compute on smaller data types.
- Use macros instead of functions: In general, using macros results in a code with smaller size and better performance.
- Custom functions and code reuse: Standard library functions are written for general purpose. They usually contain unnecessary code for checking possibly multiple execution cases. Hence, writing custom functions fitting the application may reduce the code size and increase the performance of the application code. Moreover, collecting several functions into a single module increases the code reuse. This decreases the code size.
- Use recursive programs: Recursive programs usually result in a more compact code. Hence, use of recursive algorithms decreases code size while reducing the code performance and increasing the RAM consumption.
- Using post-increment and pre-decrement in loop counters result in smaller code. Moreover, do while() loops are superior to while() loops in terms of code size.

Improving code performance The code performance can be improved with the following approaches:

- Inline functions: Inline functions improve the code performance significantly. A function is decleared as inline by using the keyword INLINE. This makes the compiler copy the implementation code into the desired place whenever the function needs to be executed. This removes the overhead resulting from function calls and results in a better execution performance. Inline functions are particularly suitable for frequently used functions. The main penalty is the increased code size.

- Loop unrolling: Loop unrolling is the most effective way for improving the code performance. For instance, the for-loop

  ```
  for(int i=0;i<5;i++)
  {dummy[i]=0;}
  ```

  is written as

  ```
  dummy[0]=0;
  dummy[1]=0;
  dummy[2]=0;
  dummy[3]=0;
  dummy[4]=0;
  ```

  to reduce the overheads resulting from maintaining the index variable and conditional branch instructions. The main disadvantage of this approach is the increased code size.

- Use of registers: Storing data in registers results in fewer loads from the data or flash memory. Since the microcontroller can directly access the registers and access to registers is faster than to the data or program memory, the code performance is improved.

- Use of static variables: Use of static variables instead of global variables increases the code performance. Thus, variables accessed inside functions should be declared as static local variables to improve the code performance.

2.3.1.2 Choice of algorithms

Guidelines on efficient programming described in the previous subsection are generic and can be applied in all application scenarios in accordance with the resource to be optimized. Choice of algorithms is largely application-centric and depends on the application scenario to be implemented. As the focus of this thesis are security mechanisms for remote programming, general guidelines for selecting cryptographic algorithms are provided.

Taxonomy of cryptographic algorithms Basic cryptographic goals which are common to most applications are confidentiality, data integrity, authentication, and finally non-repudiation. There are several cryptographic algorithms that can be used to achieve these security goals. Depending on the type of the keys and the requirements on them, cryptographic algorithms are classified into two groups: symmetric algorithms and asymmetric algorithms. Figure 2.9 provides an overview of the cryptographic methods and shows how they relate to general security goals.

Symmetric algorithms require to share the same key at all entities that wish to interact with each other. The key needs to be kept secret. In asymmetric algorithms, each entity has its own public-private key pair. The public key can be known to everybody. The private key is known only to the entity owning the key pair.

Generally speaking, symmetric algorithms are computationally cheaper than their asymmetric alternatives. Moreover, symmetric block ciphers can be used to construct pseudorandom functions and permutations which are the basis of other symmetric mechanisms for ciphers, signatures, hash functions, and MACs. This is a great property that makes it possible to reduce the code size of implementations significantly by allowing code reuse. On the other hand, key management for asymmetric algorithms is usually much simpler than that is for symmetric algorithms. Hence, many real-world applications combine both instances to inherit their advantages. A typical combination in practice is to use an asymmetric encryption for securing the transport of a secret key to be used in a symmetric mechanism. Such a cryptographic algorithm, combining asymmetric cryptography with symmetric cryptography, is called hybrid algorithm.

In the following, symmetric and asymmetric algorithms are analyzed according to their computation, communication and memory overheads.

Computation overhead

- Asymmetric cryptography: Asymmetric cryptography relies on one-way or trapdoor one-way functions [7]. A one-way function is a function which is easy to compute, but hard to revert. A trapdoor one-way function is a one-way function which is easy to revert when some extra trapdoor information is given, but hard otherwise. Roughly speaking, an asymmetric algorithm is a trapdoor one-way function parameterized with a public-private key pair. The private key is the trapdoor which enables to compute the one-way function in both directions

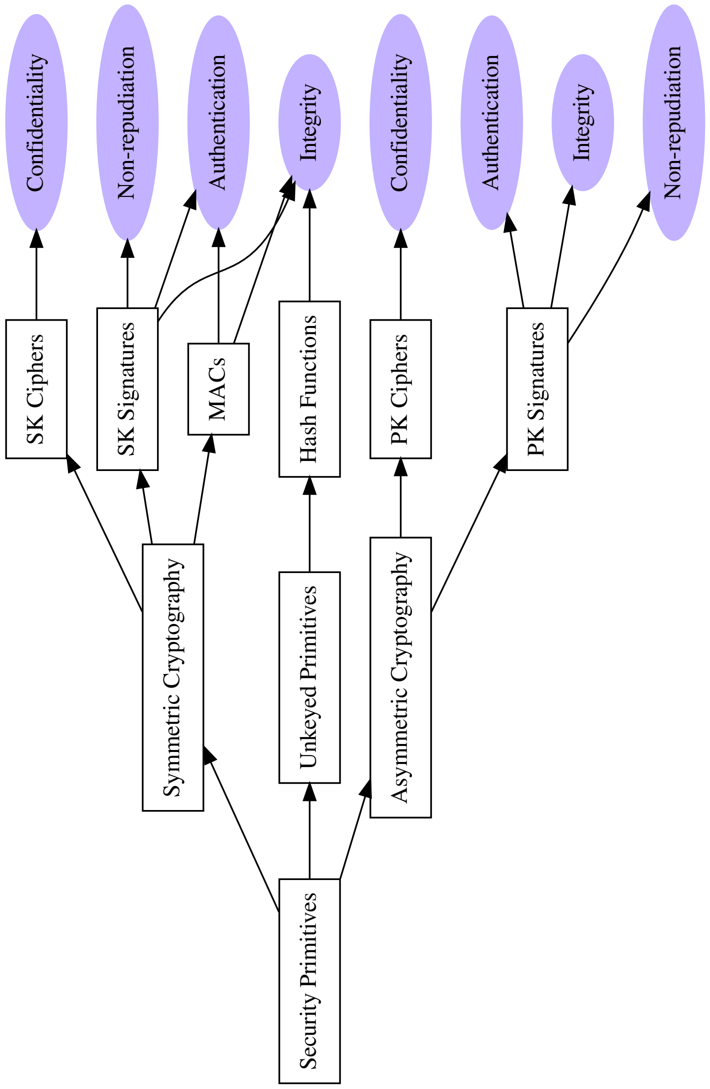


Figure 2.9: Taxonomy of security primitives with respect to basic cryptographic goals [7] - Signatures provide authentication, data integrity and non-repudiation. MACs are used to achieve authentication and data integrity. Confidentiality is provided by ciphers. Finally, hash functions can be used for protecting data integrity.

easily. The public key enables to compute the one-way function only in forward direction, but not in the opposite direction. Forward direction is typically used for encryption and signature verification, while the inverse direction is used for decryption and signature generation.

Examples of trapdoor one-way functions are the RSA problem, Rabin problem, and Diffie-Hellman problem. The security of many asymmetric mechanisms depends on the assumption that these problems are difficult to solve. For example, the security of the RSA algorithm relies on the integer factorization problem. That is, given two prime numbers, it is easy to compute the product (forward direction). However, given a positive integer, it is difficult to factor the product into primes (inverse direction). An attacker needs to factor the modulus of RSA to break its security. In general, factoring large numbers is assumed to be more difficult than factoring small numbers. Thus, the modulus chosen for RSA needs to be large enough to be secure against integer factorization. At least a 1248-bit modulus (which is also the key size) is required for obtaining a medium-term security [10] (see Table 3.4). As in RSA, it is needed to choose large parameters also for other trapdoor functions which are the basis of some other asymmetric algorithms.

As asymmetric algorithms require computations with very large numbers to provide a sufficient level of security, their computational overhead is typically very large. Thus, use of asymmetric cryptography in the design of WSN applications should be avoided, whenever alternatives relying on symmetric cryptography are available.

- Symmetric cryptography: Symmetric algorithms typically rely on pseudorandom functions and pseudorandom permutations [7]. Roughly speaking, a pseudorandom function is a keyed function whose outputs are indistinguishable from the outputs of a random function of the same length. A pseudorandom permutation is a keyed permutation which is efficiently invertible. All efficient symmetric algorithms for encryption and many algorithms for message authentication, (symmetric key) signature, and hash computation are implemented using a pseudorandom permutation. As a pseudorandom permutation is a keyed function, the key needs to be known for performing encryption and MAC generation as well as decryption and MAC verification. In contrast to an asymmetric setting, the key needs to be kept secret and shared only among allowed entities. The key used in hash constructions needs to be public.

In practice, pseudorandom permutations are given by block ciphers such as AES, DES, and RC5. The security and efficiency of block ciphers are proven in practice since many years. For example, there is still no known attack which breaks the security of 128-bit AES faster than the exhaustive search (i.e., the brute force attack) [92]. This allows to use much smaller keys in symmetric cryptography than in asymmetric cryptography for an equivalent level of security. For example, the security of a 1248-bit asymmetric key is considered to be equal to the security of an 80-bit symmetric key [10].

As symmetric algorithms are based on cheap computations such as XOR operations, substitutions, and group arithmetics with small numbers, the computational overhead is typically much smaller than that is for asymmetric cryptography. Thus, use of symmetric cryptography should be preferred in the design of WSN applications.

Memory overhead

- Code size:
  - Asymmetric algorithms: As depicted in Figure 2.10, the construction of asymmetric primitives relies on a (trapdoor) one-way function. Discrete logarithm problem, integer factorization problem, Diffie-Hellman problem and subset sum problem are the basis for the practical realization of trapdoor one-way functions. The discrete logarithm problem is also the basis of the asymmetric cryptography based on elliptic curves. The underlying computational difficulty for each problem is different. Thus, the possibility of code reuse in the implementation of asymmetric mechanisms is usually limited. For example, except the underlying finite field arithmetic, there is no central build block that can be used for implementing a cipher based on integer factorization (RSA) and a signature based on discrete logarithm problem (ElGamal). Similarly, if a hash function or MAC mechanism is

required, they do not profit from the available code and need to be implemented separately. This limits the flexibility for optimizing the resource consumptions during the design and implementation of applications. Moreover, complex algorithm structures used in symmetric cryptography result in large memory footprints and low performance compared to the symmetric cryptography based mechanisms. For this reason, use of asymmetric cryptography should be avoided in the application design for WSNs.

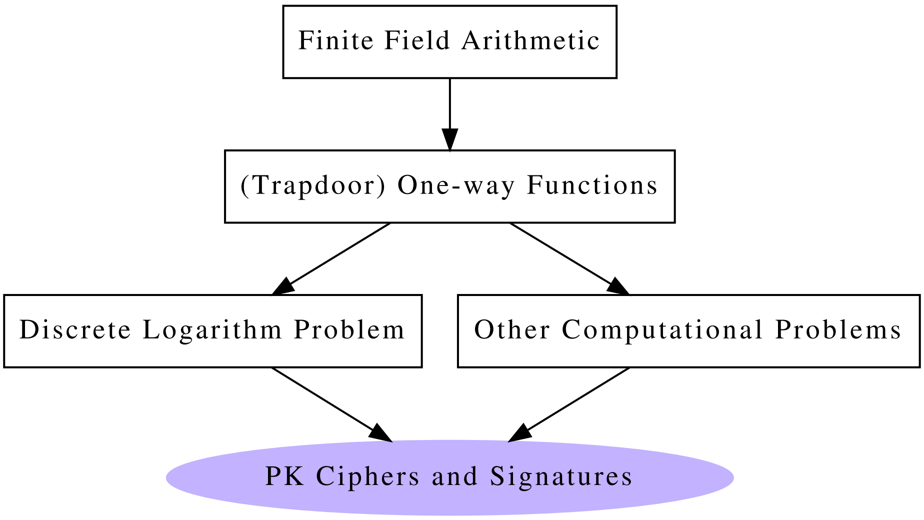

Finite Field Arithmetic
(Trapdoor) One-way Functions
Discrete Logarithm Problem
Other Computational Problems
PK Ciphers and Signatures

Figure 2.10: Construction of asymmetric algorithms from one-way functions - All asymmetric algorithms rely on a one-way function. The security of one-way functions is determined by the computational difficulty of the underlying mathematical problem. Hence, asymmetric algorithms need to operate with very large numbers to provide a sufficient level of security. This leads to implementations with large overhead in terms of code size and data memory.

– Symmetric algorithms: Almost all symmetric primitives can be implemented using a single building block, i.e. a block cipher. Block ciphers are the practical implementations of pseudorandom permutations which are the basis of all symmetric algorithms. As shown in Figure 2.11, secret-key encryption mechanisms can be implemented by combining a block cipher with a mode of operation such as the OFB mode. Similarly, message authentication mechanisms can be constructed by combining the CBC mode with a block cipher. There are also several algorithms that allow for constructing a hash function from a block cipher. Matyas-Meyer-Oseas, Davies-Meyer, and Miyaguchi-Preneel hash functions are the example of such constructions based on block ciphers. MAC constructions can alternatively be based on hash functions. NMAC and HMAC are the prominent examples that implement a MAC mechanism using a hash function. Finally, signatures based on symmetric cryptography can be constructed using a hash function. Such signature mechanisms are typically for one-time use. Thus, multiple-time signatures can be constructed by combining a one-time signature with authentication trees. Such a construction is presented in Section 4.4.

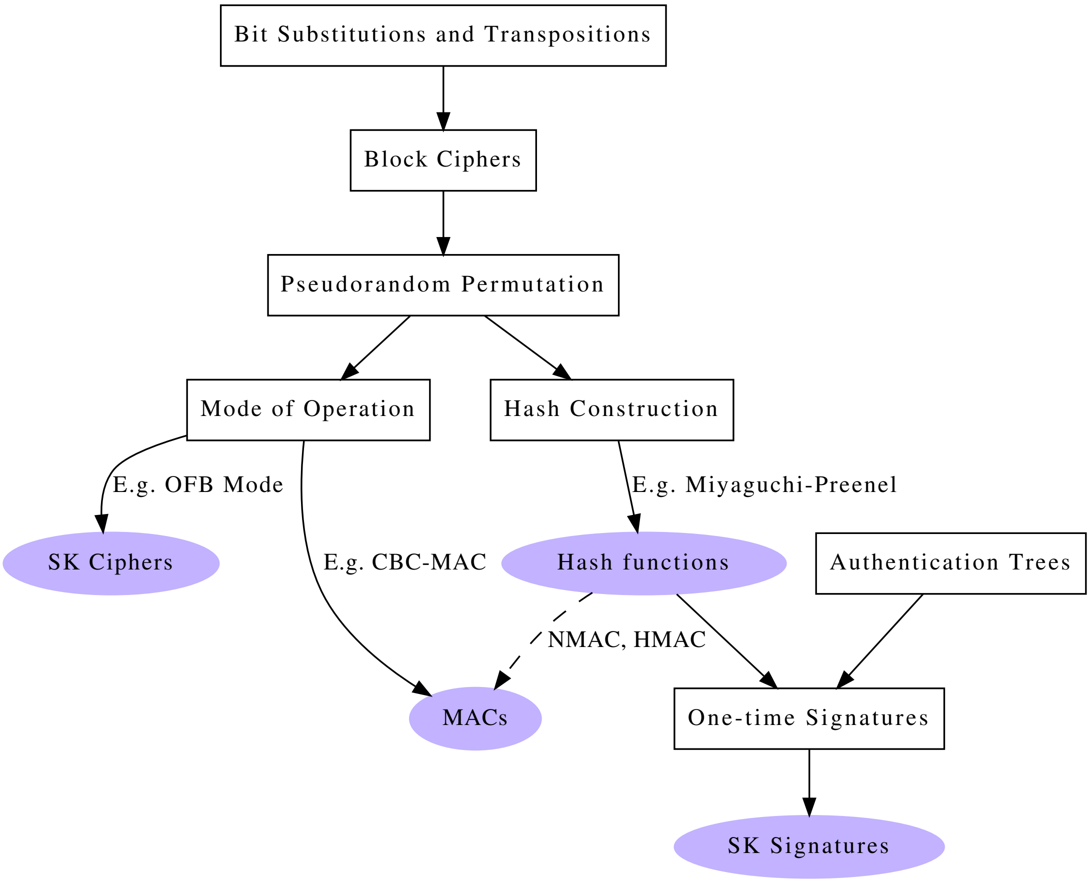


Figure 2.11: Construction of symmetric algorithms from a block cipher - Block cipher is the central building block for implementing mechanisms for symmetric cryptography. Thus, implementation of symmetric cryptography is very efficient in terms of code size and data memory.

In summary, symmetric cryptography allows to construct security primitives required for achieving general cryptographic goals such as confidentiality, data integrity, authentication, and non-repudiation from a single building block (block cipher). For this reason, use of symmetric cryptography leads to implementations with much smaller code size than implementations based on asymmetric cryptography.

- Data memory (RAM) requirement: Size of keys, ciphers, and signatures for asymmetric primitives are much larger than for symmetric primitives. For example, an 80-bit symmetric security equals to a 1248-bit asymmetric security [10] (see Table 3.4). Typically, the word size of microcontrollers used on sensor nodes are limited to 8- or 16-bit. Due to the limited register sizes, the computations with large numbers require a very large number of loads and stores between the registers and the data memory. Moreover, the intermediate results produced from the computations need to be stored in the data memory, since they do not fit into the registers. Hence, the larger is the size of data being computed, the higher is the penalty in terms of the data memory and the performance. Symmetric primitives therefore occupy less data memory than asymmetric primitives due to the smaller key and cipher sizes.

Communication overhead Considered the communication overhead, symmetric cryptography is more efficient than asymmetric cryptography. There are mainly two reasons for this. Firstly, due to the smaller key sizes, the cryptographic material that needs to be delivered for symmetric algorithms is smaller. Secondly, asymmetric algorithms are less flexible in terms of the size of the outputs. For example, the encryption of a single bit with RSA would produce a ciphertext of the size of the modulus. Given a modulus of 1024-bit, the ciphertext for one bit message encryption would be 1024-bit long. In contrast, block ciphers are more flexible. The size of ciphertexts is limited with the block size of the underlying encryption mechanism. That is, the ciphertext resulting from the encryption of one bit data, for example with RC5, would be 32-bit long.

For this reason, use of asymmetric cryptography should be avoided whenever possible. However, if it is mandatory, ECC-based mechanisms are usually a better choice. Even though their code size is larger, the advantage of ECC-based mechanisms is the smaller key, cipher, and signature sizes. For example, a 256-bit ECC key provides the same level security as a 3248-bit RSA key [10] (see Table 3.4). ECC-based mechanisms are therefore a better choice when the communication overhead needs to be reduced.

2.3.2 Efficient implementation

As it can be seen from Figure 2.10, the performance of asymmetric primitives relies on the performance of the underlying finite field arithmetic. In the following, a short introduction to the efficiency of the finite field and EC level arithmetics is provided. The reader is referred to [2, 93] for a more detailed analysis.

Finite field level Arithmetic operations for asymmetric as well as symmetric primitives are built on top of the finite field level operations. A finite field level implements basic operations such as addition and multiplication of large numbers and can be realized by using a prime field or a binary field. The main drawback of the binary field is the low performance due to high number of memory accesses required in computations and poor support of binary field arithmetic by typical microprocessors. The high cost of the inversion operation is the main drawback of the prime field. However, as the number of inversions required in e.g., EC operations can be reduced to only one by using mixed coordinate systems, the prime field allows for more efficient implementation on sensor platforms.

The multiplication is the most critical operation at the finite field level for the overall performance, as it is the dominating operation in most of the crypto mechanisms. Despite of its slightly larger memory footprint, the *Hybrid* multiplication method [94] is the most promising algorithm for the sensor platforms where the number of registers available is small and memory access is inefficient. It combines advantages of well-known *Schoolbook* and *Comba* methods. A modular reduction is executed to keep the result in the range of the chosen finite field. The *Pseudo-Mersenne prime reduction* is more efficient than the well-known *Montgomery* and *Barrett reduction.* However, it is only applicable to primes with a special form.

Elliptic curve level This level provides the point addition and doubling as well as the scalar multiplication operations. An EC point can be represented in several coordinate systems each of which differs in terms of performance and memory requirement.

The use of *mixed* coordinate systems, i.e., input and output points of an operation are represented in different coordinates, is a way of improving the efficiency at this level [95]. The basic idea is to combine the efficiency of different coordinate systems. The implementation of point additions with $\mathcal{AJJ}$[1] mixed coordinates is promising for sensor platforms due to the lower number of additions and multiplications required at the finite field level as well as the lower memory requirement. For the same reasons, using the *Jacobian* coordinates for representing the input and output operands of a point doubling operation allows for efficient implementations on sensor platforms.

The scalar multiplication is realized by using a chain of point additions and doublings with a so called *Double and Add* algorithm. It can be performed by evaluating the binary representation of the scalar from *Left-to-Right* or *Right-to-Left*. The performance of the multiplication for both directions is the same. However, the memory requirement of the Right-to-Left method is higher, as the intermediate point doubling results need to be stored for the next iteration. Hence, the Left-to-Right binary method is a proper choice when the platform's memory is limited.

For further optimization, the *Interleave method* may be used to reduce the number of point doublings at the cost of increased memory usage for storing some pre-computed points. Reducing the number of point additions is another way of improving the performance of scalar multiplication. The basic idea is to replace point additions with computationally cheaper point doublings. This can be achieved by converting the scalar into a signed representation with *non-adjacent form* (NAF) or *mutual opposite form* (MOF). Both representations require the same number of point additions. The main difference is the direction of the recording. MOF is recorded from Left-to-Right, while Right-to-Left recording is performed for NAF. Hence, the whole scalar must be converted in its signed NAF representation before a scalar multiplication with the Left-to-Right binary method is performed. This leads to higher memory consumption and makes MOF more suitable for the implementations on sensor platforms.

2.3.3 Design choices in this thesis

In the previous sections, several methods for optimizing the resource consumptions and several algorithms according to their suitability for sensor nodes have been discussed.

[1] Input operands are represented in *Jacobian* coordinate system, while the result is represented in *Affine* coordinate system.

However, the choice of concrete techniques to be applied depends on the application to be implemented. In the following, design choices for secure remote programming are discussed.

Identification of the most critical resource There is always a trade-off between the performance, code size, memory consumption, and communication overhead. Depending on the characteristics of the application being considered, one particular resource can be more important than the others. Thus, before choosing the appropriate design principles, the most important resource to be optimized needs to be identified.

Remote programming protocols in real-world deployments are mainly used for removing bugs as well as software updates to fit the functionality of a WSN to current demands. Typically, these operations are performed rather seldom. Therefore, the code performance is not very critical for a remote programming protocol. The remote programming protocol must co-exist with the actual application code. If the code size of the security mechanisms is too large, the functionality of the application needs to be reduced. Obviously, this is not desired and must be avoided. For this reason, the code size is chosen as the most critical resource in this thesis. In practice, any application running on a sensor node is paused during a remote programming. Thus, the entire RAM becomes available to the remote programming. This indicates that RAM consumption is less critical than the code size. However, security mechanisms must still fit into the available RAM together with the code of the remote programming protocol for a robust operation. For this reason, the RAM consumption is the second most important resource to protect after the program memory in this thesis. The most power consuming operation on a sensor node is the radio communication. As the lifetime of sensor nodes is desired to be as long as possible, the power consumption should be minimized. On the other hand, the power consumption caused by a remote programming is not as critical as the RAM consumption, since it is a seldom operation. Therefore, the communication overhead is the third most important criteria when design choices are made. Finally, the code performance is the least important factor in this thesis. The reason is that the power consumption spent during the execution of programs locally on a sensor node is much lower than the power spent for the communication.

In summary, the importance of the resources in this thesis is in the following order:

1. code size

2. RAM consumption

3. communication overhead

4. code performance

Design choices To keep the code size small, all security mechanisms can be implemented using symmetric cryptography. In particular, a block cipher can be used as the central building block for implementing the required security primitives including hash functions, Message Authentication Codes (MACs), and digital signatures. Chapter 4 describes how to realize such a code size optimized cryptographic toolbox using a block cipher. Moreover, the programming tips introduced at Section 2.3.1.1 are used to minimize the code size.

3

Security Models and Requirements Analysis

Protecting a remote programming application against adversarial interferences requires applying appropriate security mechanisms. The answer to the question "what are the appropriate security mechanisms to be applied?" depends on the dangers that potentially compromise the security. Hence, designing security solutions for a remote programming mechanism should start with identifying the security threats first. Once the security threats and attacks are identified, the security design is proceeded by determining the proper countermeasures against them.

Threat and attacker modeling are the well-known approaches used for determining the security threats and attacks which must be considered in the design of security solutions. In general, threat modeling is used to identify the dangers that might compromise the security. Attacker modeling is used to determine which of the identified threats are realistic in practice and thus must be mitigated.

Both models closely relate to each other. Thus, their uses are often mixed up in the literature. Most works rely on either a threat model or an attacker model but not on both while designing security solutions. In this work, both models are used as complementary to each other. That is, threat modeling is used to identify the potential threats to a remote programming mechanism. Attacker model is then used to determine those threats that must be mitigated by the security solutions as well as the level of security they must provide.

This chapter is organized as follows. Firstly, threat and attacker models are compared with each other to describe their roles in a security design. Next, general approaches for threat and attacker modeling are introduced shortly. Subsequently, by following the presented approaches, a threat as well as an attacker model for the remote programming scenario are developed. Finally, based on the models developed, security requirements for a remote programming mechanism are identified.

It is to note that the goal of this chapter is to specify the security requirements for remote programming in WSNs. The proposed solutions satisfying the specified requirements are described in Chapters 4, 6, 7, and 8.

3.1 Threat model vs Attacker model

This subsection shortly describes the roles of threat and attacker models in the design of security. The output of a threat modeling is a set of threats that might compromise the security of a system. From the security designer's point of view, a threat model provides the set of vulnerabilities that might be exploited by adversaries to implement their attacks.

Attacker models specify the abilities of adversaries and the resources available to them while performing attacks. Hence, they are based on assumptions. From the security designer's point of view, an attacker model helps with identifying the threats (among the ones identified in the threat model) which are realistic according to the capabilities of adversaries. Security solutions need to consider and mitigate only those realistic threats. Moreover, the level of security that the security solutions must provide is determined using the attacker model. This relation between both models is depicted in Figure 3.1.

The set of realistic threats may be equal to the set of threats identified in the threat model or be only a subset of them. If attacker model derives threats that are not in the threat model, this is an indication that the threat model is incomplete or that both models do not fit together. Hence, using both models provides also a check of the validity of the threat model. The attacker model considers the characteristics of adversaries while identifying the potential attacks. A typical approach for characterizing the adversaries is their classification in terms of resources. For instance, an amateur hacker is obviously much more limited in terms of resources and attacks that he/she can execute

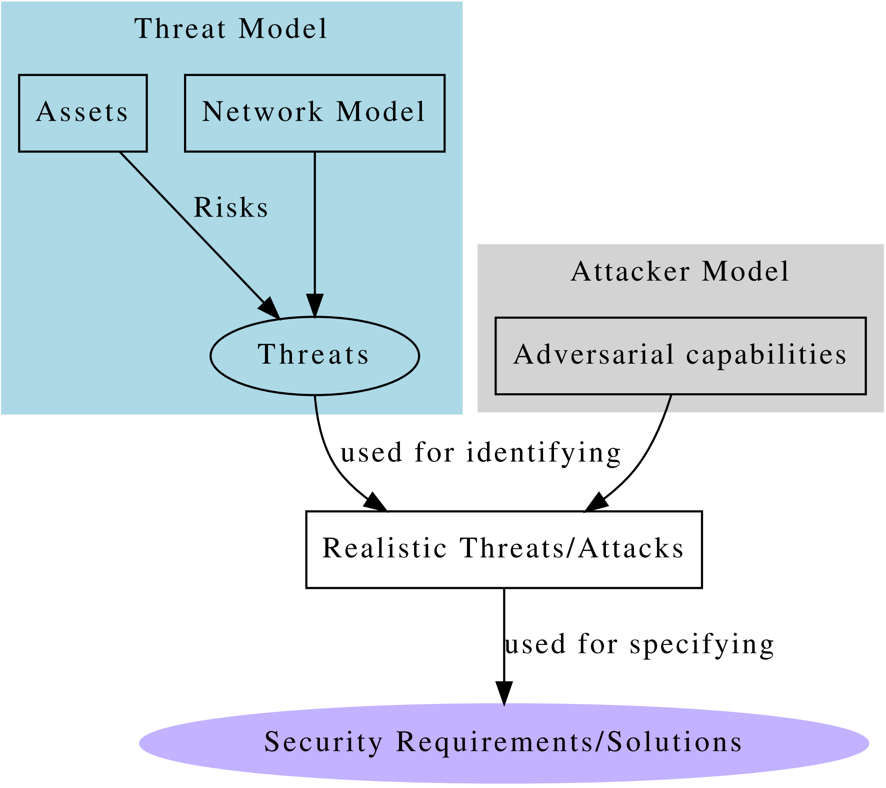


Figure 3.1: Threat model vs Attacker model - The threat model is used for determining all possible threats that may compromise the security. Threats depend on the network model and assets to be secured, which are, in turn, based on the application being considered. The attacker model is used to determine the threats that need to be considered in practice. It depends on the characteristics of adversaries being considered, which are, in turn, typically specified by the attacker model.

than a state-financed adversary. Hence, security designs in practice typically consider only those attacks that are realistic according to the attacker model, rather than all the threats determined in the threat model (see Figure 3.2). Due to the severe resource constraints, this is the case in most WSN applications.

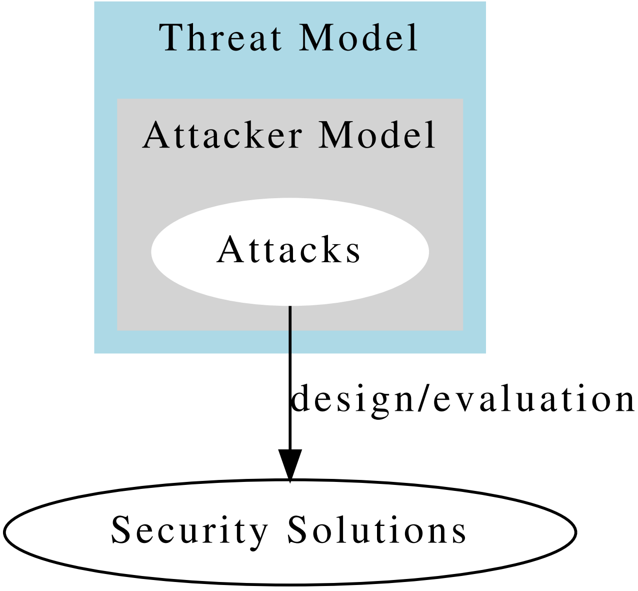


Figure 3.2: Impact of attacker model to security design - Depending on the characteristics of adversaries, attacks considered in a attacker model may be a subset of the potential threats identified in a threat model. Due to the resource constraints, the security design needs to consider only those attacks possible according to the attacker model rather than all threats identified in the threat model.

3.2 General approaches

This section introduces some of the existing approaches used for threat and attacker modeling.

3.2.1 Threat modeling

Several approaches have been proposed for threat modeling [8, 96]. They are all basically composed of the following steps [8]:

- Identification of the key assets

- Network modeling
- Decomposition of the application
- Identification of the threats
- Rating the threats according to risks they pose

The output of a threat modeling is a list of rated threats. Together with the attacker model, they are used for determining the appropriate countermeasures that need to be implemented. Each step is explained in more detail in the following.

Identification of the critical assets A threat is a potential happening that damages and compromises one or multiple assets. A vulnerability is a weakness in the system that makes it possible for threats to occur. An attack is a malicious activity that exploits some vulnerability to damage the assets. In other words, an attack is a method that realizes threats. Assets are the values to be protected. This terminology is summarized in Figure 3.3.

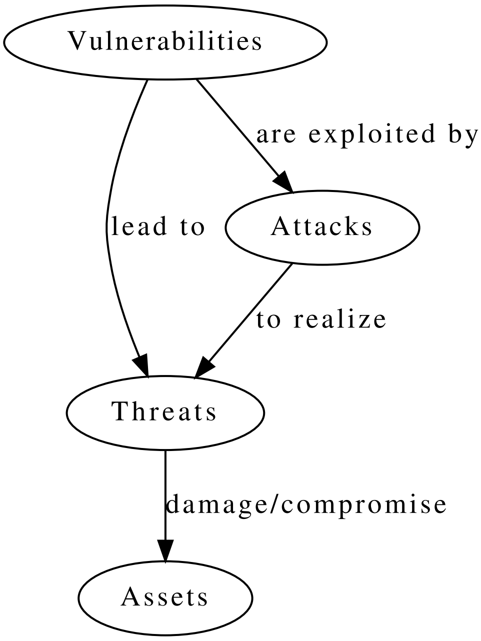


Figure 3.3: Relations between vulnerabilities, attacks, threats and assets - Vulnerabilities lead to threats. Attacks exploit the vulnerabilities to realize the threats for damaging and compromising the assets.

The first step in a threat modeling is the identification of the assets. The goal of adversaries is to compromise and damage the assets by gaining access to them. There would be no threat or attack without assets. Assets can be physical or abstract. Examples of physical assets are a sensor node, a database server, a message, a file, confidential data, etc. Examples of abstract assets are the availability of services which are provided by the physical assets, such as remote programming, as well as other aspects, such as safety, reputation etc.. Assets can interact with each other. Hence, adversaries can use some assets as an entry point to damage other assets in a system.

Network modeling The second step in a threat modeling is the specification of the network model. The goal is to identify the architecture and technologies used in the physical deployment of the application being considered. This step is crucial for identifying the vulnerabilities and threats at the subsequent steps of the modeling process.

Network modeling is composed of three subtasks: identifying the functionality of the application, establishing an architecture diagram, and identifying the technologies used in the architecture.

- Identifying the functionality of the application: The goal is to identify how the assets are used and accessed when the application is being run. This helps with determining, for example, how an access right granted to an entity can be misused to damage the values in the system when the rights are not granted properly.

- Architecture diagram: The goal is to split the application into subcomponents and specify the physical deployment characteristics. This helps with identifying the interactions between the components. This is the basis for determining the trust boundaries within the architecture during the application decomposition at the next step.

- Identifying the technologies: The goal is to identify the technologies used in the implementation of the application. This helps with finding the technology-specific threats and determining the appropriate countermeasures to mitigate them.

Decomposition of the application In this step, the security profile of the application is created. This is done by mainly identifying the trust boundaries, the data flow and the entry points. The goal is to have a high-level look at the application through the adversary's eyes.

- Identifying the trust boundaries: The goal is to identify what is trusted and what not. This helps with identifying e.g., untrusted data input to the application that can be exploited for compromising the security. Once the trust boundaries are determined, the security designer considers, for example, how to authenticate the input data which is untrusted.

- Identifying data flow: The goal is to identify the data flow between the assets and components. This helps with determining the threats when untrusted data flows across the trust boundaries.

- Identifying entry points: The goal is to identify the entry points of the application that may be exploited by adversaries to compromise the security. An entry point can be external, i.e., open to everybody or internal i.e., open to subcomponents only. Thus, external entry points are the main entry points of the attacks. For this reason, identifying entry points helps with identifying the threats that are possible when the security measures protecting the external entry points are bypassed.

Identification of the threats In this step, threats that may compromise the assets are identified. A broadly accepted method used for identifying the threats is the STRIDE model [97]. That is also the method used in this thesis.

Threats that an application may face within the STRIDE model are grouped into six categories:

- **S**poofing identity: using a false identity to gain access to the system.

- **T**ampering with data: malicious modification of data stored in the system or being exchanged between the system's components.

- **R**epudiation: ability of denying to have performed specific actions.

- **I**nformation disclosure: exposure of information to unauthorized entities.

- **D**enial of service (DoS): ability of preventing the availability of services to valid users.
- **E**levation of privilege: gaining privileged access that is sufficient to compromise or destroy the entire system.

Threats are identified by checking whether the risks presented in the STRIDE model apply to each entry point/component of the application. This is done by asking questions like "could an adversary make use of information disclosure during a remote programming to compromise any asset?" The risks answered with "yes" are included in the list of the potential threats.

Once the potential threats are identified, the next step is to determine the attacks that might realize them. An attack is a malicious action that exploits certain vulnerabilities and failures in a system to compromise its security. In other words, attacks describe how to realize threats. A well-known approach used for determining the possible attacks is the attack trees [98]. Attack trees are created by placing each threat being considered as the root node of a tree. The leaves of the trees are then created by adding the conditions/means that might be necessary/used for realizing those threats located at the root node. The leaf nodes can be labeled with AND and OR labels. An AND label implies that all conditions specified on the leaf nodes must occur for a successful attack. In case of an OR label, the occurrence of any condition specified at the leaf nodes is sufficient to realize the threat.

To give an example, assume a remote programming (RP) application. Further assume that software updates disseminated in this RP application contain some cryptographic material. Obtaining the key material carried in the software updates (SUs) during a RP would be a threat to this application. More specifically, this threat belongs to the category of the information disclosure at the STRIDE model. A simple attack tree for this threat is depicted in Figure 3.4. It shows that there are three conditions that must occur for realizing this attack. Firstly, the software update being disseminated must be in plaintext. Secondly, it must contain sensitive key material in plaintext. Finally, the adversary must be able to eavesdrop the remote programming process and recognize the keys embedded in the software updates.

A detailed analysis of potential attacks for remote programming is provided in Section 3.3.1.

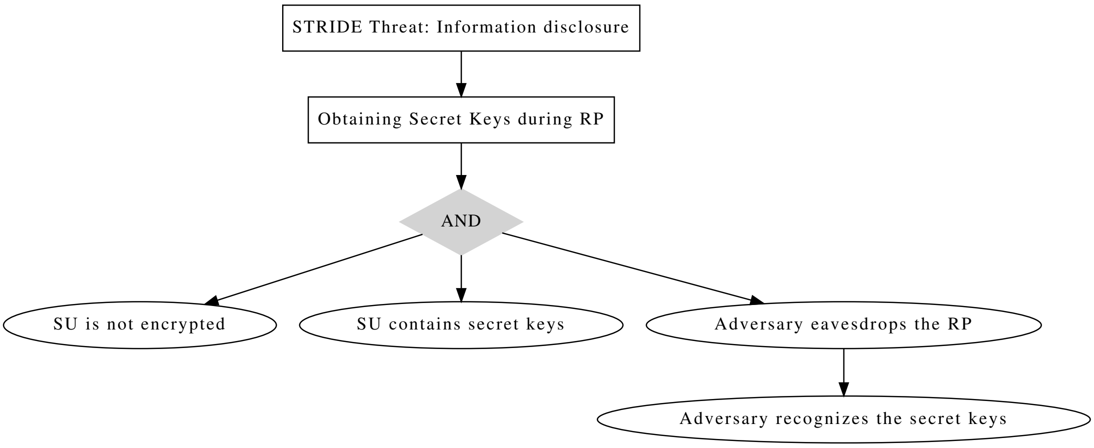

STRIDE Threat: Information disclosure
Obtaining Secret Keys during RP
AND
SU is not encrypted
SU contains secret keys
Adversary eavesdrops the RP
Adversary recognizes the secret keys

Figure 3.4: Attack tree for obtaining secret keys during remote programming - The adversary can obtain the secret keys disseminated with a software update (SU) during a remote programming (RP) only, if all of the following conditions occur. 1- Software update being disseminated is not encrypted. 2- Software update carries secret keys in plaintext embedded into it. 3- The adversary can eavesdrop the software update and recognize the secret keys embedded in the software update. This threat cannot be realized if any of these conditions does not occur.

Rating the threats Rating the threats is the final step in a threat modeling process. Once the potential threats are identified, they are rated according to risks they pose. This enables to mitigate the threats with highest risks first and then address the remaining threats. Due to resource constraints and cost reasons, it would be necessary to ignore the threats occurring with a low probability.

A simple approach for rating the threats is the use of High, Medium, and Low scales. The risk is represented as risk = damage · probability. The threats posing significant risks in terms of occurrence and damages are rated as High. High threats must be mitigated as soon as possible. Medium threats need to be addressed, but with less urgency. Finally, the threats posing low damages and are difficult to realize can be rated as Low. Low threats can be ignored if the cost of a possible mitigation is expensive. This approach is very simple. However, ratings are typically subjective and can vary significantly depending on the security engineer calculating the risks.

Another, more formal, approach to calculate risks is the DREAD model. In this approach, risks are determined by summing up the scores obtained by answering the following questions [8]:

- **D**amage potential: how great is the damage if the vulnerability is exploited?
- **R**eproducibility: how easy is it to reproduce the attack?
- **E**xploitability: how easy is it to launch an attack? How much effort, expertise, and resource is required to implement an attack?
- **A**ffected users: how many users are affected?
- **D**iscoverability: how easy is it to find the vulnerability?

Reproducibility and Exploitability can be considered as the measure of the occurrence probability of a threat. Similarly, remaining items can be considered as the measure of the criticality. It is to note that the questions above can be adapted to the needs of an application scenario. For example, the question “how many users are affected?” can be reformulated as “how many sensor nodes are affected?” while determining the risks of threats for WSN applications.

For rating a threat, every evaluation criteria needs to be scored. This is done by following the scoring scheme represented in Table 3.1. In this scheme, the highest level of risk is scored with 3, the medium level of risk is scored with 2, and finally, the lowest level of risk is scored with 1. Once all questions are scored, the overall risk of a threat is calculated by summing up all those five scores. The maximum score can be 15 ($= 3+3+3+3+3$) and the minimum score can be 5 ($= 1+1+1+1+1$). Hence, a threat with the overall risk score between 5 and 7 can be considered as Low, between 8 and 11 as Medium, and between 12 and 15 as High.

Figure 3.5 shows the risk rating of the threat depicted in Figure 3.4. The threat is that adversaries might obtain secret keys by eavesdropping the dissemination. The overall risk is calculated by summing up the five DREAD scores: Damage potential of this threat is high (3), as the disclosure of any secret key compromises the security of the mechanisms using that key. Reproducibility risk of this threat is medium (2), as the attack can be performed only during a remote programming. That is, the adversary cannot perform the attack whenever he/she wishes, but must wait for the remote programing to occur. Exploitability risk is also medium (2). Although eavesdropping

Abbr.	Criteria	High (3)	Medium (2)	Low (1)
D	Damage Potential	The attacker can subvert the security system; get full trust authorization; run as administrator; upload content.	Leaking sensitive information	Leaking trivial information
R	Reproducibility	The attack can be reproduced every time and does not require a timing window.	The attack can be reproduced, but only with a timing window and a particular race situation.	The attack is very difficult to reproduce, even with knowledge of the security hole.
E	Exploitability	A novice programmer could make the attack in a short time.	A skilled programmer could make the attack, then repeat the steps.	The attack requires an extremely skilled person and in-depth knowledge every time to exploit.
A	Affected users	All users, default configuration, key customers.	Some users, non-default configuration.	Very small percentage of users, obscure feature; affects anonymous users.
D	Discoverability	Published information explains the attack. The vulnerability is found in the most commonly used feature and is very noticeable.	The vulnerability is in a seldom-used part of the product, and only a few users should come across it. It would take some thinking to see malicious use.	The bug is obscure, and it is unlikely that users will work out damage potential.

Table 3.1: DREAD threat-rating table (taken from [8]).

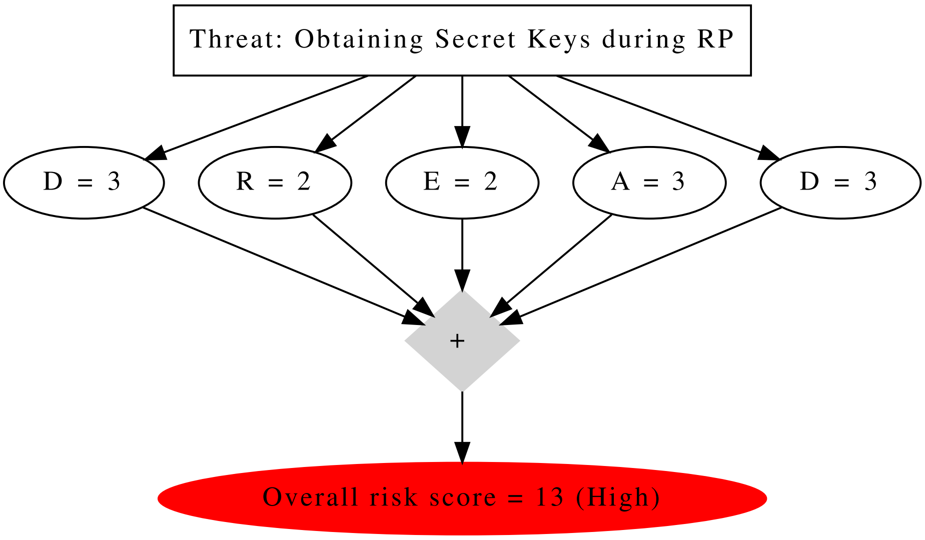

Threat: Obtaining Secret Keys during RP
D = 3
R = 2
E = 2
A = 3
D = 3
+
Overall risk score = 13 (High)

Figure 3.5: Risk calculation example with DREAD model - Consider a remote programming application disseminating software updates containing secret keys that are not encrypted. One potential threat for this application is that adversaries can obtain the secret keys by eavesdropping the dissemination process. The overall risk of this threat is calculated by summing up the risk scores obtained for evaluating the five DREAD criteria: Damage Potential (D), Reproducibility (R), Exploitability (E), Affected users (A), Discoverability (D) which are scored according to Table 3.1.

is an easy attack, the adversary must have a certain level of skills and expertise, for example, to recognize the keys in the software update. The threat “Affected users” can be interpreted as affected sensor nodes in this scenario. The risk of this threat is high (3), as all sensor nodes are affected if the keys used by them are revealed. Finally, discoverability risk is high (3), since the eavesdropping attack is a well known attack. Moreover, the adversary can analyze the dissemination protocol to discover if the software update being disseminated is encrypted or not. The sum of all these threats gives the overall risk score which is 13. As 13 is a high risk, this threat must be addressed as soon as possible.

The final step in a threat modeling is to determine the appropriate countermeasures against the identified threats. The goal is to move the threats with high security risks into the low risk category by applying appropriate mitigation techniques. This is discussed in Section 3.3.3.

3.2.2 Attacker modeling

Attacker modeling is a complementary step to threat modeling while designing security solutions. More precisely, threat modeling is used to identify the threats posing high security risks. An attacker model is used to describe precisely what adversaries can do and what not. Hence, it is the basis for determining the implementability of the identified threats in practice. The outputs of both models are then combined to develop a security design that mitigates the threats posing high security risks and are viable in practice. In fact, capabilities of adversaries are considered in the DREAD model to some extent while evaluating the Exploitability of threats. However, a more detailed and broadly accepted approach is the attacker modeling. This is the reason why this thesis incorporates both models.

A broadly accepted approach to model adversaries is their classification in terms of resources available to them [9]. The classification is done according to their technical knowledge, the tools they use for performing attacks, and their financial budget. As summarized in Table 3.2, each class has the following distinguishing characteristics:

- Class 1: Adversaries belonging to this class often do not have special knowledge of the system being attacked beyond that generally available. Hence, adversaries in this class are generally assumed to be outsiders. They gather the required

Class	Adversary	Knowledge	Tools	Budget/Time	Goal
1	unorganized hackers, script kiddies, small organizations	outsiders, limited	common tools	<$10,000, limited	challenge/prestige
2	organized hackers, criminals, medium organizations	insiders-outsiders, special	special tools	<$100,000, moderate	publicity/money
2	intelligence services, terrorists, large crime organizations, etc.	insiders-outsiders, highly specialized	most advanced tools, laboratory	>$100,000, large	varies

Table 3.2: Classification of adversaries [9].

information on the system, for example, by snooping the messages over open channels. Attacks are launched by making use of the existing vulnerabilities in the system rather than creating new ones. Moreover, they have only a limited time and budget and no special tools. Unorganized amateurs and hackers as well as small organizations are considered in this adversarial class. The goal of attacks is typically personal challenge and prestige.

- Class 2: Adversaries belonging to this class often have special knowledge of the system being attacked. Hence, adversaries in this class might be outsiders as well as insiders. An adversary is considered to be insider if she/he has access to special information such as session keys. Hence, insiders are typically assumed to have physical access to system components. Class 2 adversaries have usually deep knowledge of the system being attacked. Moreover, they have a reasonable time and budget and use special tools. Organized hackers with criminal intent and hackers from academy, medium organizations are considered in this adversarial class. The goal of attacks is typically personal use such as publicity for academicians and economic profit for criminals. Figure 3.6 illustrates the distinguishing characteristics of class 2 adversaries compared to class 1 adversaries.

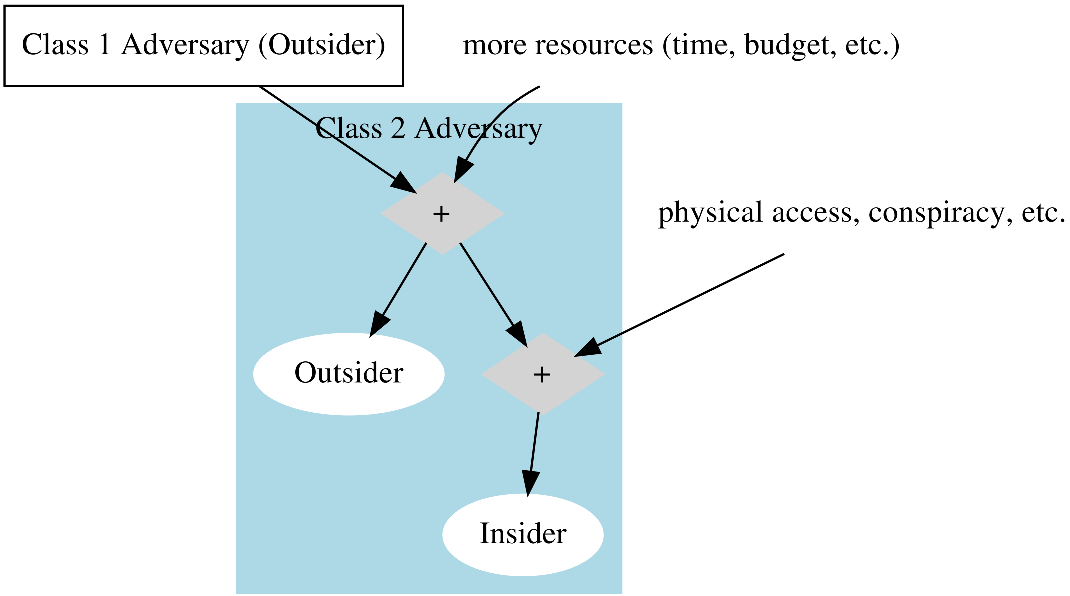

Class 1 Adversary (Outsider)
more resources (time, budget, etc.)
Class 2 Adversary
+
physical access, conspiracy, etc.
Outsider
+
Insider

Figure 3.6: Class 1 Adversary vs. Class 2 Adversary - Class 2 adversaries are distinguished from Class 1 adversaries mainly in terms of resources and information available to them. Class 2 adversaries have much more resources and typically have access to some additional information such as secret keys (obtained e.g. by physical access to system components or conspiracy).

- Class 3: Adversaries belonging to this class have deep knowledge of the system being attacked. They are typically composed of large teams[1] of specialists with an almost unlimited time and budget. Intelligence services, terrorist organizations and large crime organizations are considered in this adversarial class. The goal of attacks depends on the organizations behind them and may vary from stealing state and industrial secrets to incidents with large economic impacts.

Once the adversarial class to be considered is chosen, the next step is to identify the security goals and requirements.

[1] Class 2 adversaries can be a part of the team while launching attacks.

3.2.3 Identifying security goals and parameters

After choosing an appropriate attacker model and identifying the threats to the application to be secured, the next step is to determine the security goals. This is done in two steps: firstly, security goals that mitigate the identified threats are determined. Subsequently, the level of security for each security goal is decided.

In the following, first of all, basic countermeasures against potential threats are provided. Subsequently, general guidelines for choosing the security parameters in accordance with an attacker model are introduced.

Security goals and countermeasures Basically, any attack realizes one or multiple threats to compromise the security of a system. The threats specified in the STRIDE model are spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege. They can be mitigated by applying the countermeasures implementing authentication, integrity, non-repudiation, confidentiality, availability, and authorization. Figure 3.7 shows the potential threats and the security goals that must be achieved to protect against them. Standard means implementing the security goals are depicted in Figure 3.8.

Selection of security parameters Once the countermeasures are identified, the next step in a security design is to choose the appropriate security level that they must provide. Consider the information disclosure threat against the software updates depicted in Figure 3.4. In that threat, the adversary could obtain the keys carried in a software update by snooping the dissemination process. As it can be implied from Figure 3.8, a possible mitigation technique against this threat is to encrypt the software updates. However, two main questions, which are “How strong should the encryption be?” and “How long should the provided security persist?”, remain still unclear. Simply, choosing the maximum level of security might be more than sufficient and waste resources. Hence, a formal way is required to determine what level of security is desired and until when. Typically, this problem is addressed by the attacker model. As described in Section 3.2.2, an attacker model specifies the capabilities of adversaries in terms of resources available to them such as the computation power. Once the attack resources available to an adversary are known, one can choose the security parameters that are sufficient enough to mitigate them for sufficient time.

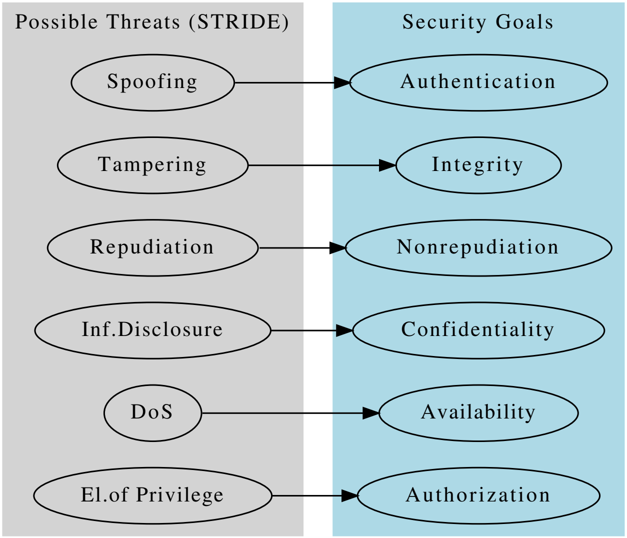


Figure 3.7: Potential threats and their mitigation - Mapping between the STRIDE threats and the cryptographic goals that mitigate them. Authentication, integrity, nonrepudiation, confidentiality, availability, and authorization protect against spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege, respectively.

The basic idea behind this methodology is that an adversary is motivated to launch an attack if its cost is lower than the value of the protected asset [10]. The cost is usually represented in terms of time and money. For example, it is unlikely that an adversary would be willing to spend a million dollar to break the security of an asset whose value is one dollar. Similarly, if an asset to be attacked is valid for a limited time (for example a sensor reading valid for hours), the adversary would not spend a week for an attack to obtain it.

The strength of cryptographic algorithms against attacks is determined by their key size. Keys need to be large enough to protect assets during their lifetime. This is the general rule to be followed when selecting the keys. Table 3.3 shows the security of symmetric keys against class 1, 2 and 3 adversaries. Security levels of asymmetric keys are typically measured by their symmetric equivalences. Table 3.4 shows the symmetric equivalents of various asymmetric key sizes.

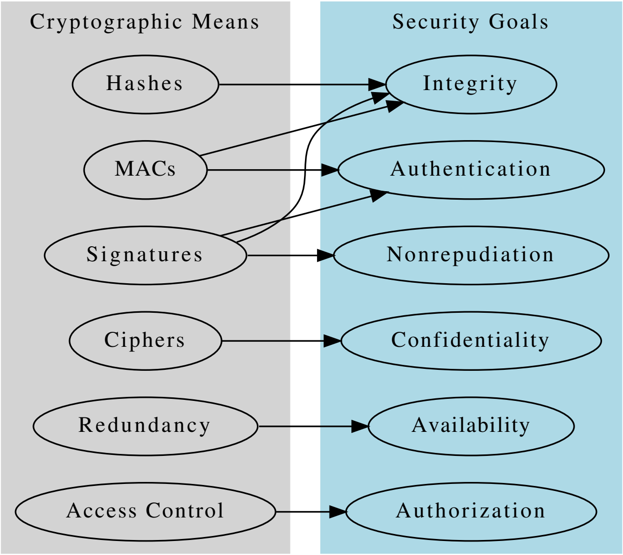


Figure 3.8: Security goals and their realization - Ciphers, signatures, hashes, MACs are used to realize the basic security properties such as confidentiality, authentication, integrity, and nonrepudiation. Authorization is typically realized via access control mechanisms. Redundancy is a broadly accepted means to increase availability.

Security level	Key size	Adversary	Protection level	Comment
1	32-Bit	class 1	real-time	only acceptable for authentication tags
2	64-Bit	class 1	very short-term	should not be used for confidentiality in new systems
3	72-Bit	class 1	medium-term	
		class 2	short-term	
4	80-Bit	class 1	long-term	smallest general-purpose
		class 2	medium-term	
		class 3	very short-term	
5	96-Bit	all	legacy standard	10 years protection
6	112-Bit	all	medium-term	20 years protection
7	128-Bit	all	long-term	30 years protection
8	256-Bit	all	forseeable future	good protection against against quantum computers

Table 3.3: Security level of symmetric keys (adapted from [10])

.

Symmetric key size (Bits)	RSA/DLOG keys (Bits)	EC DLOG keys (Bits)
48	480	96
56	640	112
64	816	128
80	1248	160
112	2432	224
128	3248	256
160	5312	320
192	7936	384
256	15424	512

Table 3.4: Equivalence of symmetric and asymmetric keys with respect to underlying mathematical problems [10].

3.3 Remote programming scenario

In the previous Section, general approaches for threat and attacker modeling are introduced. In this section, by following those approaches, security requirements for remote programming mechanisms are identified.

Terminology of threats The following terminology used in this thesis:

- Eavesdropping messages: The adversary learns the content of the messages being transmitted over a communication channel.

  The communication channel can be wireless (e.g sensor node to sensor node communication) or wired (e.g. communication over Ethernet).

  There are two basic requirements for eavesdropping on a wireless communication. The first one is that the adversary (or a tool controlled by him/her) must be in the range of the wireless communication that he/she wants to eavesdrop on. The second one is that the adversary must possess an appropriate hardware. For instance, to be able to eavesdrop on a sensor node communication based on IEEE 802.15.4, the adversary must equip his/her laptop with a wireless module which is capable of IEEE 802.15.4 standard. Once the adversary has the required hardware, he/she can eavesdrop on a wireless communication using a wireless packet sniffing tool such as AirSNORT.

Eavesdropping on a communication over Ethernet is more challenging. The adversary must have access to the network in which the communication to be eavesdropped on takes place. Typically, computers in an Ethernet network communicate with each other and the Internet through a router or a switch. Thus, to be able to eavesdrop on an Ethernet communication taking place between two endpoints, the adversary must compromise the router/switch connecting those endpoints. Once the adversary has access to the routers/switches, it can eavesdrop on all communication through them using a network analyzer tool such as Wireshark.

- Intercepting messages: Similar to the eavesdropping, the adversary learns the content of the messages being transmitted over a communication channel. The difference is that the intercepted messages do not reach the destination. They are dropped/destroyed by the intercepting adversary.

  In contrast to the eavesdropping, intercepting a wireless communication is relatively challenging. The reason is that the adversary must ensure that wireless messages do not reach the destination. Jamming is a method that is used for intercepting wireless communications. It is also called a Denial-of-Service attack. Jamming is performed as follows: The adversary broadcasts dummy wireless packets at a higher power and frequency to collide with the legitimate wireless packets. Due to packet collisions, the communication between legitimate wireless endpoints is disrupted. To be able to intercept an IEEE 802.15.4 based communication between sensor nodes, a laptop equipped with an IEEE 802.15.4 module would be sufficient. The only thing the adversary needs to do is to broadcast dummy IEEE 802.15.4 packets at high frequencies and power within the area to be jammed. Two directional antennas might be needed to be able to learn messages while jamming.

  To intercept an Ethernet communication, the adversary must drop the packets being transmitted at routers or switches. Thus, similar to eavesdropping, to be able to perform this attack, the adversary needs to compromise the switches or routers through which the communication to be intercepted takes place.

- Inserting messages: The adversary delivers additional (malicious) messages over a communication channel by using the identity of others. Messages can be addressed to a single receiver or multiple receivers. In the former case, the adversary needs to know the address of the receiver. In the latter case, messages are delivered by broadcast or multicast.

Choosing the identity that the adversary uses and determining the addresses of the receivers are the main requirements for inserting new messages into a wireless or Ethernet communication. The adversary can obtain both information by eavesdropping on the communication before performing the attack. Hardware requirements for inserting packets are similar to those for intercepting messages. Thus, in general, any intercepting adversary can also insert new messages into a communication channel.

- Delaying messages: The adversary delays the messages being transmitted over a communication channel.

  Any adversary who can intercept and insert messages can also delay messages. Thus, requirements for delaying messages are the combination of requirements for intercepting and inserting messages.

- Modifying messages: The adversary manipulates the messages being transmitted over a communication channel.

  Any adversary who can delay messages can also manipulate messages. Thus, requirements for modifying messages are identical to the requirements for delaying messages.

- Compromising network components: In order to compromise network components, the adversary has to have a physical access to them. Compromising attacks can be divided into the following categories:

  - Memory reading attacks: The adversary reads the memory content of the component being attacked. The content can be cryptographic material, programs, or application data.

    As shown in [68], compromising a sensor node is very straightforward. The adversary merely needs to extract the sensor node's memory (internal and external flash) to obtain the secret keys and data stored in it. Once the adversary has a physical access to the sensor node, extracting the memory content is easily possible with a laptop, a serial cable and a freely available tool called UISP.

  Compromising other network components such as workstations, routers or switches is also easy. For example, the adversary can read the memory content of a workstation by connecting its hard disk to an external disk reader. A laptop with USB connection would be sufficient to perform this attack. An important thing to note is that access control mechanisms based on username/password pairs do not protect against such an attack.

  - Memory manipulation attacks: The adversary modifies the memory content of the component being attacked. The content can be cryptographic material, programs, or application data.

    The adversary can, for example, reprogram the component being attacked with a malicious software. It can provide the functionality of the legitimate program, but it can also include some hidden functionalities that enable the adversary to perform malicious operations on the component or in the network. Such an attack on a sensor node would require a laptop with a serial cable. In case of a workstation, the adversary would require only to access its hard disk. That is possible with a laptop and an external hard disk reader.

  - Replacement attacks: The adversary replaces the component or its parts with a malicious one which is fully under control of the adversary.

    The purpose of this attack is possibly the same as the memory manipulation attack. However, the adversary might prefer this attack for example due to time constraints. Replacing a component is potentially much faster than modifying its memory content with some external tools.

- Adding malicious components: The adversary add new components in the network which are under his/her control.

Acronyms The following acronyms are used in the remainder of this chapter:

- MD: Monitoring Device

- GW: Gateway Device

- SKN: Sink Node
- SN: Sensor Node
- SU: Software Update
- RP: Remote Programming
- Adv: Adversary

3.3.1 Threat modeling

Threats to a remote programming can be determined by following the five steps introduced in Section 3.2.1: identification of the critical assets, network modeling, decomposition, threat identification, and threat rating.

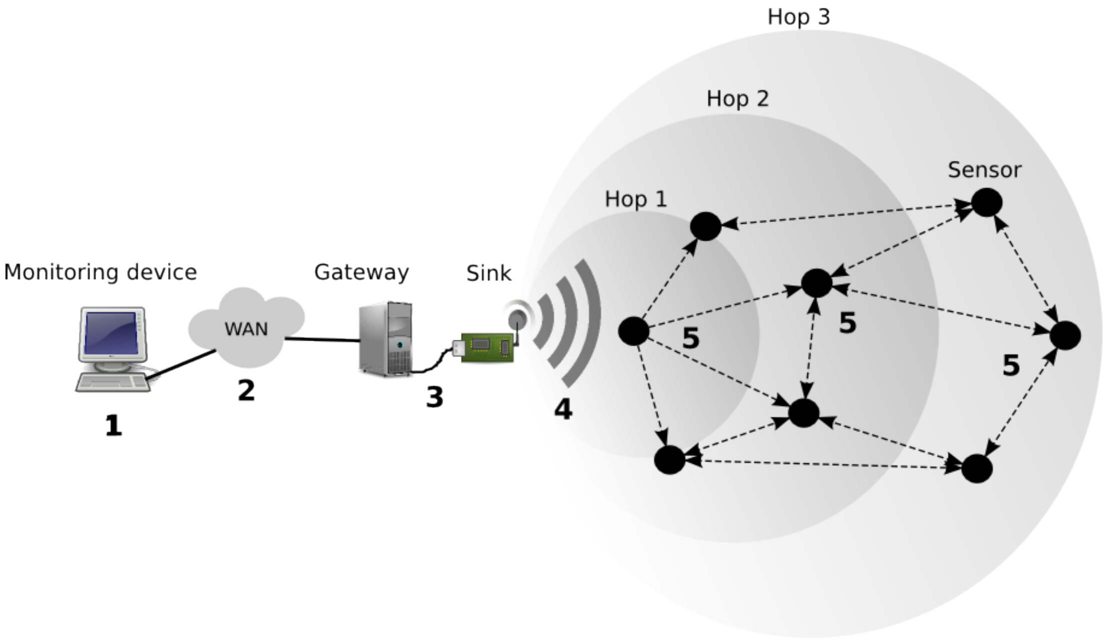


Figure 3.9: Network components of a remote programming mechanism - A remote programming architecture is composed of a monitoring device, a gateway, a sink node, and multiple sensor nodes.

Critical assets The main asset to protect in a remote programming mechanism is the software update being disseminated. As shown in Figure 3.9, a remote programming architecture is composed of a monitoring device, a gateway device, a sink node, and multiple sensor nodes. All these components can be exploited to damage the software update. Thus, the assets to be considered in this thesis are the monitoring device, the gateway device, the sink node, the sensor nodes, as well as the software update being disseminated.

Network model The network model describes the function of each component used in an RP architecture as well as the key technologies they use:

- MD: This component, typically located in an office environment, is the origin of the software updates. It is connected to the gateway device over a wide area network (WAN) such as the Internet. The connection from the monitoring device to the WAN within the office environment is established over an Ethernet network. Hardware resources available to the monitoring device are comparable to those found on workstations.
- GW: This component, typically located next to the deployment field which is typically public, is the interface between the monitoring device and the sink node. It is connected to the monitoring device over a WAN such as the Internet. The connection to the sink node is typically established with a cable over a serial port. It receives the software updates from the monitoring device via WAN and delivers them to the sink node via the serial connection. Hardware resources available to the gateway device are comparable to them found on workstations.
- SKN: This component is located in the communication range of the first-hop sensor nodes to be programmed. Therefore, it is publicly accessible. It is used to disseminate the software updates received from the gateway to the WSN. The dissemination is performed over a wireless channel which is typically based on the IEEE 802.15.4 standard. Hardware resources available to the sink node are comparable to those found on typical sensor nodes.
- SNs: Sensor nodes are the endpoints of an RP mechanism. The goal is to update the software running on them to a new version. Depending on their location with respect to the sink node, sensor nodes receive the updates directly from the sink node (single-hop setting) or from the other nodes (multi-hop setting). Thus, sensor nodes in a multi-hop setting need to disseminate the updates they received

further to the nodes located at the next hop (5). Sensor nodes communicate with each other over a wireless channel which is typically based on the IEEE 802.15.4.

Decomposition Each component is analyzed through the adversary's eyes to determine potential attack points:

- MD: This component manages the remote programming mechanism. Software updates are invoked and controlled via this component. Thus, compromising the monitoring device allows the adversary to take the software updates fully under his/her control. The adversary can perform any malicious activity easily if he/she gains access to this component.

  Another potential attack point is the communication channel to the gateway device. The adversary can manipulate the software updates by tampering with this channel.

- GW: The software updates sent by the monitoring device are passed to the sink node through this component. Thus, compromising the gateway device allows the adversary to tamper with the software updates. The adversary can perform any malicious activity on the updates easily, if he/she gains access to the gateway device.

  Other potential attack points are the communication channels with the monitoring device and the sink node. The adversary can tamper with these channels to compromise the security of the software updates. The gateway device is typically located in an application field where it is potentially accessible to everybody. Thus, compromising the gateway device itself or its communication channels is likely to be easier than compromising the monitoring device.

- SKN: The software updates received from the gateway device are disseminated to the sensor nodes through this component. Thus, compromising the sink node allows the adversary to manipulate the software updates in any possible way. By gaining access to the sink node, the adversary can perform any malicious activity easily.

  The adversary has physical access to the sink node since it is typically deployed in a public environment. This allows the adversary to compromise it physically.

Moreover, the sink node communicates with the sensor nodes over a wireless channel. Thus, the adversary can tamper with the software updates being disseminated.

- SNs: In a multi-hop WSN setting, sensor nodes disseminate the software updates they received to the other sensor nodes. Thus, the adversary can manipulate the software updates during their dissemination within the WSN. Moreover, sensor nodes can be compromised easily, since they are deployed in a public environment.

Identifying the threats The next step after determining the potential attack points is the precise identification of the threats. Once a threat is identified, it needs to be classified according to the STRIDE model. This helps with identifying the potential countermeasures later.

- MD: By gaining access to the monitoring device, the adversary can perform any malicious activity. He/she can take the entire WSN under his/her control by programming sensor nodes with a malicious software. Thus, that is the fundamental threat to be considered.

  The adversary can gain access to the monitoring device possibly in two ways: by physically compromising it and by compromising it through the network.

  Compromising it physically is very easy. If it is not protected with an access control mechanism (e.g. username/password pairs, certificates, etc.), the adversary can simply login to the monitoring device to perform any malicious activity. If it is protected with an access control mechanism, the adversary needs to obtain the access credentials. This can be done by launching e.g., a memory reading attack. Other possibilities are the social engineering and brute-force attacks. These attacks are possible if the adversary is an insider and can physically access to the monitoring device.

  Gaining access through the network is also possible, since the monitoring device establishes a connection with the gateway device. For example, the adversary can perform brute-force attacks on the open ports of the monitoring device (http, ssh, ftp, etc.). Attacks through the network are possible only if the monitoring device is accessible over the network (internal or external).

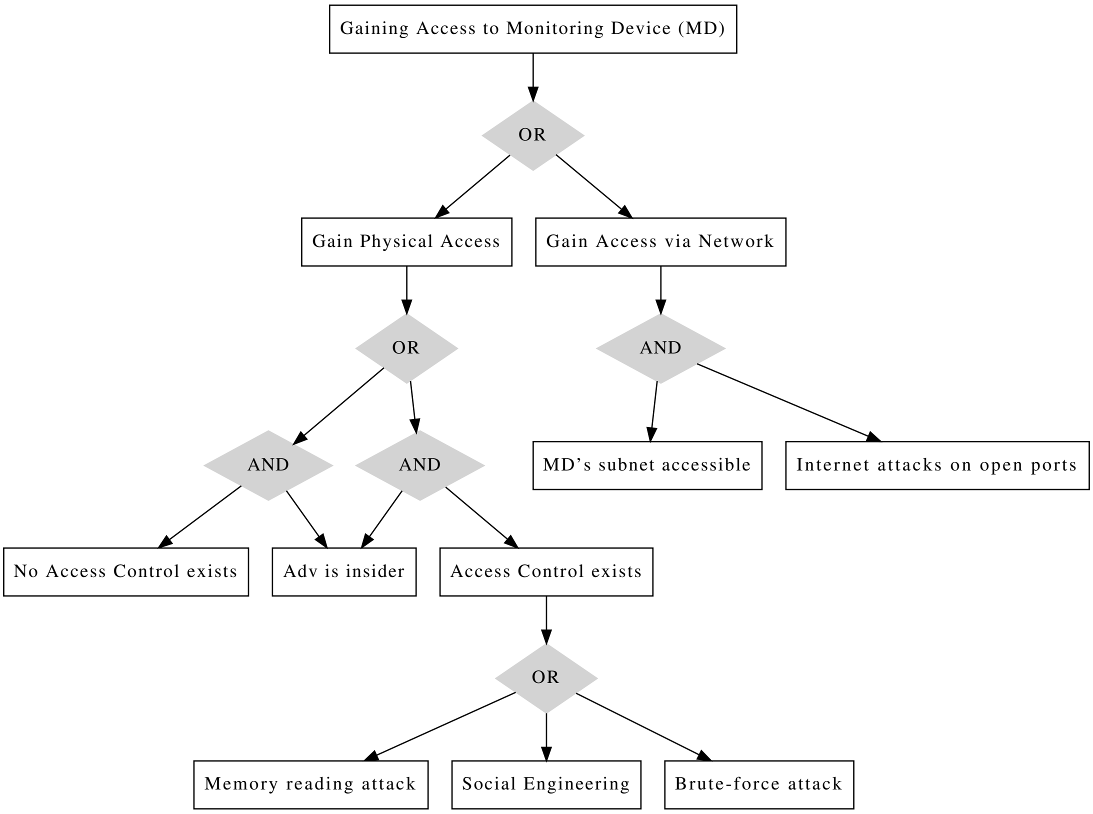


Figure 3.10: Threats to monitoring device for gaining access to it - Gaining access to the MD can be realized via internal or external threats. Internal threats include memory reading attacks, brute-force attacks, and social engineering attacks. External threats include the attacks on the open ports (ssh, ftp, http, etc.) of the MD.

Figure 3.10 summarizes the potential threats to the monitoring device.

- MD-GW communication: Threats to the communication channel between the monitoring device and the gateway can be considered in two groups: threats from the insider adversaries and threats from the outsider adversaries. Insider adversaries are assumed to be the part of the internal office network. They can access the network components such as routers and switches through which the monitoring device communicates with the gateway. Outsider adversaries are assumed to have no access to the internal office network. They are assumed to have access only to the gateway device.

  As the insider adversaries have access to the network components, particularly to the switches and routers, they can compromise them physically. As a result, the insider adversaries can tamper with the software updates transmitted from the monitoring device to the gateway. Malicious activities that they can perform include eavesdropping, intercepting, delaying, modifying, and dropping the software updates before they leave the office network. Moreover, they can insert new software updates by impersonating the monitoring device. These attacks are summarized in Figure 3.11.

  Outsider adversaries are assumed to have physical access to the gateway device. Thus, outsider adversaries can perform a man-in-the-middle attack on the software updates. For this, the adversary can add a malicious device between the gateway device and the Ethernet cable connected with the monitoring device. By doing so, the adversary routes the software updates through the malicious device and can perform all the attacks that insider adversaries can perform. As switches and routers are not required to be compromised, this attack is even likely to be simpler to launch than the insider attacks. These attacks are summarized in Figure 3.12.

- GW: The software updates to be disseminated are routed through the gateway device before being passed to the sink node for the final dissemination to the sensor nodes. Thus, gaining access to the gateway device allows the adversary to perform the attacks similar to those that it can perform after adding a malicious component between the MD and the GW. That is, the adversary can eavesdrop,

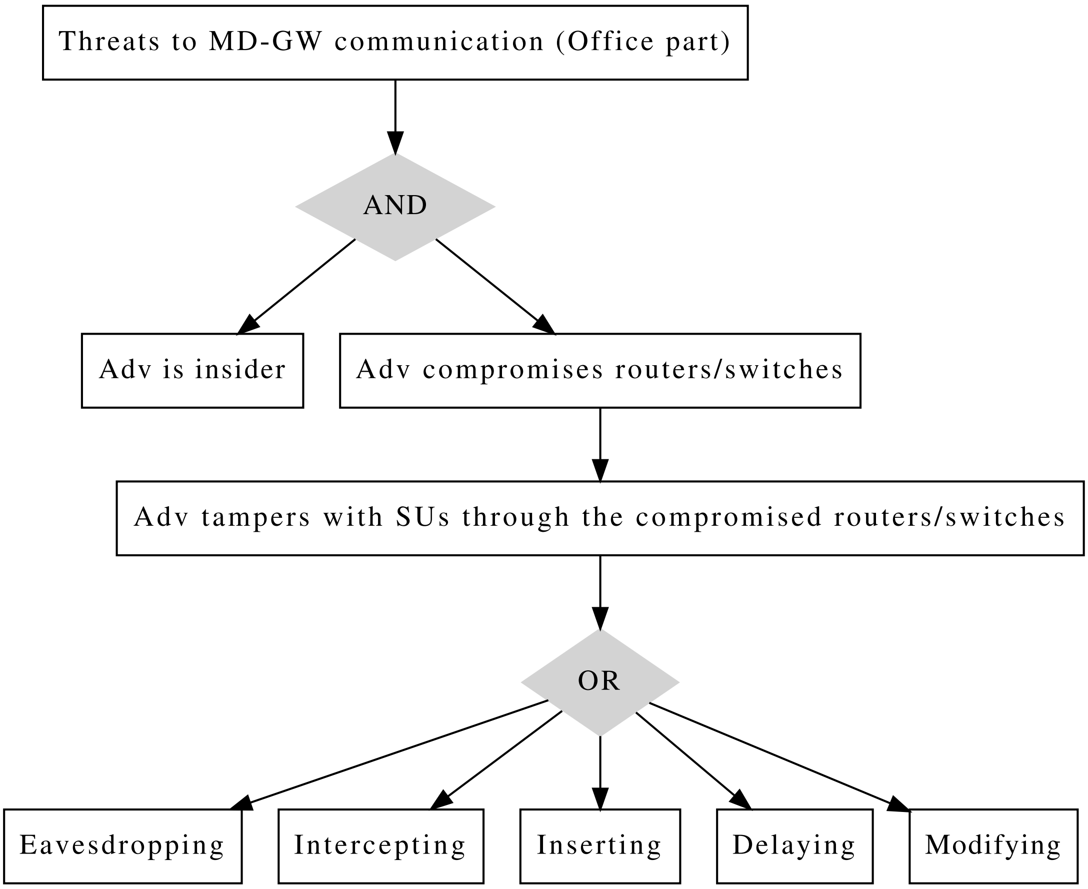

Threats to MD-GW communication (Office part)
AND
Adv is insider
Adv compromises routers/switches
Adv tampers with SUs through the compromised routers/switches
OR
Eavesdropping
Intercepting
Inserting
Delaying
Modifying

Figure 3.11: Threats to the MD-GW communication from insiders - If the adversary (Adv) has access to the internal office network, he/she can tamper with the software updates (SUs) being communicated between the monitoring device (MD) and the gateway (GW).

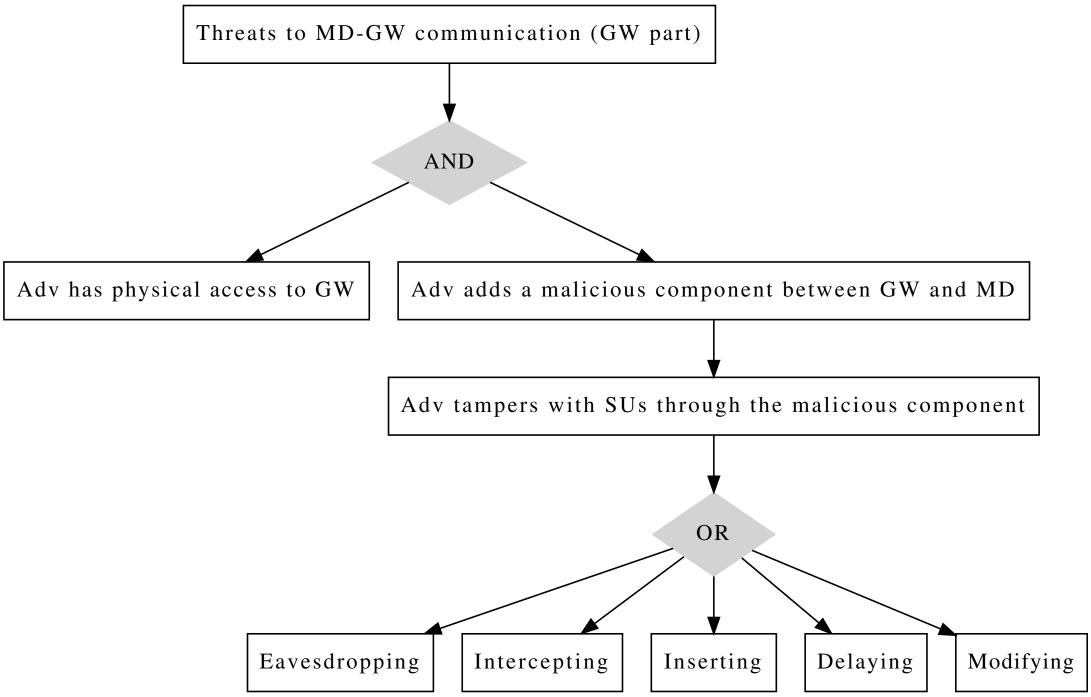

Threats to MD-GW communication (GW part)
AND
Adv has physical access to GW
Adv adds a malicious component between GW and MD
Adv tampers with SUs through the malicious component
OR
Eavesdropping
Intercepting
Inserting
Delaying
Modifying

Figure 3.12: Threats to the MD-GW communication from outsiders - If the adversary (Adv) is outsider and has no access to the internal office network, it can tamper with the software updates (SUs) by adding a malicious component between the GW and the MD. For this, the adversary merely needs to have physical access to the GW. Since all traffic from the MD would be routed though the malicious component under adversary's control, he/she could tamper with the SUs.

intercept, modify, or delay the software updates sent from the monitoring device as well as insert new updates. The adversary might also obtain the secret information stored on the gateway if it exists.

The adversary can gain access to the gateway by physically compromising it or via the Internet. It is to note that (physically) compromising the gateway device is likely to be simpler than compromising the monitoring device. The reason for this is that the gateway device is located at the deployment field of the WSN. Thus, the adversary will possibly have sufficient time for compromising the gateway device physically without being noticed. By physically compromising the gateway, the adversary takes the remote programming protocol fully under his/her control.

Since the gateway communicates with the monitoring device via the Internet, the adversary can also exploit the potential vulnerabilities of the Internet communication to gain access to the gateway. Figure 3.13 summarizes the threats that enable the adversary to gain access to the gateway device.

- GW-SKN communication: The gateway and the sink node are connected with a cable. Software updates are passed through the gateway to the sink node. Due to the public deployment, the adversary can tamper with the cable communication. It can, for example, perform a man-in-the-middle attack by adding a malicious component and redirecting the communication through it. Malicious activities that the adversary can perform are the same as the threats to the GW-MD communication. They are eavesdropping, replacing, and delaying the software updates sent to the sink node. These attacks are summarized in Figure 3.14.

- SKN: Threats to the sink node are similar to those threats that apply to the gateway device. The difference is that gaining access to the sink node is possibly easier than gaining access to the gateway device. Once the adversary has the physical access to the sink node, he/she can reprogram it with his/her own software (or replaces it with a malicious one) to manipulate its functionality. By doing so, the adversary can eavesdrop, delay, drop or intercept the software updates sent from the monitoring device as well as insert new updates. Moreover, the adversary obtains the secret information stored on the SKN if it exists. Figure 3.15 summarizes the threats that enable the adversary to take the sink node under his/her control.

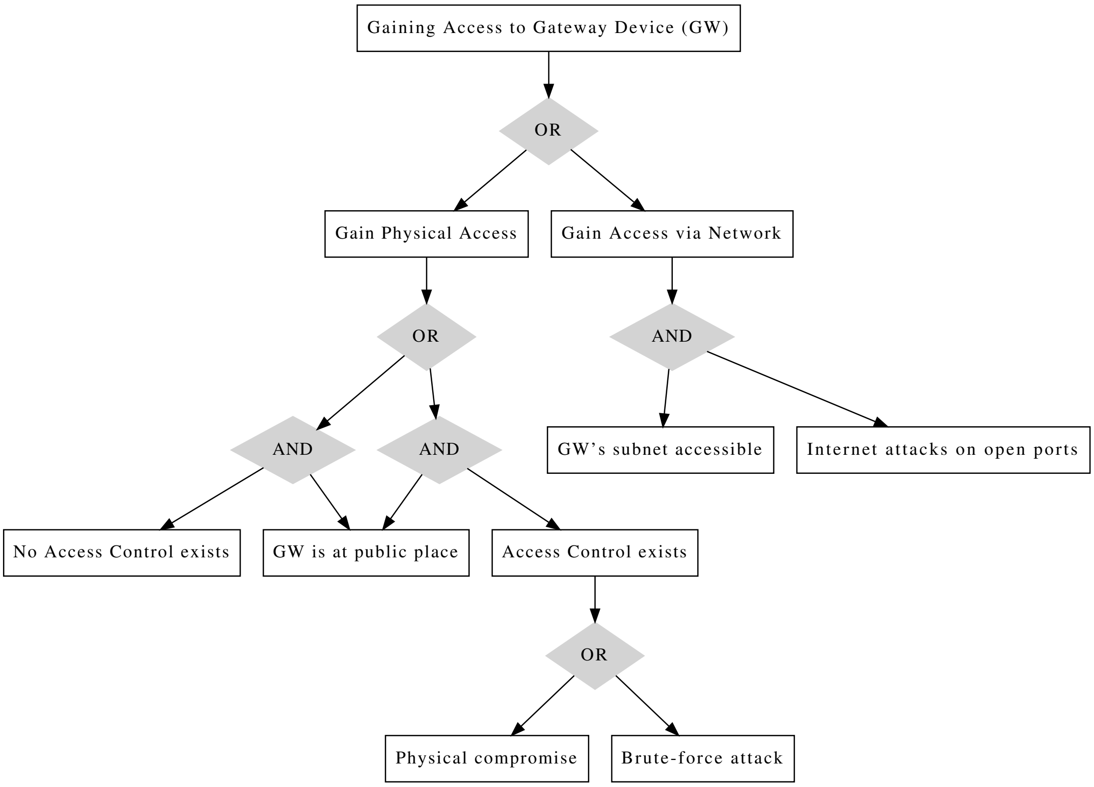


Figure 3.13: Threats to gateway device for gaining access to it - The adversary can gain access to the GW by physically compromising it or by exploiting the network vulnerabilities.

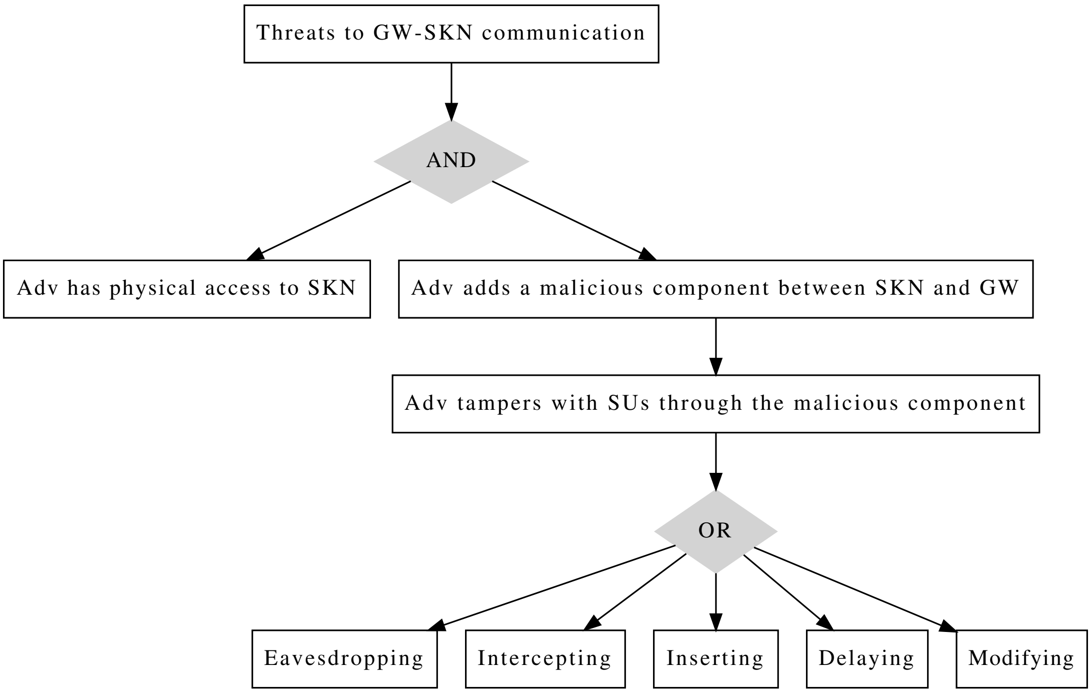

Threats to GW-SKN communication
AND
Adv has physical access to SKN
Adv adds a malicious component between SKN and GW
Adv tampers with SUs through the malicious component
OR
Eavesdropping
Intercepting
Inserting
Delaying
Modifying

Figure 3.14: Threats to the GW-SKN communication - If the adversary has access to the sink node, it can add a malicious component between the GW and the SKN. Since all traffic from the GW would be routed though the malicious component under adversary's control, he/she could tamper with the SUs in any possible way.

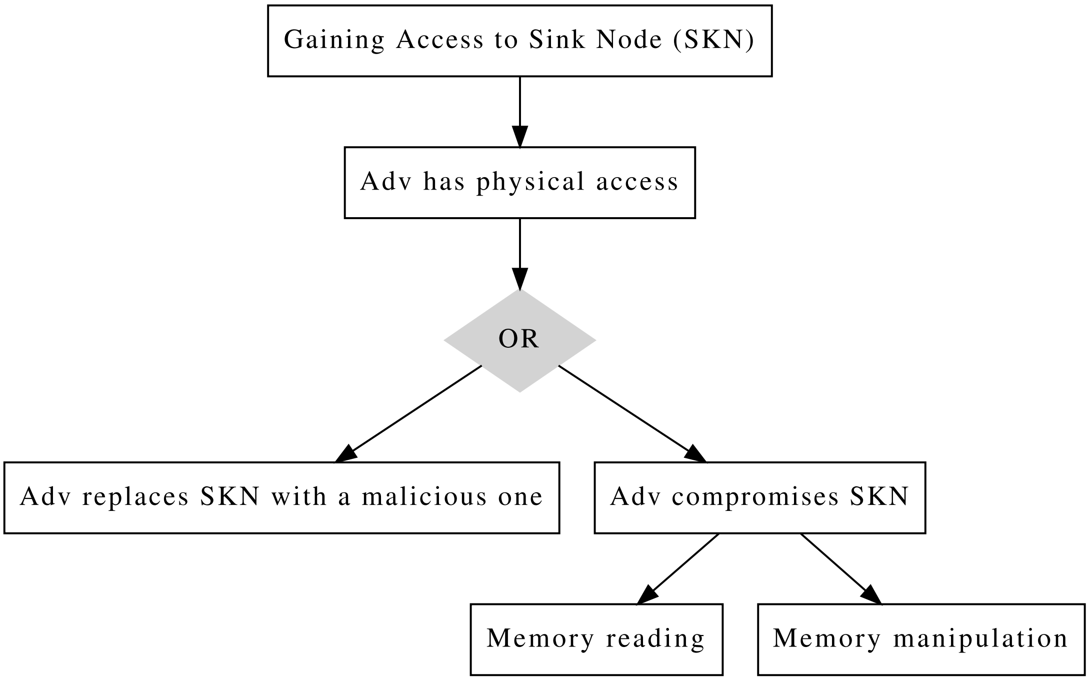


Figure 3.15: Threats to sink node - The adversary can replace the sink node with a malicious one as well as compromise it physically to take the remote programming under his/her control.

- SKN-SN communication: The sink node and the sensor nodes communicate wirelessly. Hence, the adversary can eavesdrop, intercept[1], modify, delay, and insert new packets during a remote programming operation. Since the entry point of the software updates to the WSN is this communication channel, any manipulation made on data is spread to all sensor nodes. The adversary is required to be within the communication range of the sink node to tamper with this communication channel. Figure 3.16 summarizes these threats.

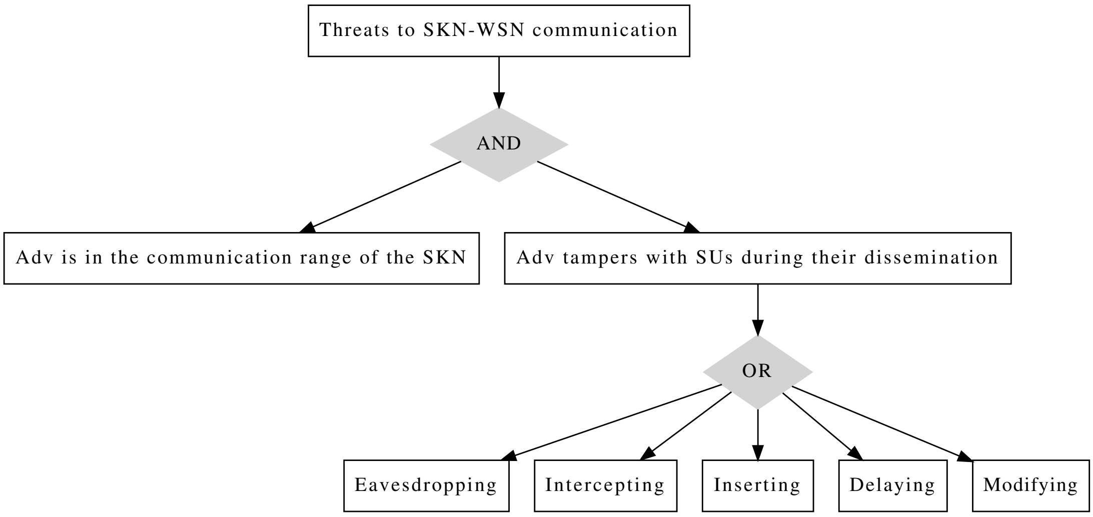

Threats to SKN-WSN communication
AND
Adv is in the communication range of the SKN
Adv tampers with SUs during their dissemination
OR
Eavesdropping
Intercepting
Inserting
Delaying
Modifying

Figure 3.16: Threats to the SKN-SN communication - If the adversary is in the communication range of the sink node, it tamper with the communication by eavesdropping, intercepting, delaying, modifying, and dropping the software updates as well as by inserting new updates.

- SNs: The adversary can physically compromise the sensor nodes, for example, to obtain any sensitive information stored on them. Moreover, the adversary can add malicious sensor nodes or reprogram them with a malicious software. Malicious or compromised nodes can manipulate the data transmitted to other sensor nodes by delaying, modifying, intercepting it, or by inserting new messages into the communication. For example, they can provide wrong information in order to be selected as a source node during the software dissemination. Once they are selected as a source node, they can infect all of its subscribers with malicious data.

[1] Jamming attacks are a subset of intercepting attacks.

- SN-SN communication: The adversary can eavesdrop and manipulate data exchanged between the sensor nodes.

Threat classification using STRIDE model After identifying the threats, the next step is their classification using the STRIDE model. All threats identified above are based on the attacks (threat terminology) introduced at the beginning of this section. Thus, they are classified with respect to attacks they are based on. That is, if a threat is realized by e.g. eavesdropping on a communication channel, that threat is classified as Information Disclosure in the STRIDE model. Table 3.5 shows those classifications.

Rating the threats The final step is to rate the identified threats in terms of the damage risks they pose. Table 3.5 shows the identified threats and the risks they pose. Risks are computed according to the DREAD model as follows:

- Damage potential:
  - Gaining access via network attacks:
    - MD: Damage potential of this attack is high (3), since the adversary can take the remote programming mechanism fully under his/her control, if he/she controls the monitoring device.
    - GW: Damage potential of this attack is medium (2), since even if the adversary has the GW under his control, he/she can only tamper with the software updates which are sent from the MD during their dissemination. That is, the adversary can merely eavesdrop, intercept, delay, drop, and modify the software updates. However, if the adversary does not have the cryptographic material (e.g. signature keys) used by the MD to authenticate software updates, he/she still cannot generate valid software updates.
  - Gaining access by a physical compromise:
    - MD: Same as the threats to the MD described above.
    - GW: Same as the threats to the GW described above.

Component	Threats	STRIDE classification	DREAD risks
MD	Gaining access via network attacks	Spoofing, Elevation of privilege	13 (high)
	Gaining access by a physical compromise	Spoofing, Information disclosure, Tampering	14 (high)
MD-GW Communication	Eavesdropping	Information disclosure	11 (medium)
	Intercepting	DoS	12 (high)
	Inserting	Spoofing	14 (high)
	Delaying	Spoofing, DoS	11 (medium)
	Modifying	Tampering	12 (high)
GW	Gaining access via network attacks	Spoofing, Elevation of privilege	12 (high)
	Gaining access by a physical compromise	Spoofing, Information disclosure, Tampering	14 (high)
GW-SKN Communication	Eavesdropping	Information disclosure	12 (high)
	Intercepting	DoS	13 (high)
	Inserting	Spoofing	15 (high)
	Delaying	Spoofing, DoS	12 (high)
	Modifying	Tampering	13 (high)
SKN	Replacement of SKN with a malicious one	Spoofing	14 (high)
	Memory Reading	Information disclosure	13 (high)
	Memory Manipulation	Spoofing, Tampering	13 (high)
SKN-SN Communication	Eavesdropping	Information disclosure	12 (high)
	Intercepting	DoS	12 (high)
	Inserting	Spoofing	15 (high)
	Delaying	Spoofing, DoS	10 (medium)
	Modifying	Tampering	12 (high)
SN	Replacement of SN with a malicious one	Spoofing	12 (high)
	Adding malicious SNs	Spoofing	12 (high)
	Memory Reading	Information disclosure	12 (high)
	Memory Manipulation	Spoofing, Tampering	11 (medium)
SN-SN Communication	Eavesdropping	Information disclosure	10 (medium)
	Intercepting	DoS	8 (medium)
	Inserting	Spoofing	11 (medium)
	Delaying	Spoofing, DoS	8 (medium)
	Modifying	Tampering	8 (medium)

Table 3.5: Classification of threats to a remote programming mechanism using the STRIDE model and their risk scores according to the DREAD model.

    - SKN: Damage potential of this attack is medium (2), since compromising the SKN allows the adversary to launch the same attacks as the attacks that he/she can perform when the GW is compromised.
    - SN: Damage potential of this attack is medium (2), since compromising the SNs allows the adversary to launch the same attacks as the attacks that he/she can perform when the SKN is compromised.
  - Eavesdropping on a communication:
    - MD-GW: Damage potential of this attack is medium (2), since the information/cryptographic material that the adversary may obtain is limited with the information carried in the software updates.
    - GW-SKN: Same as the threats to the MD-GW communication.
    - SKN-SN: Same as the threats to the MD-GW communication.
    - SN-SN: Same as the threats to the MD-GW communication.
  - Intercepting a communication:
    - MD-GW: Damage potential of this attack is high (3), since the adversary prevents the sensor nodes from updating their software by launching this attack. Moreover, the adversary obtains the information/cryptographic material carried in the software updates.
    - GW-SKN: Same as the threats to the MD-GW communication..
    - SKN-SN: Same as the threats to the MD-GW communication.
    - SN-SN: Damage potential of this attack is low (1), since the adversary can prevent only those sensor nodes, which are within the communication range of the attacked node, from receiving the software updates.
  - Inserting new messages into a communication:
    - MD-GW: Damage potential of this attack is high (3), since the adversary can pollute the software updates being sent to the sensor nodes.
    - GW-SKN: Same as the threats to the MD-GW communication.
    - SKN-SN: Same as the threats to the MD-GW communication.
    - SN-SN: Damage potential of this attack is low (1), since the adversary can affect only those sensor nodes which are within the communication range of the node being attacked.

- Delaying a communication:
    - ∗ MD-GW: Damage potential of this attack is medium (2), since the software updates reach the sensor nodes. The damage that the adversary causes is the increased energy consumption during the remote programming. Moreover, the adversary obtains the information/cryptographic material carried in the software updates.
    - ∗ GW-SKN: Same as the threats to the MD-GW communication.
    - ∗ SKN-SN: Same as the threats to the MD-GW communication.
    - ∗ SN-SN: Damage potential of this attack is low (1), since the adversary can affect only those sensor nodes which are within the communication range of the node being attacked.
- Modifying a communication:
    - ∗ MD-GW: Damage potential of this attack is high (3), since the adversary can cause the same damage as the damage that he/she causes by inserting new messages into the communication channel.
    - ∗ GW-SKN: Same as the threats to the MD-GW communication.
    - ∗ SKN-SN: Same as the threats to the MD-GW communication.
    - ∗ SN-SN: Damage potential of this attack is low (1), since the adversary can affect only those sensor nodes which are within the communication range of the node being attacked.

- Reproducibility:
    - Gaining access via network attacks:
        - ∗ MD: Reproducibility of this attack is high (3), since the monitoring device is always online due to the Internet connection to the Gateway device. The adversary has unlimited time to compromise the monitoring device and can perform the attack at any time.
        - ∗ GW: Reproducibility of this attack is high (3), since the adversary can tamper with all subsequent software updates after he/she gain access to the GW.
    - Gaining access by a physical compromise:

  - MD: Reproducibility of this attack is high (3). If the adversary has access to the monitoring device, he/she can potentially perform the attack at any time.
  - GW: Same as the threats to the GW described above.
  - SKN: Same as the threats to the GW.
  - SN: Same as the threats to the SKN.
- Eavesdropping on a communication:
  - MD-GW: Reproducibility of this attack is low (1), since the adversary can perform the attack only during a remote programming operation. He/she needs to wait for the software updates to be able to eavesdrop on them.
  - GW-SKN: Same as the threats to the MD-GW communication.
  - SKN-SN: Same as the threats to the GW-SKN communication.
  - SN-SN: Same as the threats to the SKN-SN communication.
- Intercepting a communication:
  - MD-GW: Reproducibility of this attack is low (1), since the adversary can perform the attack only during a remote programming operation. He/she needs to wait for the software updates to be able to intercept them.
  - GW-SKN: Same as the threats to the MD-GW communication.
  - SKN-SN: Same as the threats to the GW-SKN communication.
  - SN-SN: Same as the threats to the SKN-SN communication.
- Inserting new messages into a communication:
  - MD-GW: Reproducibility of this attack is high (3), since the adversary can send malicious updates at any time by claiming himself/herself to be the MD.
  - GW-SKN: Same as the threats to the MD-GW communication.
  - SKN-SN: Same as the threats to the GW-SKN communication.
  - SN-SN: Same as the threats to the SKN-SN communication.
- Delaying a communication:

        - MD-GW: Reproducibility of this attack is low (1), since the adversary can perform the attack only during a remote programming operation. He/she needs to wait for the software updates to be able to delay them.
        - GW-SKN: Same as the threats to the MD-GW communication.
        - SKN-SN: Same as the threats to the GW-SKN communication.
        - SN-SN: Same as the threats to the SKN-SN communication.
    - Modifying a communication:
        - MD-GW: Reproducibility of this attack is low (1), since the adversary can perform the attack only during a remote programming operation. He/she needs to wait for the software updates to be able to modify them.
        - GW-SKN: Same as the threats to the MD-GW communication.
        - SKN-SN: Same as the threats to the GW-SKN communication.
        - SN-SN: Same as the threats to the SKN-SN communication.

- Exploitability:
    - Gaining access via network attacks:
        - MD: Exploitability of this attack is low (1) due to following reason: the monitoring device should have some security vulnerabilities that the adversary can exploit to get access into it over the network. Typically, devices connected to an office network are maintained by an administrator who regularly applies security patches to them. Thus, it is unlikely that the monitoring device runs services or programs which contain obvious security vulnerabilities.
        - GW: Same as the GW.
    - Gaining access by a physical compromise:
        - MD: Exploitability of this attack is medium (2). It is not low since the adversary only needs to open the cover of the MD, for example to get access to the hard disk. It is not high, since the adversary still needs to extract the e.g., secret keys from the hard disk using some external tools which require a certain level of expertise.

    - GW: Exploitability of this attack is high (3), since the adversary has sufficient time to compromise the GW and analyze the cryptographic material stored in it.
    - SKN: Same as the threats to the GW.
    - SN: Same as the threats to the SKN.
  - Eavesdropping on a communication:
    - MD-GW: Exploitability is medium (2), since the adversary only needs to compromise the switches/routers through which the communication takes place. However, compromising the network components presents a certain level of challenge.
    - GW-SKN: Exploitability is high (3), since the adversary only needs to compromise the switches/routers through which the communication takes place. However, compromising the communication between the GW and the SKN is likely to be easier than compromising the communication between the MD-GW. The reason is that the former is accessible physically for everyone and the attack can be performed, for example, by replacing the SKN with a malicious one.
    - SKN-SN: Same as the GW-SKN communication. Important to note that, eavesdropping on the SKN-SN communication is easier than eavesdropping on the MD-GW and the GW-SKN communication. The reason is that the former does not require a physical access to the communication channel. Being within the wireless communication range is sufficient to perform the attack.
    - SN-SN: Same as the threats to the SKN-SN communication.
  - Intercepting a communication:
    - MD-GW: This threat poses the same risks as the risk that the eavesdropping threat posses.
    - GW-SKN: Same as eavesdropping on the GW-SKN communication.
    - SKN-SN: Exploitability is medium (2), since the adversary must perform a jamming attack to prevent the sensor nodes from receiving the software updates.

    - SN-SN: Same as the threats to the SKN-SN communication.
  - Inserting new messages into a communication:
    - MD-GW: Same as intercepting the MD-GW communication.
    - GW-SKN: Same as intercepting the GW-SKN communication.
    - SKN-SN:Exploitability is high (3), since the adversary can send any kind of messages to the sensor nodes by claiming his/her identity as the SKN or the MD. The adversary only needs to be within the communication range of the sensor nodes.
    - SN-SN: Same as the threats to the SKN-SN communication.
  - Delaying a communication:
    - MD-GW: Same as intercepting threats to the MD-GW communication.
    - GW-SKN: Same as intercepting threats to the GW-SKN communication.
    - SKN-SN: Exploitability is low (1), since the adversary must perform a jamming attack to prevent the sensor nodes from receiving the software updates and subsequently send the jammed packets again after some delay.
    - SN-SN: Exploitability is medium (2), since the attack can be performed by adding a malicious sensor node in the WSN that delays the packets before forwarding to the other nodes.
  - Modifying a communication:
    - MD-GW: Same as intercepting threats to the MD-GW communication.
    - GW-SKN: Same as intercepting threats to the GW-SKN communication.
    - SKN-SN: Same as intercepting threats to the SKN-SN communication.
    - SN-SN: Same as delaying threats to the SN-SN communication.

- Affected sensor nodes:
  - Gaining access via network attacks:

  - MD: The risk is high (3). Once the adversary gains access to the monitoring device, he/she can affect all sensor nodes that are managed by the remote programming mechanism.
  - GW: Same as the threats to the MD.
- Gaining access by a physical compromise:
  - MD: Same as the threats to the MD described above.
  - GW: Same as the threats to the GW described above.
  - SKN: Same as the threats to the GW.
  - SN: The risk is low (1). The adversary can affect only those sensor nodes which are within the communication range of the nodes under his/her control.
- Eavesdropping on a communication:
  - MD-GW: The risk is high (3). The adversary can obtain the information/secret material sent to any sensor node within the WSN.
  - GW-SKN: Same as the threats to the MD-GW communication.
  - SKN-SN: Same as the threats to the GW-SKN communication.
  - SN-SN: The risk is low (1). The adversary can obtain the information/secret material from the eavesdropped communication channel only.
- Intercepting a communication:
  - MD-GW: The risk is high (3). All sensor nodes are affected by this attack.
  - GW-SKN: The risk is high (3). All sensor nodes are affected by this attack.
  - SKN-SN: The risk is high (3). All sensor nodes are affected by this attack.
  - SN-SN: The risk is low (1). This attack affects only the sensor nodes which are within the range of the intercepted communication.
- Inserting new messages into a communication:
  - MD-GW: The risk is high (3). All sensor nodes are affected by this attack.

        * GW-SKN: The risk is high (3). All sensor nodes are affected by this attack.
        * SKN-SN: The risk is high (3). All sensor nodes are affected by this attack.
        * SN-SN: The risk is low (1). Inserted messages are received by only those sensor nodes that are within the communication range of the attack.
    - Delaying a communication:
        * MD-GW: The risk is high (3). All sensor nodes are affected by this attack.
        * GW-SKN: The risk is high (3). All sensor nodes are affected by this attack.
        * SKN-SN: The risk is high (3). All sensor nodes are affected by this attack.
        * SN-SN: The risk is low (1). This attack affects only the sensor nodes which are within the range of the delayed communication.
    - Modifying a communication:
        * MD-GW: The risk is high (3). All sensor nodes are affected by this attack.
        * GW-SKN: The risk is high (3). All sensor nodes are affected by this attack.
        * SKN-SN: The risk is high (3). All sensor nodes are affected by this attack.
        * SN-SN: The risk is low (1). Modified messages are received by only those sensor nodes that are within the communication range of the attack.

- Discoverability:
    - Gaining access via network attacks:
        * MD: Discoverability of this attack is high (3). The adversary can scan the open ports on the monitoring device. Using this information, he/she can discover the services running on the monitoring device. For example, if the ssh port is open, the adversary can launch a brute-force attack to discover the password.

  - GW: Same as the MD.
- Gaining access by a physical compromise:
  - MD: Discoverability of this attack is high (3). The adversary knows that the MD contains some information valuable to him/her for compromising the software updates.
  - GW: Same as the threats to the MD.
  - SKN: Same as the threats to GW.
  - SN: Same as the threats to SKN.
- Eavesdropping on a communication:
  - MD-GW: Discoverability of this attack is high (3). How to perform an eavesdropping attack is well documented.
  - GW-SKN: Same as the threats to the MD-GW communication.
  - SKN-SN: Same as the threats to the GW-SKN communication.
  - SN-SN: Same as the threats to the SKN-SN communication.
- Intercepting a communication:
  - MD-GW: Discoverability of this attack is high (3). How to perform an intercepting attack is well documented.
  - GW-SKN: Same as the threats to the MD-GW communication.
  - SKN-SN: Same as the threats to the GW-SKN communication.
  - SN-SN: Same as the threats to the SKN-SN communication.
- Inserting new messages into a communication:
  - MD-GW: Discoverability of this attack is high (3). How to insert new messages into a communication channel is well documented.
  - GW-SKN: Same as the threats to the MD-GW communication.
  - SKN-SN: Same as the threats to the GW-SKN communication.
  - SN-SN: Same as the threats to the SKN-SN communication.
- Delaying a communication:
  - MD-GW: Discoverability of this attack is high (3). How to delay messages during their transmission over a communication channel is well documented.

    - GW-SKN: Same as the threats to the MD-GW communication.
    - SKN-SN: Same as the threats to the GW-SKN communication.
    - SN-SN: Same as the threats to the SKN-SN communication.
  - Modifying a communication:
    - MD-GW: Discoverability of this attack is high (3). How to modify messages during their transmission over a communication channel is well documented.
    - GW-SKN: Same as the threats to the MD-GW communication.
    - SKN-SN: Same as the threats to the GW-SKN communication.
    - SN-SN: Same as the threats to the SKN-SN communication.

3.3.2 Attacker modeling

The next step is to determine the realistic threats that need to be mitigated by applying appropriate security mechanisms. For this, first of all, the adversarial class to be considered needs to be chosen. Based on this choice, the realistic threats are determined.

This thesis assumes that the adversaries, who wish to attack to remote programming, belong to the class 2 described in Section 3.2.2. Furthermore, it is assumed that the adversaries are insiders and have physical access to all components of a remote programming mechanism. The class 2 adversaries are assumed to have sufficient budget and technical support. Thus, all of the threats posing medium or high risks must be mitigated.

3.3.3 Identifying security goals and parameters

- Security goals for the MD:
  - Gaining access via network attacks: The MD should be protected via an authentication and authorization mechanism.

    Only entities with a valid identity should be allowed to login to the MD by using an authentication mechanism e.g. based on a password or one-time token scheme. Passwords should be chosen randomly and long enough to mitigate, for example, the brute-force attacks.

  An authorization mechanism based on the access control should be applied to prevent the elevation of privilege. Kernel extensions such as the SELinux, implementing the mandatory access control mechanism, can be used for this purpose.

  - Gaining access by a physical compromise: The MD should be protected with an entity authentication mechanism as well as an authorization mechanism as described above.

    Additionally, cryptographic material stored in the MD should not be accessible to the adversaries even if they can obtain the content of the hard disk via a memory reading attack. This can be achieved by encrypting the hard disk using a hard disk encryption mechanism or by using an external media to store the cryptographic material. An example of such an external media is a smart card which is under control of operator.

- Security goals for the MD-GW communication:

  - Eavesdropping: Software updates and any other data being transmitted should be encrypted to prevent eavesdropping adversaries from gaining sensitive information.
  - Intercepting: Software updates and any other data being transmitted should be encrypted to prevent intercepting adversaries from gaining sensitive information.

    An intercepting adversary can drop or destroy messages. In order to mitigate such attacks, redundancy can be applied. For this, two additional devices can be configured as the MD and the GW, respectively that are activated only if an attack on the communication between the master devices is suspected.
  - Inserting: Software updates and any other data being transmitted should be authenticated to mitigate message inserting attacks.

    Furthermore, the connection between the MD and the GW should be established after a successful authentication. For this purpose, e.g. HTTPS can be used.
  - Delaying: The combination of the countermeasures against the intercepting and inserting adversaries can be applied to mitigate delaying adversaries.

  - Modifying: The integrity of the software updates and any other data being transmitted should be protected to mitigate them against modifications. For example, hash chains in a combination with a signature mechanism can be used to achieve this goal.

- Security goals for the GW: Hardware resources available to the GW are similar to them that are available to the MD. Moreover, threats to the GW are the same as the threats to the MD. Thus, the security goals identified for the MD are applicable to the GW.

- Security goals for the GW-SKN communication:

  - Eavesdropping: Software updates and any other data being transmitted should be encrypted to prevent eavesdropping adversaries from gaining sensitive information.
  - Intercepting: Software updates and any other data being transmitted should be encrypted to prevent intercepting adversaries from gaining sensitive information.

    An intercepting adversary can drop or destroy messages. In order to mitigate such attacks, redundancy can be applied. For this, two additional device can be configured as the GW and the SKN, respectively that are activated only if an attack on the communication between the master devices is suspected.
  - Inserting: Software updates and any other data being transmitted should be authenticated to mitigate message inserting attacks.

    Furthermore, the connection between the GW and the SKN should be established after a successful authentication. For this purpose, e.g. HTTPS can be used. The hardware resources available to the SKN are similar to them they are available to the sensor nodes. Thus, HTTPS on the SKN node can be implemented using e.g. Sizzle [23].
  - Delaying: The combination of the countermeasures against the intercepting and inserting adversaries can also be applied to mitigate the delaying adversaries.

  - Modifying: The integrity of the software updates and any other data being transmitted should be protected to mitigate them against modifications. For example, hash chains in a combination with a signature mechanism can be used to achieve this goal.

- Security goals for the SKN:
  - Replacement of the SKN with a malicious one: Sensor nodes should accept the software updates and any other data sent from the SKN only if they are authentic. However, sensor nodes are seriously resource constrained. Thus, an efficient mechanism for authenticating the software updates and other data is required.
  - Memory reading attacks: Software updates and any other data stored in the SKN should be encrypted to mitigate this kind of attacks. Moreover, the sink node ideally should not store any cryptographic material which are not public.
  - Memory manipulation: Mitigating this attack using only cryptography is difficult. The reason is that any secret material stored in the SKN can be obtained by memory reading attacks. Thus, possible mitigation techniques basing on authentication would not work, unless the SKN is equipped with a tamper-resistant hardware protecting the stored cryptographic material.

- Security goals for the SKN-SN communication: Although the sensor nodes and the SKN have the similar hardware resources, the power resource available to the SKN is unlimited. The reason is that the SKN is powered by the GW. Thus, countermeasures based on HTTPS (and Sizzle) cannot be applied to the sensor nodes. The security mechanisms should be based on the security of data being transmitted. Moreover, some remote programming mechanisms apply e.g. Fountain-code based data transmission to achieve redundancy. Thus, the security mechanisms should support packet encodings as well.
  - Eavesdropping: Software updates and any other data being transmitted should be encrypted to prevent eavesdropping adversaries from gaining sensitive information.

  - Intercepting: An intercepting adversary can drop or destroy messages. In order to mitigate such attacks, some remote programming mechanisms apply e.g. Fountain-code based data transmission. Thus, the encryption and authentication mechanisms should support the packet encodings.
  - Inserting: Software updates and any other data being transmitted should be authenticated to mitigate message inserting attacks.
  - Delaying: The combination of the countermeasures against the intercepting and inserting adversaries can be applied to mitigate the delaying adversaries.
  - Modifying: The integrity of the software updates and any other data being transmitted should be protected to mitigate them against modifications. For example, hash chains in a combination with a signature mechanism can be used to achieve this goal.

- Security goals for the SN:
  - Replacement of the SKN with a malicious one or adding new malicious nodes: Sensor nodes should accept the software updates from other nodes only if they are authentic. Thus, an efficient mechanism for authenticating the data being disseminated is required.
  - Memory reading attacks: Software updates and any other data stored in the SN should be encrypted to mitigate this attack. Moreover, sensor nodes ideally should not store any cryptographic material which are not public.
  - Memory manipulation: Mitigating this attack using only cryptography is difficult. Any secret material stored in a SN can be obtained by memory reading attacks. Thus, possible mitigation techniques basing on authentication would not work unless the SN is equipped with a tamper resistant hardware providing a tamper-proof storage for cryptographic material.
- Security goals for the SN-SN communication: The same as the security goals identified for the SKN-SN communication.

Component	Threats	Security goals	Potential Means
MD, GW	Gaining access via network attacks	entity authentication and authorization	one-time tokens, passwords, SElinux
	Gaining access by a physical compromise	entity authentication and authorization, encryption	one-time tokens, passwords, hard disk encryption
MD-GW, GW-SKN Communication	Eavesdropping	encryption	HTTPS, SU encryption
	Intercepting	encryption, redundancy	HTTPS, SU encryption, multiple MDs and GWs
	Inserting	authentication	HTTPS, SU authentication
	Delaying	encryption, authentication	HTTPS, SU authentication & encryption
	Modifying	integrity	HTTPS, SU integrity
SKN,SN	Replacement of SKN with a malicious one	authentication	SU authentication
	Memory Reading	encryption	SU encryption
	Memory Manipulation	tampering	tamper proof hardware
SKN-SN, SN-SN Communication	Eavesdropping	encryption	SU encryption
	Intercepting	encryption, redundancy	SU encryption, Fountain codes
	Inserting	authentication	SU authentication
	Delaying	encryption, authentication	SU authentication & encryption
	Modifying	integrity	SU integrity

Table 3.6: Realistic threats to a remote programming mechanism and potential countermeasures.

The analysis above is summarized in Table 3.6. The smallest level of general-purpose security against the class 2 adversaries requires 80-Bit symmetric keys. Hence, symmetric keys required in this thesis are chosen 80-bit long.

3.4 Scope of the thesis

In the previous section, security goals for the remote programming scenario in WSNs are identified. Some of the identified goals can be realized by applying standard solutions such as HTTPS, password based authentication mechanisms, one-time tokens as well as SELinux. Therefore, they are not discussed in the remainder of this thesis any more.

However, securing software updates to protect their authenticity, confidentiality as well as integrity is an important challenge that needs to be solved. Providing efficient solutions for software authenticity, integrity and confidentiality is the main focus of this thesis.

In summary, this thesis provides security mechanisms for achieving:

- software update confidentiality to protect the software updates against an eavesdropping adversary.
- software update integrity and authenticity to prevent the sensor nodes from receiving malicious or corrupted updates.
- DoS protection by preventing an adversary from exploiting the packet encodings to corrupt the complete software updates by modifying only a few packets.

4

Cryptographic Primitives and Code Size optimized Toolbox

As analyzed in Chapter 2, using only symmetric algorithms on resource limited sensor platforms is a better choice when a code size optimized implementation is required. This section describes how a code-size-optimized cryptographic toolbox can be realized using only a single symmetric cryptography primitive, block cipher. It contains all essential cryptographic primitives for obtaining the basic security goals. More precisely, it is composed of a hash function, which is the fundamental primitive for almost all security solutions, a MAC function for protecting message authenticity and integrity, and finally a signature scheme, for providing non-repudiation in addition to authenticity and integrity protection. The signature scheme is a dedicated design for the secure remote programming mechanism. Encryption mechanisms are not considered in our analysis. The reason is that this thesis uses a hardware implementation of AES provided in the radio chip whenever encryption is required.

This chapter is organized as follows: first, all primitives are described shortly. Subsequently, possible construction mechanisms for them are analyzed in terms of their code size, security and performance. Finally, the most appropriate construction method leading to the best code size is determined thorough analysis.

4.1 Hash functions

Definition A hash function is a function that maps arbitrary-length binary strings into binary strings of fixed length [7] (see Figure 4.1).

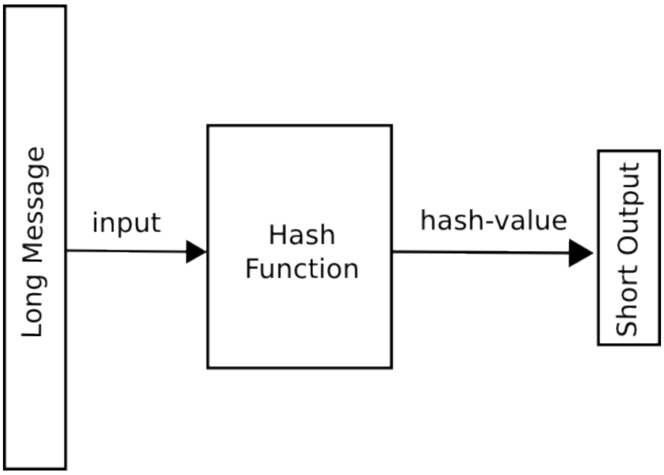


Figure 4.1: General hash principle - A hash function is a function that maps a message of arbitrary-length to an output of fixed length. The output is referred to as hash-value [7].

4.1.1 Properties and requirements

A function h is a hash function if it satisfies the following requirements [7]:

Compression h maps an input of arbitrary-length binary string x into an output of fixed-length binary string $h(x)$ [7].

Ease of computation For every input x, $h(x)$ is easy to compute [7].

- Praxis relevance: One of the important applications of hash functions is the modification detection in large downloads from the Internet[1]. For detecting any modification, the hash-values of the files (e.g. software) to be downloaded are computed and published on the Internet. Anyone, who downloaded a file, verifies its content by comparing its hash-value, which is computed locally, with the hash-value that was computed by the owner of the file and provided in the Internet.

[1] Modifications can be intentional or can occur due to e.g., communication errors.

If both hash-values are equal, the user can be sure that the file has not been modified. The files, that need to be hashed in this scenario, might be several gigabytes large. Hence, an ideal hash function should be able to hash even very large data very quickly.

Preimage resistance h is a one-way function. That is, given any hash-value $h(x)$, where the corresponding input x is unknown, it is computationally infeasible to find an input x' such that $h(x') = h(x)$ (see Figure 4.2) [7].

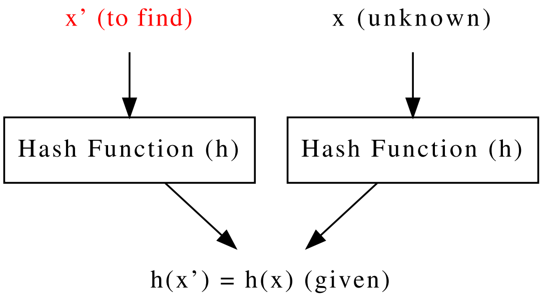


Figure 4.2: Preimage resistance - Given any hash-value $h(x)$, it is computationally hard to find an input x' that hashes to the given output $h(x)$.

- Praxis relevance: Protecting password files, for example the ones stored in an operating system (OS), is one of the most common application scenarios of hash functions. Instead of storing the passwords in plaintext, their hash-values are stored in a password file. When a password verification is required, for example when a user wants to login, the hash-value of the password, entered by the user, is computed and compared with the hash-value stored in the password file. The login is allowed only if both hash-values are equal. Obviously, a hash function to be used in such an application scenario must be preimage resistant. Otherwise, anyone, who can access the password file, can also compute the actual passwords by using the hash-values stored in it.

- Ideal security level: An n-bit hash function has ideal preimage security if the most efficient attack for producing a preimage takes approximately 2^n hash evaluations [7].

Second preimage resistance Given any input x, it is computationally infeasible to find an input x' (different from x) such that $h(x') = h(x)$ (see Figure 4.3). Second preimage resistance is also called weak collision resistance [7].

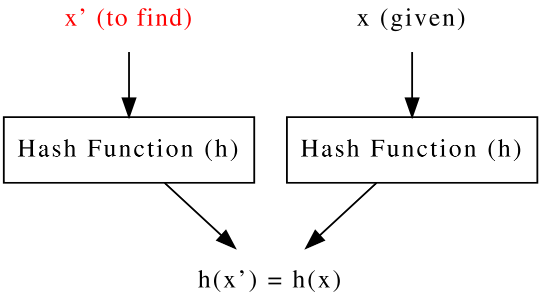


Figure 4.3: Second preimage resistance - Given any input x, it is to find another input x' (different from x) such that $h(x) = h(x')$.

- Praxis relevance: Consider the application scenario given for easy computation above. The hash-values are typically stored in a different location (e.g. in another FTP server) than the files themselves to improve security. The files might even be distributed via a peer-to-peer network, such as torrents, to allow for fast downloads. For example, the software images of Ubuntu Linux, a very famous Linux distribution, are officially encouraged to be download from torrent networks. The reason for this is to reduce the bandwidth costs as well as to improve the download speeds. However, the hash-values of those images are stored in a server managed by Ubuntu.

  Now, suppose that an adversary wants to fool the users by a fake Ubuntu image. The adversary has access both to the correct software image (s) and its hash-value $h(s)$, since they are available for everybody in the Internet. Now, assume that the hash function h used for computing $h(s)$ is not second preimage resistant. As h is not second preimage resistant, computing a fake image s' with $h(s') = h(s)$ is easy for the adversary. Users would install the fake content s', since it hashes to the expected hash-value $h(s)$. It is to note that the adversary does not need to compromise any server for performing this attack.

  Example 1 *Assume that an adversary has downloaded a good version of an OS image (s) which is depicted in Listing 4.1. Further, suppose that the OS image* *was protected with an 8-bit hash-value. The adversary can produce a malicious OS image (s') with a valid hash-value by launching a second preimage attack as follows: He/she writes a malicious image that contains an 8-bit dummy counter. Subsequently, the adversary produces multiple versions of that malicious image which differ only in terms of the value assigned to the counter. Each malicious version s_j hashes to a different hash-value. The adversary stops when a hash-value match is found (i.e. $h(s) = h(s'_j)$ for some j). Listing 4.2 represents such a match found at the 64th attempt*[1]*, i.e. for $j = 64$. In practice, hash functions with much larger outputs (e.g. 80 bits) are used to mitigate such second-preimage attacks.*

- Ideal security level: An n-bit hash function has ideal second-preimage security, if the most efficient attack for producing a second-preimage takes approximately 2^n hash evaluations [7].

Listing 4.1: Good Image (s)

```
public class OSCode {
  public static void main(String[] args) {

    System.out.println("This␣is␣a␣good␣OS␣image.");

  }
}
```

Listing 4.2: Bad Image ($s'_{j=64}$)

```
public class OSCode {
  public static void main(String[] args) {
    int versionCounter = 64;
    System.out.println("This␣is␣a␣malicious␣OS␣image");
    /* Adversary does bad things here...*/
  }
}
```

Collision resistance It is computationally infeasible to find two inputs x and x' such that $h(x') = h(x)$. Collision resistance is also called strong collision resistance [7].

- Praxis relevance: An important application scenario of hash functions is their use in combination with digital signatures. Signing small data is much faster than signing large data. Furthermore, signatures of small data are in general much shorter than signatures of large data. Hash functions compress large data into small data, called hash-value. Signatures are typically computed over the hash-value of data to be signed. This allows to obtain optimal performance in terms of signing performance as well as signature size.

[1] 8-bit hash-values were obtained by taking the last byte of the 32-byte outputs of SHA-2. Java version was 1.6.0_26.

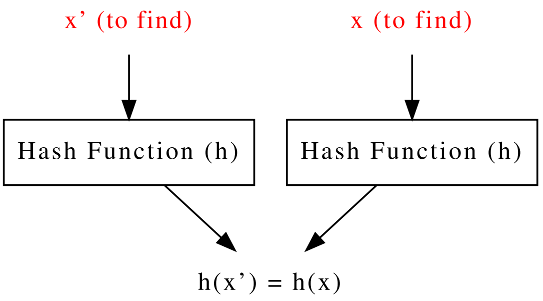


Figure 4.4: Collision resistance - It is to find two distinct inputs x' and x such that $h(x) = h(x')$.

Now, consider an outsourcing company writing software applications (e.g., for banks, industry etc.) with an insider adversary. The adversary wants to deliver a malicious software s', containing a backdoor, to the clients instead of a good software s. The goal of the adversary is to prepare a malicious software with $h(s) = h(s')$. Assume that h is an n-bit hash function. The adversary can proceed as follows: He/she prepares $\lambda = v \cdot 2^{(n/2)}$ identical versions of the good software s. This can be done easily by adding e.g. a dummy counter in the code and increasing it by one at each version[1]. All versions of s together with their hash-values $(s_i, h(s_i))$ are stored. Similarly, λ identical versions of the malicious code are prepared and stored together with their hash-values $(s'_i, h(s'_i))$. According to the birthday problem, the probability of having two software versions s_i and s'_j with the same hash-value (for some i, j) is approximately

$$Pr[h(s_i) = h(s'_j)] \approx 1 - e^{-v^2} \tag{4.1}$$

[99]. That is 0.63 for $v = 1$ (i.e., $\lambda = 2^{n/2}$ software-hash pairs) and 0.98 for $v = 2$ (i.e., $\lambda = 2^{(n+2)/2}$ software-hash pairs). This means, there is a good chance of having success even for $v = 1$. Once the adversary finds such a version pair with the same hash-value, the adversary sends the good version s_i to the client for the inspection. The client signs the hash-value $h(s_i)$ of the good software s_i to confirm its release. However, the adversary releases the malicious version s'_j with $h(s_i) = h(s'_j)$. The signature originally produced for the good one would be

[1] In case of a text document, multiple versions can be produced by adding e.g. empty spaces.

valid also for the malicious one, since the hash-values for the both versions are the same. Attacks based on this principle are referred to as birthday attacks.

Example 2 *Two java programs (s, s'), hashing to the same 8-bit*[1] *hash-value with SHA-2, are given in Listings 4.3 and 4.4. Multiple versions of each program were simply produced by incrementing a dummy variable (int versionCounter). In total, 29 hash evaluations were necessary to find a collision between two versions of the good and bad software in this example ($h(s_{i=14}) = h(s'_{j=7})$). This example shows that it is easy to find a collision for a hash function whose output size is not large enough. Hence, hash functions in practice need to be chosen with much larger outputs (e.g. 160 bits) for mitigating such collision attacks. It is to note that, due to Equation 4.1, the effective security against collisions attacks is the half of the output size of the underlying hash function. That is, if a hash function produces 160-bit outputs, its security against collision attacks is only 80 bits.*

- Ideal security level: An n-bit hash function has ideal collision security, if the most efficient attack for finding a collision takes approximately $2^{n/2}$ hash evaluations [7].

Listing 4.3: Good Version ($s_{i=14}$)

```
public class BankSoftware {
  public static void main(String[] args) {
      int versionCounter = 14;
      System.out.println("This␣is␣a␣good␣version");

  }
}
```

Listing 4.4: Malicious Version ($s'_{j=7}$)

```
public class BankSoftware {
  public static void main(String[] args) {
      int versionCounter = 7;
      System.out.println("This␣is␣a␣malicious␣version");
      /* Adversary does bad things here...*/
  }
}
```

4.1.2 Construction of hash functions

Generally speaking, an arbitrary length hash function is composed of two main components: a compression function and an iteration algorithm. A compression function is a function that maps fixed-length inputs to fixed-length outputs, where the input size is slightly larger than the output size [100]. A compression function is often called a fixed-length hash function [100]. The iteration algorithm is used for converting a

[1] 8-bit hash-values were obtained by taking the last byte of the 32-byte outputs of SHA-2. Java version was 1.6.0_26.

compression function into an arbitrary-length hash function. Hence, arbitrary-length hash functions in practice are also called iterated hash functions. A compression function can be built from a block cipher (or a customized block cipher like component) or from a computationally hard problem from number theory. The general construction principle, that many practical hash functions are based on, is sketched in Figure 4.5.

As shown in Figure 4.5, there are four main building blocks for hash functions. These are block ciphers, compression algorithms and functions, and finally, iteration algorithms. In the following, each of those building blocks is described in more detail.

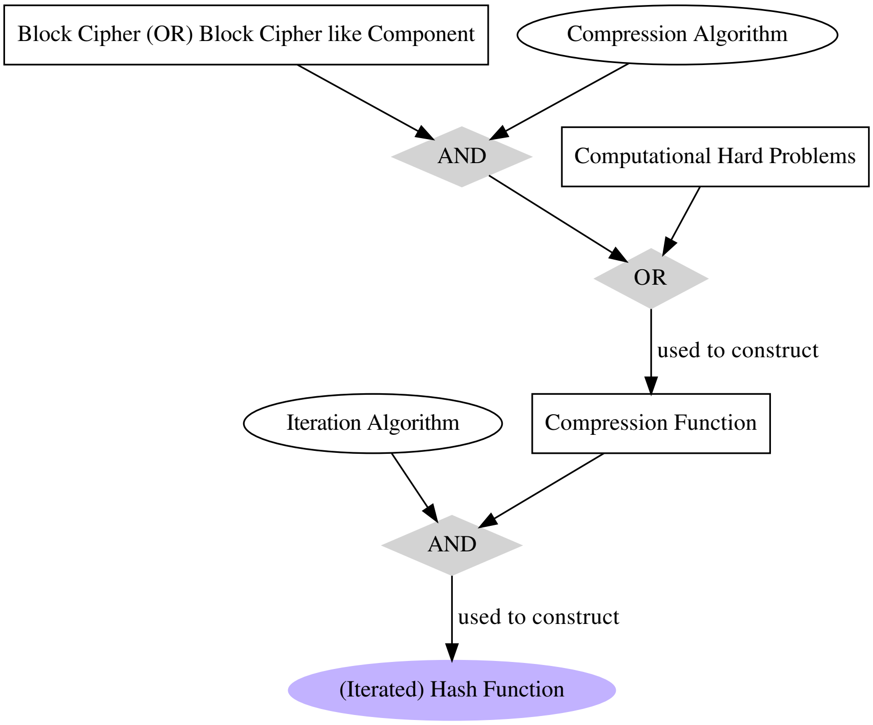


Figure 4.5: General construction principle of several hash functions - The main building block of many practical hash functions, e.g. SHA-1, SHA-2, MD5, is a block cipher like component. It is used to construct a one-way compression function. A compression function can also be built from a number-theoretic hard problem. Finally, an iteration algorithm (mostly Merkle-Damgard) is used to transform the fixed-length compression function into an arbitrary length hash function.

Block ciphers Block ciphers, among computationally hard problems, are the basic building blocks of compression functions and, hence, hash functions.

A block cipher p is an efficient keyed pseudorandom permutation [100]. It takes two inputs, x and k, and produces a single output, y. That is, $p(k, x) = y$. The input k is called key. The length of the input is equal to the length of the output, i.e., $|x| = |y|$. It is called block length. The length of the key is typically, but not necessarily, equal to the block length, i.e., $|k| = |x| = |y|$.

A block cipher is a compression function, since it takes two inputs of length $|k| + |x|$ and produces an output of length $|x|$ $(= |y|)$, only. When this fact is considered, the following question naturally arises: "Why do hash functions use a compression algorithm for building a compression function from a block cipher instead of using it directly if it already compresses?". The answer is that block ciphers are efficiently invertible. That is, for a block cipher p, there is an algorithm p^{-1} that computes $p^{-1}(k, y) = x$ efficiently. This obviously contradicts the preimage-resistance property required from hash functions.

Block ciphers, such as AES, DES, Threefish etc., are the practical construction of (strong) pseudorandom permutations [100]. Hence, they can be used as the main building block of hash functions. A recent example is the Skein. It is based on Threefish and was one of the five finalists in the NIST hash function competition for SHA-3. Some other practical hash functions such as SHA-1, SHA-2 an MD5 are based on a dedicated block cipher designed for hashing. They are based on a dedicated block cipher instead of a standard one due to the performance and security[1] reasons.

In the following, several methods for constructing a compression function from a block cipher are presented.

Compression algorithms and functions A compression function can be built from a block cipher or from a number-theoretic hard problem:

- Construction from block ciphers: An algorithm is required to turn a block cipher into a one-way compression function. There are 12 such secure algorithms [101]. In fact, 8 of them are vulnerable to fixed-point attacks[2]. However, fixed-point

[1] This does not mean that hash functions based on a standard block cipher is not secure. It is meant that hash functions based on a standard block cipher are typically limited in terms of their output lengths, hence, the security level that they can provide.

[2] A fixed-point of a block cipher p is an input x with $p(k, x) = x$ for key k. If p is a secure block cipher, the probability of finding a fixed-point is negligible.

attacks are not realistic if the underlying block cipher is secure [101]. Davies-Meyer, Miyaguchi-Preneel, and Matyas-Meyer-Oseas constructions are the most well-known ones [7]. This thesis describes only the Davies-Meyer construction due to its much broader use in practice. The reader is referred to [101] for a nice overview of the remaining methods.

Davies-Meyer compression function: Consider a block cipher $p : K \times X \rightarrow X$ taking two inputs, a key $k \in K$ and a message $x \in X$, and producing an output $y \in X$. Furthermore, for simplicity assume that $|k| = |x| = |y|$.

Given p, a one-way compression function $c : K \times X \rightarrow X$, mapping two inputs of length $|k| + |x|$ in total into the outputs of length $|x|$, can be constructed as follows [7]:

$$c(k, x) = p(x, k) \oplus k. \tag{4.2}$$

If p is a secure block cipher, the construction above yields a one-way compression function (i.e. a fixed-length hash function) by setting k with a constant value, e.g., $k = 0x0...0$. Notice that the compression function passes the message as the key to the underlying block cipher.

Any fixed-length hash function can be converted into an arbitrary-length hash function by using an iteration algorithm as described in the subsequent paragraph.

Proposition 4.1.1 *The Davies-Meyer compression function c defined above is preimage resistant (i.e. one-way) if the underlying block cipher p is secure.*

PROOF (Sketch): Assume that c is not one-way. Furthermore, assume that an output z with $z = c(k, x)$ is given, where the corresponding input x is unknown. Since c is not one-way, then, there exists an efficient inversion algorithm c^{-1} such that $c^{-1}(k, z) = x$.

Now, assume that $p(x, k) = y$. Since $y = z \oplus k$, the existence of c^{-1} implies the existence of an efficient algorithm computing the key x[1] of a block cipher when a plaintext-ciphertext pair is known (k and y)[2]. However, this is computationally

[1] Notice that the compression function passes the message as the key to the underlying block cipher.

[2] Running $c^{-1}(k, k \oplus y)$ would yield the key x used for producing the ciphertext $y = p(x, k)$ from the plaintext k.

infeasible if the block cipher p is secure. Thus, the assumption that c is efficiently invertible must be false. This proves the proposition.

The reader is referred to [101] for a very detailed security analysis of the compression algorithms.

- Construction from number-theoretic hard problems: Compression functions can be constructed from computationally hard problems such as discrete logarithm or integer factorization problem, too. However, such compression functions are not used in practice due to their very poor performance. Hence, a detailed analysis in this thesis is skipped.

  A simplified example based on discrete logarithm problem: Consider a function c defined as

  $$c(k, x) = u^k \cdot v^x \text{mod } p \tag{4.3}$$

  where p is a large prime number (e.g. 1024 bits), $u, p \in \{1, ..., p\}$ are two random numbers, and $k, x \in \{0, ..., p-1\}$ are two inputs.

  c is a compression function since it takes two $|p|$ bits numbers k and x and outputs an output of $|p|$ bits. Its security is based on the discrete logarithm assumption. The reader is referred to e.g. [102] for further overview.

Iteration algorithms Given a secure compression function, an iteration algorithm is required to convert it into an arbitrary-length hash function. A widely used iteration algorithm is the Merkle-Damgard construction [103, 104].

Assume that a compression function $c(k, x)$ is given. Furthermore, for simplicity, assume that $|k| = |x| = |y| = n$. The Merkle-Damgard construction converts $c(k, x)$ into an arbitrary length hash function h as follows [100]: Let X denote an arbitrary-length message to be hashed. X is divided into t chunks of length of n-bit each, $x_1, x_2, ..., x_t$, with $t = \lceil \frac{|X|}{n} \rceil$. Pad x_t with zeros if necessary. Finally, append a final n-bit chunk x_{t+1} that contains the length of the original message before padding, i.e. $x_{t+1} = |X|$. The hash result $h(X) = h_{t+1}$ is obtained by computing

$$h_i = c(h_{i-1}, x_i) \tag{4.4}$$

where $h_0 = IV$, IV is an Initialization Vector (e.g. a constant composed of n 0s), and $i = 1, ..., t+1$.

Most of the practical hash functions available today, such as MD- and SHA-family hash functions, follow the Merkle-Damgard transformation [7].

Remark: Assume that c is an n-bit collision resistant compression function. Then, an arbitrary length hash function, built from c by following the Merkle-Damgard construction, is also n-bit collision resistant [7].

4.1.3 Construction choices for optimized code size

The core component of any hash function is the compression function. It is the component determining hash function's performance and security. Compression functions can be constructed from hard problems in number theory, general block ciphers (pseudorandom permutations) or dedicated block ciphers designed only for hash functions. Constructions based on number theoretic problems are very slow due to heavy computations required. Hence, they are not considered in the following analysis.

In the remainder of this section, hash functions based on general block ciphers and customized hash functions are analyzed in terms of their security, performance, and code size[1]. However, first of all, some observations on the impacts of block ciphers are provided.

4.1.3.1 Impacts of block ciphers on design

Impacts to security Assume that a secure block cipher (i.e., a strong pseudorandom permutation) is used for constructing a compression function by following a (secure) algorithm like Davies-Meyer. Then, the resulting compression function is at most as secure as the underlying block cipher.

For example, suppose that the block cipher is n-bit secure (i.e., mapping n-bit plaintexts to n-bit ciphertexts using an n-bit key). Then, the compression function is at most n-bit secure against both preimage and second-preimage attacks. However, it is only $n/2$-bit collision secure due to birthday attacks. Hence, for building a compression function with n-bit collision security, a block cipher with a block size of $2n$ bits is required.

The block size of the practical and secure block ciphers available today is mostly up to 128 bits. There is only very few block ciphers supporting larger block sizes. A

[1] Code size is the most important selection criteria.

prominent example is the block cipher Rijndael (AES) [105]. The original Rijndael specification supports key and block of lengths 128, 160, 192, 224, and 256 bits. However, only the block length of 128 bits and the key lengths of 128, 192 or 256 bits are standardized in the AES specification [106]. Thus, for obtaining e.g., an 80-bit collision security, the compression function needs to be built from a block cipher like Rijndael supporting the block length of 160 bits, minimum.

Remark: In case, only a block cipher with smaller block length is available, then a double-length compression function such as MDC-2 and MDC-4 must be used. Given an n-bit block cipher, MDC-2 and MDC-4 produces $2n$-bit outputs, respectively [7].

Impacts to performance The performance of an iterated hash function depends on the number of iterations required for processing the entire input message. In turn, the number of iterations required depends on the compression ratio of the underlying compression function. Finally, the compression ratio depends on two factors: the key and block lengths of the underlying block cipher and the compression algorithm.

- Impact of the key and block lengths: Consider a block cipher with a block length of n bits and a key length of k bits. Furthermore, suppose that this block cipher is converted to a compression function using the Davies-Meyer method. In such a case, the number of iterations required for hashing a message X would be $(|X|/k)+1$. Thus, using a block cipher with $k > n$ results in a better performance than a block cipher with $k \leq n$.

  Consider the same block cipher above again. However, now assume that the compression function is constructed by following the Matyas-Meyer-Oseas method instead of the Davies-Meyer method. Roughly speaking, the Matyas-Meyer-Oseas method is identical to the Davies-Meyer method with a single difference. That is, the inputs play reversed roles. Hence, in such a case, the performance of hashing would mainly depend on the block length n. More specifically, the number of iterations required for hashing a message X would be $(|X|/n) + 1$.

- What is the optimal choice?: If the block cipher available for constructing a compression function operates with a key of length $k < n$, the Matyas-Meyer-Oseas or Miyaguchi-Preneel method lead to a better performance. This is due to the fact that the number of block cipher operations required for hashing a message depends on the block length n. The larger is the block length, the fewer block cipher operations are required. In other cases, i.e. $k \geq n$, the compression function built with the Davies-Meyer method performs better. The reason is that the number of block cipher operations depends on the key size k in such a case.

Most modern block ciphers use variable length keys and sometimes also variable length blocks (e.g., Rijndael). The choice of the block length depends on the security level required. Thus, there is not much flexibility while determining the block length n. However, the keys might be chosen from a large set of possibilities. For example, for $n = 128$ bits, the key length k for Rijndael might be 128, 160, 192, 224, and 256 bits. Due to the observations above, one might think that choosing the largest possible key length results in a hash function with the best performance when the Davies-Meyer mode is used. Unfortunately, this is not fully correct. The reason is that each computation with a block cipher consists of two steps. The first step is referred to as key setup. The second step is the encryption (i.e., actual block cipher operation). The performance of these two steps often depends on the key length.

Both the key setup and encryption need to be executed at each iteration step during hashing a data with an iterated hash function. Thus, the performance improvement gained by reducing the number of iterations with increasing the key length must be larger than the performance penalty to the key setup and encryption operations.

Optimal choice for the Davies-Meyer mode:

Assume that the block cipher supports variable block and key lengths. Let N and K be the set of possible block and key lengths, respectively. Furthermore, suppose that $t_{k,n}^{ksetup}$ and $t_{k,n}^{enc}$ denote the execution time of a key setup and an encryption operation with a key and block of length k and n bits, respectively. Then, the optimal length pair, leading to the best hash performance, is $((\max(k' \in K)$, $\min(n' \in N))$ with

$$\frac{|X| \cdot (t_{k,n}^{enc} + t_{k,n}^{ksetup})}{k} \geq \frac{|X| \cdot (t_{k',n'}^{enc} + t_{k',n'}^{ksetup})}{k'} \tag{4.5}$$

where k denotes the minimum block length for the hash function to be secure and X the message to be hashed. Simplifying this equation yields :

$$\frac{k'}{k} \geq \frac{(t^{enc}_{k',n'} + t^{ksetup}_{k',n'})}{(t^{enc}_{k,n} + t^{ksetup}_{k,n})}. \tag{4.6}$$

Optimal choice for the Matyas-Meyer-Oseas and the Miyaguchi-Preneel modes: In these modes, the inputs play reversed roles compared to the Davies-Meyer mode. Hence, the optimal length pair, leading to the best hash performance, is $((\min(k' \in K), \max(n' \in N))$ with

$$\frac{|X| \cdot (t^{enc}_{k,n} + t^{ksetup}_{k,n})}{n} \geq \frac{|X| \cdot (t^{enc}_{k',n'} + t^{ksetup}_{k',n'})}{n'} \tag{4.7}$$

where n denotes the minimum block length for hash to be secure and X the message to be hashed. Simplifying this equation yields :

$$\frac{n'}{n} \geq \frac{(t^{enc}_{k',n'} + t^{ksetup}_{k',n'})}{(t^{enc}_{k,n} + t^{ksetup}_{k,n})}. \tag{4.8}$$

Remark: A slight modification of Equation 4.6 (or Equation 4.8 depending on the compression mode) can be used for determining the optimal key length for optimal energy efficiency: Assign $t^{ksetup}_{k,n}$ and $t^{enc}_{k,n}$ with the energy consumption of the key setup and encryption operations, respectively.

Impacts to code size Block ciphers are the main building block of many cryptographic primitives such as hash functions, MACs, encryption mechanisms, symmetric key signatures etc. Building all of them from a (single) block cipher significantly reduces the overall code size.

4.1.3.2 Comparisons and design decisions

Customized hash functions and hash functions based on block ciphers are analyzed in terms of their performance, code size and security.

Security The security level provided by a hash function is mainly based on its output length. Customized hash functions consider this fact in their design. Hence, they allow for variable output sizes from a large range. This is necessary to satisfy the security demands of general application scenarios possible in practice. For example, the output size of a SHA-family hash function might be 160, 224, 256, 384, or 512 bits.

Hash functions based on block ciphers are (almost) as flexible as the customized hash functions in terms of this aspect. The length of their outputs is determined by the block length of the underlying block cipher. Currently, block ciphers with a block length of 128, 160, 192, 224, and 256 bits are available. A hash function built from a 160-bit block cipher (e.g. Rijndael) satisfies the mid-term security requirements of a secure hash function at large extent. Even a 512-bit hash function can be constructed using Rijndael in combination with the MDC-2 compression algorithm.

Thus, customized hash functions provide no significant security advantage over the hash functions based on block ciphers in terms of security.

Performance The performance of a hash function depends on the block size of the underlying block cipher and its performance. Customized hash functions are based on dedicated block ciphers which are designed specifically for hashing. Thus, they are especially optimized for providing good performance.

Hash functions based on block ciphers need to be constructed from a generic block cipher. Generic block ciphers are designed specifically for encryption. Thus, they work with relatively small blocks compared to dedicated block ciphers. This results in a hash function that requires larger number of block cipher operations than customized hash functions for hashing large inputs.

Thus, customized hash functions are likely to be superior to hash functions based on block ciphers in terms of performance.

Code size The dedicated block cipher of a customized hash function is integrated into the implementation code. Therefore, it is typically not possible to share that block cipher with other components. Hence, any component, requiring a block cipher for its functionality, needs to implement its own block cipher.

Hash functions based on block ciphers can share a block cipher with other components. This allows for increasing code reuse, hence, helps with reducing the overall code size.

Thus, hash functions based on block ciphers are likely to be superior to the customized hash functions in terms of code size.

Conclusions As discussed in Chapter 2, the most important design criteria for the secure remote programming scenario is the code size. Thus, constructing an iterated hash function in the Davies-Meyer mode, based on a block cipher with 160-bit block and key lengths, meets the security and code size requirements of a secure remote programming mechanism.

4.2 Message authentication codes (MACs)

Definition A message authentication code (MAC) algorithm is a keyed hash function. It takes an arbitrary-length message and a fixed-length key and produces a fixed-length output [7]. The output value is often called MAC-tag or simply tag. Compared to a general hash function, a MAC guarantees also the message origin authentication in addition to the message integrity. That is, given a message-tag pair, the receiver can verify if a received message is exactly the message that has been sent by the other party. A MAC is a secret (symmetric) key algorithm. Thus, all communicating parties must share the same secret key for producing message-tag pairs valid for others, as well as for verifying message-tag pairs received from others. Figure 4.6 illustrates the general MAC principle.

4.2.1 Properties and requirements

A keyed hash function $\mathrm{MAC}_k(\cdot) = \mathrm{MAC}(k, \cdot)$ is a message authentication code if it satisfies the following requirements [7]:

Compression Given a fixed-length key k, MAC_k maps an input of arbitrary-length binary string x into an output of fixed-length binary string $\mathrm{MAC}_k(x)$ [7].

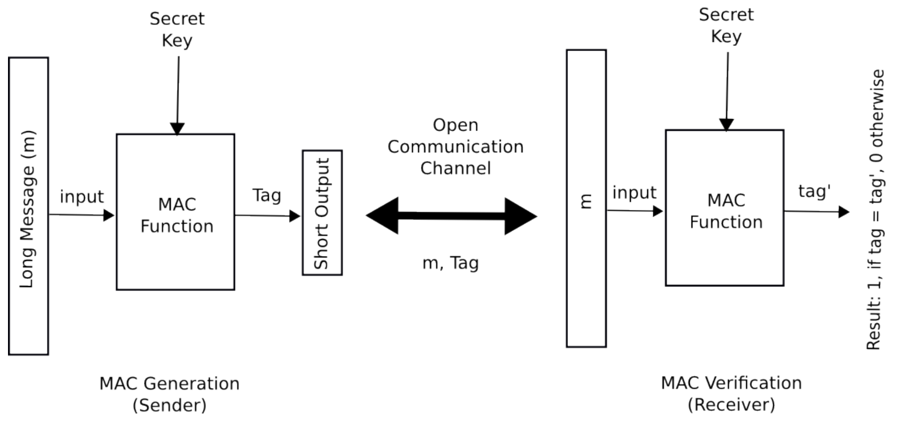


Figure 4.6: General MAC principle - MAC is used for verifying message integrity and authenticity. The sender and the receiver share a secret key. The sender computes the tag for a message (m) using a MAC algorithm and the secret key. This is called MAC generation. The sender sends the message (m) and the MAC tag over an open channel to the receiver. The receiver verifies the authenticity of the massage by computing the tag' of the message under the secret key and by comparing it with the received tag. The message is authentic, if tag = tag'. This is called MAC verification. MAC verification outputs 1 if the message is authentic, otherwise 0.

Ease of computation For every input x and key k, $\text{MAC}_k(x)$ is easy to compute [7].

- Praxis relevance: MACs are used for identifying the origin of messages and files while protecting their integrity. For example, one application scenario is the verification of the source of the large downloads from the Internet. For this, the MAC tag of the files (e.g. software) available for download is computed by their owner and attached to them. Anyone who downloaded a file verifies its origin and content by comparing the MAC tag, which is computed locally, with the tag that was downloaded together with the file. If both tag values are equal, the user is assured that the file has not been modified and it has been generated by the claimed source (i.e., from a party sharing the secret key used for computing the tags.). The size of the files can be several gigabytes large. Hence, an ideal MAC function must be efficient enough to compute the tags very quickly even for very large files.

Computation resistance (Existential unforgeability under an adaptive chosen-message attack) Suppose a MAC algorithm MAC_k, where the key k is fixed and unknown to the adversary. MAC_k is existentially unforgeable under an adaptive chosen-message attack (or simply secure [100]) if the adversary cannot produce any valid message-tag pair (x,$\text{MAC}_k(x)$) even if he/she is given zero or more valid message-tag pairs of his/her choice (x_i,$\text{MAC}_k(x_i)$) with $x \neq x_i$ [7]. Generally speaking, computation resistance implies that the adversary cannot authenticate any message using a MAC when the MAC key is unknown to him/her.

- Praxis relevance: It is often wrongly argued that the definition of the computation resistance above is too strong. In particular, the assumption that the adversary is given some message-tag pairs (of his/her choice) is assumed to be unrealistic in real life. The wrong belief is that the adversary cannot obtain such message-tag pairs, hence, the definition should be relaxed by excluding that misassumption.

  Indeed, obtaining any message-tag pair is quite simple. Consider an application scenario where a MAC algorithm is used for authenticating the messages (e.g. contracts for orders, emails for payments, or even simple email conversations, etc.) exchanged over an open channel between two companies: a supplier and a client. The adversary can obtain any number of valid message-tag pairs by simply eavesdropping the communication channel between those two companies[1].

  The limitation of the example above is that the adversary does not control the message-tag pairs. However, the definition is stronger and allows the adversary to do that. That is, the adversary is given the tags for the messages of his/her choice. So, the question is how realistic is this assumption. Not surprisingly, obtaining such message-tag pairs is also realistic in practice. Consider the supplier-client example again. The adversary can obtain the tag for a message of his/her choice as follows: He/she sends the supplier an email and asks for an offer for some products under the name of a valid customer[2]. By capturing the supplier's response through e.g., a compromised router, the adversary obtains the tag for a message of his/her choice authenticated with the key of the impersonated customer. For

[1] The communication channel does not need to be encrypted for achieving authentication.

[2] In practice, no authentication is required for such emails asking for an offer.

such an attack, the adversary needs to impersonate the email address of a valid customer which is in general not a very challenging attack.

In order to cover all of such (potential) attack surfaces in all possible application scenarios, MACs are designed under such strong assumptions [100].

- Ideal security: Given a fixed key k, a MAC algorithm MAC_k has ideal computation resilience if the most efficient attack for producing a valid message-tag pair takes approximately $min(2^{|k|}, 2^n)$ MAC evaluations where n is the size of the MAC tags [7].

4.2.2 Construction of MACs

As shown in Figure 4.7, there are mainly four classes[1] of MACs: MACs based on pseudorandom permutations (block ciphers), MACs based on hash functions, and finally customized MACs.

Customized MACs The Message Authenticator Algorithm (MAA) and the MDx-MAC are the examples of customized MACs [7].

The MAA was developed by Davies et al.. It was accepted as the banking standard in ISO 8731-2 in 1983. The length of MAC tags in MAA is 32 bits. Hence, the MAA is outdated for the security requirements of today. For this reason, the MAA is not further discussed in this thesis.

The MDx-MACs [107] represent a family of customized hash functions. They are built from a hash function following the construction principle of the MD4 hash function. The MD5, RIPEMD, and SHA-0 are the examples of such hash functions. An MDx-MAC is constructed by incorporating a secret key into the underlying hash function. This is done so that the secret (MAC) key gets involved into all iteration steps performed while computing a hash-value. The MDx-MACs take a key of length up to 128 bits. 128-bit keys provide a level of security which is still sufficient for the security standards of today. However, there are several efficient collision attacks on the earlier versions of MD4 based hash functions [108]. In general, MDx-MACs inherit the security

[1] MACs based on stream ciphers as well as MACs based on universal hash functions, such as UMAC and Poly1305AES MAC, and MACs offering both confidentiality and authentication, such as CWC, EAX and GCM, are out of the scope of this thesis.

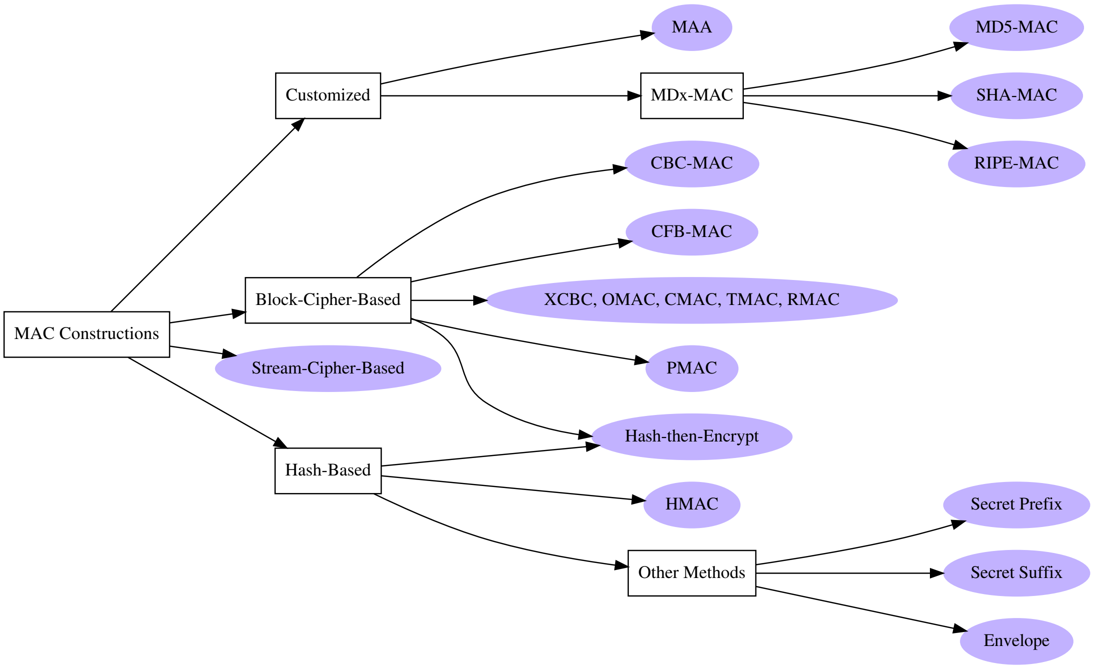


Figure 4.7: Taxonomy of MAC constructions - MACs can be constructed from block ciphers, stream ciphers, hash functions as well as customized algorithms. Due to its high performance and security, HMAC is the most widely used method in practice.

of its underlying hash function. Thus, use of the MDx-MACs are not recommended. For this reason, MDx-MACs are not further discussed in this thesis.

MACs based on block ciphers One of the broadly accepted approach for constructing MACs is the use of block ciphers. The basic idea is very similar to the idea of hash functions based on block ciphers. Both methods are based on applying a block cipher iteratively on the input data. The main differences are basically twofold. Firstly, MACs use a secret key in combination with a block cipher for producing the MAC tags. In case of a hash function, no secret key is required for computing the hash-values. Secondly, hash functions are built on top of a one-way compression function constructed from a block cipher. MACs use a block cipher as it is.

The use of a block cipher in the cipher block chaining (CBC) mode is the most widespread way of constructing MACs [109]. Broadly speaking, encryption of a message with a block cipher using a secret key constitutes a fixed-length MAC. The role of the CBC mode is to convert such a fixed-length MAC into a variable-length MAC. Thus, the cipher block chaining mode can be seen as an iteration algorithm.

In the following, several MAC constructions based on block ciphers are presented.

- CBC-MAC [100]: A MAC constructed from a block cipher using the CBC mode as the iteration algorithm is called CBC-MAC. An n-bit CBC-MAC is defined as follows: Given an n-bit block cipher p (i.e., block size of p is n bits), a secret MAC key k for p, and an arbitrary-length message X. X is divided into t chunks of length n bits each, $x_1, x_2, ..., x_t$, with $t = \lceil \frac{|X|}{n} \rceil$. x_t is padded with zeros if necessary. The n-bit MAC tag of the message, CBC-MAC$_k(X) = t_t$, is obtained by iteratively computing:

$$t_i = p(k_n, t_{i-1} \oplus x_i) \tag{4.9}$$

  where $t_0 = IV$ (a constant composed of n 0s), $k_n = p(k, |X|)$, and $i = 1, ..., t$. Deriving the key k_n from the length of the message $|X|$ is crucial for the security of the CBC-MAC for variable-length messages [100].

- CFB-MAC [101]: A MAC constructed from a block cipher using the CFB mode as the iteration algorithm is referred to as CFB-MAC. Given the same parameters as above, the n-bit MAC tag of the message X, CFB-MAC$_k(X) = t_{t+1}$, is obtained by iteratively computing:

$$t_{t+1} = p(k_n, t_t) \tag{4.10}$$

where

$$t_i = p(k_n, t_{i-1}) \oplus x_i \tag{4.11}$$

and $t_0 = IV$ (a constant composed of n 0s), $k_n = p(k, |X|)$, and $i = 1, ..., t$.

The CFB-MAC requires one additional block cipher operation before finalizing the output (see Equation 4.10). Thus, the CBC-MAC is superior to the CFB-MAC. For this reason, the latter is not discussed further in this thesis.

- Other constructions: The CBC-MAC is not secure for variable-length messages if k is used as the MAC key instead of k_n in the above construction. Several mechanisms have been proposed to mitigate this security flaw. The XCBC-MAC [110] is one of them. The OMAC [111] is another proposal which improves the XCBC-MAC by reducing its key size. Finally, the CMAC, a variation of the OMAC, is a MAC algorithm recommended by NIST. Deriving the key k_n from k with $k_n = p(k, |X|)$ is referred to as input-length key separation [112]. This method ensures that messages of different lengths are authenticated with different keys. Thus, this method can be used for obtaining a secure CBC-MAC for variable-length messages. Due to its simplicity, this thesis prefers this approach over other methods like CMAC.

  All MACs presented so far are sequential. That is, each iteration step t_i depends on the previous step t_{i-1}. Thus, they cannot take advantage of the parallel computing offered by most of the modern CPUs. In order to address this bottleneck, the PMAC (Parallelizable MAC) [113] was proposed. By allowing the computation of the t_is in parallel, the PMAC results in a much faster MAC implementation on modern platforms. However, sensor platforms are equipped with rather slow and small CPUs. Hence, the PMAC is not considered as an option in this thesis.

MACs based on hash functions MACs can also be constructed from hash functions. As described in the previous subsection, a hash function takes a single input of arbitrary length and produces an output of fixed length. However, a MAC function takes two inputs, a message and a secret key, and produces an output, called tag.

Hence, a method is required to convert a single input hash function into a two inputs MAC function.

Including a secret MAC key as part of the message being hashed is a way of converting a hash function into a MAC function [114]. However, simply appending or prepending a key to the message results in insecure MAC constructions. Hence, methods meeting certain security requirements are required. Such a secure method is the hash-then-encrypt method [115]. In this method, the MAC tag for a message is simply computed by encrypting the hash-value of that message with a secret key.

Basically, MACs based on hash functions can be grouped into two classes: the hash-then-encrypt method and the methods including a key into the message securely. In the following, several examples from each class are presented.

- Hash-then-Encrypt construction [115]: Given a general hash function h, an n-bit block cipher p, a secret (MAC) key k, and an arbitrary-length message X, the n-bit tag of the message is computed as:

  $$\mathrm{MAC}_k(X) = p(k, h(X)). \tag{4.12}$$

  This construction is secure if the output size of the hash function equals the block size of p.

- HMAC (Hash-based MAC) [116]: Given an n-bit general hash function h, a secret (MAC) key k, and an arbitrary-length message X, the n-bit tag of the message is computed as:

  $$\mathrm{HMAC}_k(X) = h((k \oplus \mathit{opad})||h((k \oplus \mathit{ipad})||X)) \tag{4.13}$$

  where $\mathit{opad} = 0x36...36$ and $\mathit{ipad} = 0x5c...5c$ are the constants of length of the block size of the underlying compression function used in h.

  HMAC is a FIPS standard and is used widely in practice due to its efficiency and proven security.

  Remark: NMAC is a hash-based construction which is very similar to HMAC. However, NMAC requires modifications on the underlying hash function and uses two secret keys [100]. Hence, it is not very practical to be used in practice. For this reason, its description is skipped in this thesis.

- Other constructions [107]: There are some other methods which propose including the secret key into the messages to be hashed. These are secret prefix, secret suffix, and envelope methods. The first method proposes to add the secret key as a prefix to the message before hashing, i.e.,

$$\mathrm{MAC}_k^{\mathrm{sprefix}}(X) = h(k||X). \tag{4.14}$$

The secret suffix method uses the key as a suffix of the message being hashed, i.e.,

$$\mathrm{MAC}_k^{\mathrm{ssuffix}}(X) = h(X||k). \tag{4.15}$$

Finally, roughly speaking, the envelope method proposes to add the key as a prefix and a suffix to the message before hashing, i.e.,

$$\mathrm{MAC}_k^{\mathrm{envelope}}(X) = h(k||X||k). \tag{4.16}$$

These constructions are very efficient. However, they suffer from several security weaknesses [107]. Hence, secure constructions (e.g., HMAC or CBC-MAC) must be used instead of those methods.

4.2.3 Construction choices for optimized code size

In the previous subsection, several ways of constructing MAC functions are presented. However, only a few of them are suitable for practical use due to performance or security reasons (see previous section for details). Those are the hash-then-encrypt method, the CBC-MAC, and the HMAC. The underlying component for all of them is either a hash function or a block cipher. However, since hash functions can be constructed from a block cipher, all of those methods can be implemented using a block cipher, too. Section 4.1.3 proposes to build hash functions from a block cipher. Thus, it is assumed that a block cipher is already available on the implementation platform. For keeping the code size at minimum, this thesis proposes to construct the MAC functions from block ciphers as well. Overall code size is optimized as a result of the increased code share.

In the following, the hash-then-encrypt method, the CBC-MAC method and the HMAC method are analyzed in terms of their code size, performance and security for determining the best option for a secure remote programming scenario. Before proceeding with the analysis, some observations on the impact of block ciphers are presented.

4.2.3.1 Impacts of block ciphers on design

Impacts to security The main security requirement for all MAC functions is the computation resistance. Informally speaking, the computation resistance assures that the adversary cannot produce a valid tag for any message if the secret MAC key is not known to him/her. There are basically two factors affecting the computation resistance of a MAC function: the underlying block cipher and the chaining mode converting a fixed-length MAC into a variable-length MAC. If an underlying block cipher is not secure, then, any variable-length MAC built from it is also not secure whatever secure chaining mode is used. If the underlying block cipher is secure, then the variable-length MAC built from it is also secure when the used chaining mode is secure. For simplicity, let's ignore the chaining mode by assuming it as a secure mode. Furthermore, let's consider a very simple MAC construction for studying the impact of a block cipher on its security. Let the MAC function be defined as

$$\mathrm{MAC}_k(X) = p(k, X) \tag{4.17}$$

where p is an n-bit (ideal) block cipher, k is an m-bit secret key, and X is an n-bit message.

Assume that a message-tag pair (tag_x, X) is given to the adversary. The key is unknown to the adversary as it is secret. The adversary has basically two possibilities for producing a valid tag for any message. The fist one is so-called key-recovery attack. That is, the adversary tries all possible keys $k_i = 0, ..., 2^m - 1$ to find a key k_j such that $tag_x = \mathrm{MAC}_{k_j}(X)$ holds. In such a case, the probability is very high[1] that $k_j = k$. Once k is recovered, the adversary can obviously compute a valid tag for every message (total break).

The second approach the adversary can follow is as follows: The adversary chooses a key $k_i \in \{0, ..., 2^m - 1\}$, randomly, and set it as the MAC key. Then, he/she tries all possible messages $x_i = 0, ..., 2^n - 1$ to find a message x_j such that $tag_x = \mathrm{MAC}_{k_i}(x_j)$

[1] It is not 1. However, the probability $Pr[k_j \neq k]$ is negligible in practice if p is a secure block cipher.

holds. By finding such a message, the adversary breaks the computation security of the MAC function.

For a secure block cipher p and a randomly chosen key k, the success probabilities of these two attacks are no better than 2^{-m} and 2^{-n}, respectively. Any attack, which is more efficient than those two in this setting, must exploit some vulnerability of the block cipher.

A MAC construction is at most as secure as the underlying block cipher. The security of a block cipher depends on its key and block length. Hence, for building a MAC function with n-bit computation security, a block cipher with a key and block length of n bits is required. A $2n$-bit block cipher must be chosen to have an n-bit security against collision attacks.

Impacts to performance Assume an n-bit block cipher is used for building a MAC function. That is, the block cipher takes an n-bit input and produces an n-bit output. Furthermore, for simplicity, suppose that the message X, for which that MAC tag to be computed, is exactly $(t \cdot n)$-bit long and the length of the keys for the block cipher and MAC are n bits. The performance of the hash-then-encrypt method, the CBC-MAC method and the HMAC method are analyzed under these assumptions:

- CBC-MAC: Computing the tag for the message requires $(|X|/n) = t$ iterations. In addition, one block cipher operation is required for deriving the key k_n from k to obtain different keys for messages of different lengths. Thus, CBC-MAC requires $t + 1$ block cipher operations to compute the tag of X.

- Hash-then-encrypt: Assume that the hash function required in this method is built from the block cipher by following the Merkle-Damgard construction with the Davies-Meyer compression mode. Then, $t + 1$ block cipher operations are required for computing the hash-value. One additional block cipher operation is required for encrypting the hash-value to obtain the tag. Thus, the total performance overhead is $t + 2$ block cipher operations.

- HMAC: Two hash computations need to be performed in this method: inner and outer. The inner hash operation maps an $((t + 1) \cdot n)$-bit message to an n-bit output, requiring $t+2$ block cipher operations in total. The outer hash operation takes a $2n$-bit input and produces an n-bit output, the tag. This requires 2 additional block cipher operations. Thus, the total performance overhead of the HMAC is $t + 4$ block cipher operations.

Remark on the impact of the key and block lengths: The performance of an iterated hash function, depending on the compression mode, depends on the key and block lengths of the underlying block cipher (see Section 4.1.3.1). For example, in case of the Matyas-Meyer-Oseas mode, large block lengths improve the performance. The same holds for the MACs. However, the performance penalty of the key setup is much smaller. The reason is that the same key is used in all iteration steps during a MAC computation. Hence, the key setup needs to be executed only once[1] at the beginning. Then, all subsequents block cipher computations use that key.

Thus, Equation 4.8, given for choosing the optimal block length, needs to be slightly modified for the MACs. Let's assume the same notation used in Equation 4.8. Then, the optimal length pair, leading to the best MAC performance, is $((\min(k' \in K), \max(n' \in N))$ with

$$\frac{|X| \cdot t^{enc}_{k,n}}{n} + t^{ksetup}_{k,n} \geq \frac{|X| \cdot t^{enc}_{k',n'}}{n'} + t^{ksetup}_{k',n'} \tag{4.18}$$

where k and n denote the minimum key and block lengths for MAC to be secure and X the message for which the tag is being computed. The key setup is executed only once. Thus, by assuming its performance penalty on the overall execution time as negligible, the following approximation is obtained:

$$\frac{n'}{n} \geq \frac{t^{enc}_{k',n'}}{t^{enc}_{k,n}}. \tag{4.19}$$

Impacts to code size For reducing the code size of a MAC, it can be implemented together with the hash function in a single method. Algorithms 1 and 2 and 3 provide a pseudocode for such an implementation for the hash-then-encrypt method, the CBC-MAC method, as well as the HMAC method. The implementation of the hash function follows the Merkle-Darmgard principle with the Davies-Meyer mode. The method, providing both hash and MAC operations, is named HashAndMAC.

All MAC methods require adding only a few additional if branches into the implementation of the hash function to provide the corresponding MAC functionality. For

[1] In case of HMAC, two key setup operations are required due to two (inner and outer) hash operations required.

example, given a block cipher, the hash-then-encrypt method can be implemented by adding only a single *if* branch into the implementation of the hash function. Other methods require slightly more complex additions to the implementation. Thus, the hash-then-encrypt method seems to be the most efficient choice when the code size is considered. It is followed by the HMAC and the CBC-MAC methods.

Algorithm 1 Co-implementation of a block-cipher-based hash function with the hash-then-encrypt MAC

```
Require: an $n$-bit block cipher $p(\cdot,\cdot)$, an $n$-bit key $k$, a message $X = x_1...x_t$ with
    $t = \lceil \frac{|X|}{n} \rceil$
Ensure: $n$-bit MAC tag of $X$ if opMode= 1, $n$-bit hash-value of $X$ otherwise
 1: function HashAndMAC($X, k, opMode$)
 2: $h_0 \leftarrow 0$, $x_{t+1} \leftarrow |X|$
 3: for $i$ from 1 by 1 to $t+1$ do
 4:     $h_i \leftarrow p(x_i, h_{i-1}) \oplus h_{i-1}$
 5: end for
 6: $output \leftarrow h_{t+1}$
 7: if $opMode == 1$ then
 8:     {/* Operation Mode is MAC Function: output is MAC tag */}
 9:     $output \leftarrow p(k, h_{t+1})$
10: end if
11: return $output$
12: {/* Notes: */}
13: {Step 3: Merkle-Damgard iteration (see Equation 4.4)}
14: {Step 4: Davies-Meyer compression function (see Equation 4.2)}
15: {Step 9: Hash-then-Encrypt method (see Equation 4.12)}
```

4.2.3.2 Comparisons and design decisions

Depending on their performance, code size and security, the most appropriate MAC construction for the secure remote programming scenario is determined.

Security The security levels provided by all methods are the same from the theoretical point of view. However, the security of the HMAC is proven by its widespread use in practice for years. Thus, the HMAC is likely to be the best choice in terms of security.

Algorithm 2 Co-implementation of a block-cipher-based hash function with the CBC-MAC

Require: an n-bit block cipher $p(\cdot,\cdot)$, an n-bit key k, a message $X = x_1 ... x_t$ with $t = \lceil \frac{|X|}{n} \rceil$

Ensure: n-bit MAC tag of X if opMode= 1, n-bit hash-value of X otherwise

```
 1: function HashAndMAC($X, k, opMode$)
 2: $h_0 \leftarrow 0$
 3: if $opMode == 1$ then
 4:    $k_n \leftarrow p(k, |X|)$
 5: end if
 6: for $i$ from 1 by 1 to $t$ do
 7:    if $opMode == 0$ then
 8:       $h_i \leftarrow p(x_i, h_{i-1}) \oplus h_{i-1}$
 9:    end if
10:    $h_i \leftarrow p(k_n, h_{i-1} \oplus x_i)$
11: end for
12: $output \leftarrow h_t$
13: if $opMode == 0$ then
14:    {/* Operation Mode is Hash Function: output is hash-value */}
15:    $x_{t+1} \leftarrow |X|$
16:    $h_{t+1} \leftarrow p(x_{t+1}, h_t) \oplus h_t$
17:    $output \leftarrow h_{t+1}$
18: end if
19: return $output$
20: {/* Notes: */}
21: {Step 4: Input-length key separation required for securing CBC-MAC for variable-
    length inputs}
22: {Step 10: CBC-MAC (see Equation 4.9)}
23: {Step 6,15: Merkle-Damgard iteration (see Equation 4.4)}
24: {Step 8,16: Davies-Meyer compression function (see Equation 4.2)}
```

Algorithm 3 Co-implementation of a block-cipher-based hash function with the HMAC

Require: an n-bit block cipher $p(\cdot,\cdot)$, an n-bit key k, a message $X = x_1...x_t$ with $t = \lceil \frac{|X|}{n} \rceil$

Ensure: n-bit MAC tag of X if opMode= 1, n-bit hash-value of X otherwise

```
 1: function HashAndMAC($X, k, opMode$)
 2: $h_0 \leftarrow 0$, $x_{t+1} \leftarrow |X|$
 3: if $opMode == 1$ then
 4:     $ipad = 0x5c...$
 5:     $opad = 0x36...$
 6:     $k_{inner} \leftarrow p(k \oplus ipad, h_0) \oplus h_0$
 7:     $k_{outer} \leftarrow p(k \oplus opad, h_0) \oplus h_0$
 8:     $h_0 \leftarrow k_{inner}$
 9: end if
10: for $i$ from 1 by 1 to $t + 1$ do
11:     $h_i \leftarrow p(x_i, h_{i-1}) \oplus h_{i-1}$
12: end for
13: $output \leftarrow h_{t+1}$
14: if $opMode == 1$ then
15:     {/* Operation Mode is MAC Function: output is MAC tag */}
16:     $h_{t+2} \leftarrow p(h_{t+1}, k_{outer}) \oplus k_{outer}$
17:     $output \leftarrow h_{t+2}$
18: end if
19: return $output$
20: {/* Notes: */}
21: {Step 10: Merkle-Damgard iteration (see Equation 4.4)}
22: {Step 11: Davies-Meyer compression function (see Equation 4.2)}
23: {Step 13: Result of the inner hash operation with $IV_{inner} = k_{inner}$ (see Equation 4.13)}
24: {Step 16: Result of the outer hash operation with $IV_{outer} = k_{outer}$ (see Equation 4.13)}
```

Performance The CBC-MAC requires $t + 1$ block cipher operations for computing the tag of a message X of length $t \cdot n$ bits, where n is the block size of the underlying block cipher. The hash-then-encrypt and the HMAC methods require $t + 2$ and $t + 4$ block cipher operations, respectively. Hence, the CBC-MAC is superior to others in terms of performance.

Code size As shown in Algorithms 1 and 2 and 3, the hash-then-encrypt method is likely to be superior to other methods in terms of the code size. It is followed by the HMAC and the CBC-MAC methods.

Conclusions When constructed from a generic block cipher such as Rijndael-160, all MAC methods considered are almost equally good in terms of their code size and performance. However, they are not equally secure. For example, the security of the hash-then-encrypt method depends on the collision-resistance of the underlying hash-function. The HMAC remains secure even if the collision-resistance of the underlying hash function is broken, since its security depends on the randomness property of the underlying hash function. Moreover, the HMAC is a standard solution that has been in use for many years. Therefore, this thesis proposes to use the HMAC method for computing message authentication codes whenever required.

4.3 Digital signatures

Definition A digital signature is an asymmetric algorithm which takes a fixed-length secret key and a fixed-length message and produces a fixed-length output. The output is often called signature. The signature is verified with a public key. Hence, digital signatures are the public-key equivalent of MACs [100]. They are used to ensure both authenticity and integrity of messages. That is, when a message-signature pair is received, the receiver can verify its origin and content. Another very important property provided by digital signatures is non-repudiation of origin. That is, a signer cannot later deny his/her own signatures which were signed with a secret key whose corresponding non-secret part is known to public.

Digital signatures are generated and verified using separate keys and algorithms. The key, used for signing a message, is called private key (sk). It is secret and known only to the signer. The key, used for verifying a signature, is called public key (pk). It is public and known to everybody. The algorithm, used for generating a signature, is referred to as signing algorithm (Sig). The algorithm, used for verifying a signature, is referred to as verification algorithm (Vrfy). Figure 4.8 illustrates the general application principle of digital signatures.

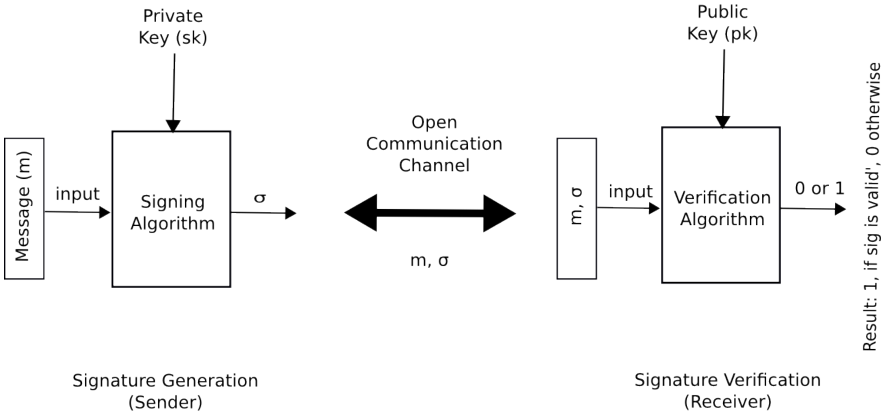

Private Key (sk)
Message (m)
input
Signing Algorithm
σ
Open Communication Channel
m, σ
Public Key (pk)
m, σ
input
Verification Algorithm
0 or 1
Result: 1, if sig is valid', 0 otherwise
Signature Generation (Sender)
Signature Verification (Receiver)

Figure 4.8: General digital signature principle - Digital signatures are used for verifying integrity and authenticity of messages. The sender generates a public-private key pair pk, sk. The public key pk is shared with the potential communication partners. The private key sk is kept secret. The sender computes the signature (σ) of a message (m) using the secret key and a Signing algorithm. This process is called signature generation. The sender sends the message (m) and the signature (σ) over an open channel to the receiver. The receiver checks the validity of the received signature using the public key, the message, and a Verification algorithm. A valid signature ensures that the message has not been altered and originates by the claimed sender. The verification algorithms outputs 1 if the signature is valid, otherwise 0.

4.3.1 Properties and requirements

Assume a signature scheme, $\mathrm{S} = (\mathrm{Gen}, \mathrm{Sig}, \mathrm{Vrfy})$, consisting of three algorithms, where Gen is the key generation algorithm, Sig is the signing algorithm, and Vrfy is the verification algorithm. S should satisfy the following requirements:

Ease of computation The algorithms $\mathrm{Gen}, \mathrm{Sig}, \mathrm{Vrfy}$ should be easy to compute. That is, for every public-private key pair (pk, sk) generated by Gen and every message x of typically fixed (and short) lengths, the signature generation, $\sigma = \mathrm{Sig}_{sk}(x) =$

$\mathrm{Sig}(sk, x)$, and verification, $\mathrm{Vrfy}_{pk}(x, \sigma) = \mathrm{Vrfy}(pk, x, \sigma)$, should be computable, efficiently.

Remark: Signature schemes are often used in a combination with a hash function h. That is, instead of signing a message (e.g., x), its hash-value is signed (i.e., $\mathrm{Sig}_{sk}(h(x))$). Similarly, the verification is performed over the hash-value of the message. This is referred to as hash-and-sign paradigm [100]. By following this approach, the length of data to be signed is significantly reduced. Hence, signing/verifying even a very large message requires to perform the signing/verification algorithm only once. This, not only minimizes the performance penalty of an inefficient signature scheme, but also provides more secure signatures.

Computation resistance (Existential unforgeability under an adaptive chosen-message attack) The computation resistance property of signatures is similar to that for MACs. Informally speaking, a signature scheme is existentially unforgeable under an adaptive chosen-message attack (or simply secure [100]) if the adversary cannot produce any valid message-signature pair (x,σ) even if he/she is given zero or more valid message-signature pairs of his/her choice (x_i,σ_i) with $x \neq x_i$ [100]. This means, the adversary cannot produce a valid signature of any message when the private signature key is unknown to him/her.

- Praxis relevance: The motivation behind this (strong) security notion is similar to that for MACs. The assumption, that the adversary can obtain signatures for the messages of his/her choice, seems to be too strong to be realistic in practice. Some examples are given for justifying a similar assumption for MACs in Section 4.2.1. Those attacks, or slight variations of them, hold for signature schemes. More importantly, signature schemes are designed to be secure against such strong assumptions in order to cover all (potential) attack surfaces in all possible application scenarios.

4.3.2 Construction of digital signatures and design decisions

As shown in Figure 4.9, digital signatures can be built from pseudorandom permutations (block ciphers) or trapdoor one-way functions.

Most practical digital signatures, such as RSA and DSA, are built from trapdoor one-way functions. Trapdoor one-way functions are based on computationally hard problems. Hence, such signatures are designed for operating with very large numbers to provide sufficient level of security. This leads to implementations with large memory footprint and poor performance. This makes them an improper choice for resource restricted platforms such as sensor nodes.

One-time signatures are based on cryptographic primitives with symmetric property, such as block ciphers or hash functions. Hence, they lead to implementations which are very efficient not only in terms of memory footprint, but also performance. The optimization is mainly obtained by sharing their main building blocks with other components on the platform. Their main limitation is that one-time signatures are only for one-time use. That is, for a public-private key pair, only one message can be signed and verified. If more than one message is signed with the same key pair, signatures can be forged. For signing each new message, a new key pair must be generated and used. Therefore, one-time signatures are not very practical for typical application scenarios.

However, a remote programming protocol has different requirements and characteristics. For example, software updates are typically disseminated in a low frequency and sequential order. This observation has motivated the authors of [3] to construct a T-time signature from the Merkle's one-time signature with a simple key management. It is described in the following section.

4.4 A $\mathcal{T}$-time signature for remote programming mechanisms

This section describes a signature mechanism proposed to be used on resource limited sensor platforms [3].

Problem statement and motivation Due to the limited hardware resources available on the sensor nodes, the implementation of the conventional public-key cryptography primitives is a challenging task. For instance, even the efficient implementation of an elliptic curve digital signature scheme occupies 28.1% (13.5KB) of the available program memory (ROM) [69]. Even more efficient signature schemes such as RSA, NTRUSign, and XTR-DSA still introduce 7.4KB, 11.3KB, and 24.3KB ROM overhead

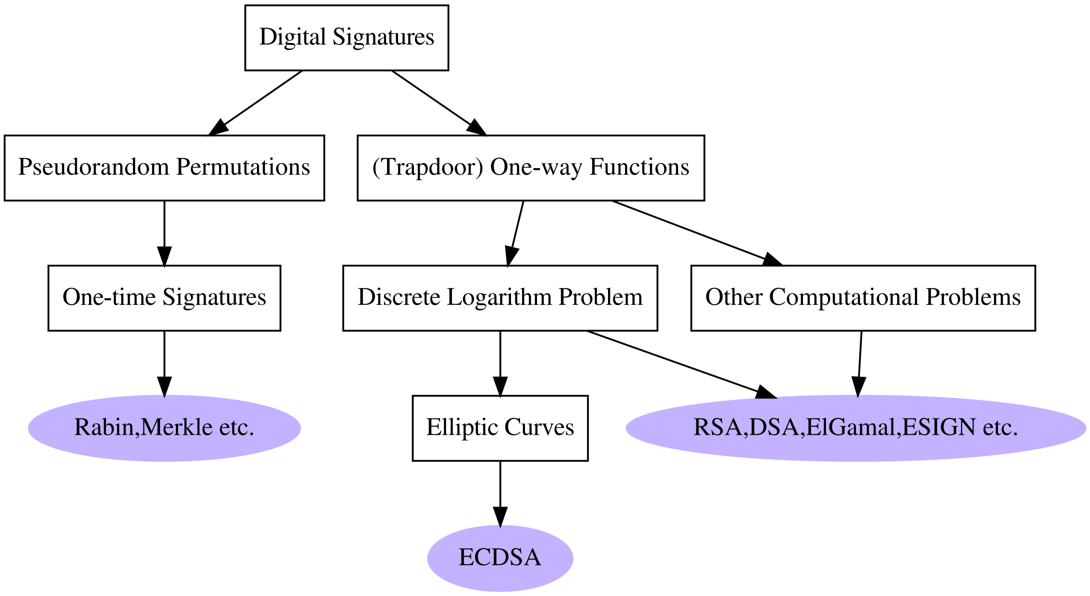

Digital Signatures
Pseudorandom Permutations
(Trapdoor) One-way Functions
One-time Signatures
Discrete Logarithm Problem
Other Computational Problems
Rabin,Merkle etc.
Elliptic Curves
RSA,DSA,ElGamal,ESIGN etc.
ECDSA

Figure 4.9: Taxonomy of signatures in terms of their building blocks - One-time signatures can be built from pseudorandom permutations (block ciphers). Public key signatures are based on (trapdoor) one-way functions. These are based on hard problems from number theory. One-time signatures allow for code size optimized implementations by sharing block ciphers with other primitives like hash functions and MACs.

[70]. Such large memory footprints prohibit a shared implementation of application and software update mechanism without cutting back on functionality or security.

This thesis describes an efficient signature mechanism with a very small memory footprint that addresses this problem. It is motivated by characteristics of remote program mechanisms. Ideally, the signed messages have a strictly sequential order, appear in a low frequency and are simultaneously verified. The fixed order of signatures to be verified allows the verifier to keep a state based on past signatures.

The signature is the extension of Merkle's one-time signature to a $\mathcal{T}$-time signature with a simple state (key) management. The main component of the $\mathcal{T}$-time signature scheme is a one-way compression function providing two major advantages:

- Low program memory usage: it uses only a one-way function and a hash function instead of arithmetic operations based on finite fields or elliptic curves with large memory footprints.
- Low computational cost: the verification and generation of a signature providing n-bit security requires only one hash computation and fewer than $l = n + \lfloor log(n) \rfloor + 1$ computations with a one-way function.

4.4.1 Preliminaries

Let $\mathcal{H}$ be the family of (iterated) functions, constructed as described in Equation 4.4, mapping arbitrary-length binary strings into fixed-length binary strings. An algorithm for building such a hash function is given in Algorithm 2. Furthermore, let $\mathcal{F}$ be the family of fixed-length hash functions, constructed as described in Equation 4.2, mapping fixed-length binary strings into binary strings of the same length. A function $f \in \mathcal{F}$ is one-way (see Proposition 4.1.1). That is, it is easy to compute, but difficult to invert. A hash function $h \in \mathcal{H}$ is meant to be at least second pre-image resistant.

Definition 1 (*Stateful-verifier* signature scheme) *A* stateful-verifier *signature scheme is a signature scheme where the signer and the verifier maintain states. By maintaining a state, it is meant that the signer and the verifier update their states after every signature generation and verification, respectively. For a signature, being* $\mathcal{T}$-time *expresses that the signer can only generate* $\mathcal{T} \in \mathbb{N}$ *signatures with this scheme.*

More precisely, a stateful-verifier $\mathcal{T}$*-time signature scheme is specified by three probabilistic polynomial time algorithms* (Gen, Sig, Vrfy).

- Gen *is the key-generation algorithm. On input of a security parameter n and $\mathcal{T} \in \mathbb{N}$ it outputs a quadruple (pk, sk, st_1^s, st_1^r). These parameters represent the public-key pk, the private key sk, the initial state for the signer st_1^s, and the (public) initial state for the verifier st_1^r, respectively;*

- Sig *is the signing algorithm. On input of a private key sk, a state st_i^s, and a message $m \in \{0,1\}^*$ it outputs a signature σ_m and a new state st_{i+1}^s;*

- Vrfy *is the verification algorithm. On input of a public key pk, a message m, a state st_i^r and a signature σ_m, it outputs a new state st_{i+1}^r and a bit $b = 1$ meaning* valid *and $b = 0$ otherwise.*

The state st_i^s at the signer side specifies the private key to be used to generate the ith signature σ_m. The state st_i^r at the verifier side specifies the public key which should be used to verify the signature σ_m.

Remark: A stateful-verifier signature scheme is not as flexible as a standard signature scheme. However, it provides all properties needed for an application such as remote programming in WSNs. Having a state restricts the verifier to only be able to verify a specific signature after all previous signatures of the same signer have been verified and the state was updated accordingly. While this in general makes the use of such a signature scheme difficult, this does not restrict the use of this signature for secure remote programming in WSNs due to the following characteristics of this application:

- the messages (i.e. software updates) are issued sequentially with a low frequency and every verifier needs to receive the message;

- the set of verifiers (i.e., sensor nodes) remains the same for all messages.

Merkle's one-time signature scheme [71] It works as follows: On input of a security parameter n, the signer selects $l = n + log(n)$ random secret strings k_i each of size n-bit and a one-way function $f \in \mathcal{F}$. The public and private keys are $pk = (u_1, \ldots, u_l)$ and $sk = (k_1, \ldots, k_l)$, respectively, where $u_i = f(k_i)$. To sign a n-bit message x in binary representation, the signer first appends to x the binary string (checksum) representing the number of zeros in x. Let x' be the l-bit resulting string. The signature is $\mathrm{Sig}(x) = (s_1, \ldots, s_v)$, where $s_i = k_i$ when $x'_i = 1$ (i.e., ith bit of x'

is 1), for all $1 \leq i \leq l$. Upon receiving the message x and its signature $(s_1, \ldots, s_v)$, the verifier first computes the string x' as in the signature generation. The signature is *valid*, iff for each ith bit of x', which is 1, the signature contains a value s_i such that $u_i = f(s_i)$.

The main restriction of this signature scheme is that it is secure as long as it is used to sign at most one message without refreshing the public key. Merkle proposed the use of an authentication (binary) tree in a combination with one-time signatures to provide $\mathcal{T}$-time signatures [71]. Motivated by Merkle's solution, several works have been done e.g., for reducing the signature and public key size and for improving the efficiency of the signature verification [117, 118, 119]. However, they impose conflicting requirements to be used on resource-constrained devices: They improve e.g., the verification time, while increasing the signature size and the storage requirements or vice versa.

4.4.2 Proposed signature scheme

The following stateful-verifier $\mathcal{T}$-time signature scheme to be used in resource restricted environments is proposed:

Definition 2 *The* $\mathcal{T}$-time stateful-verifier signature scheme, (Gen, Sig, Vrfy)*, is defined as follows;*

- Gen*: On input of a security parameter n and $\mathcal{T} \in \mathbb{N}$, proceed as follows:*

  1. *Compute l-random secret strings $k_i \leftarrow \{0,1\}^n$, where $l = n + \lfloor log(n) \rfloor + 1$;*
  2. *Compute one-way chains $u_i = f^{\mathcal{T}}(k_i)$ for all $1 \leq i \leq l$, where $f^{\mathcal{T}}(k)$ denotes the $\mathcal{T}$-fold composition of $f(k)$ with itself;*
  3. *Output the public key $pk = (u_1, \ldots, u_l)$, the private key $sk = (k_1, \ldots, k_l)$, and the initial state for the signer $st^s_{1,j} = (\mathcal{T}-1)$ and for the verifier $st^r_1 = pk$ for all $1 \leq j \leq l$.*

- Sig*: On input of the private key sk, the state st^s_i as above, and a message $x \in \{0,1\}^*$, proceed as follows: if $st^s_{i,j} \geq 0$ for all $1 \leq j \leq l$ then do the following. Otherwise stop (i.e, the private key sk has expired).*

  1. *Compute the n-bit hash of the message $h(x)$;*
  2. *Compute the checksum as a binary string φ, representing the number of zeros in $h(x)$'s binary representation;*

3. *Compute the signature* $\sigma_x = (s_1, \ldots, s_v)$ *such that* $s_q = f^{st_{i,q}}(k_q)$ *if the qth bit of* $h(x)||\varphi$ *is 1 for all* $1 \leq q \leq l$ *and set the new state* $st^s_{i+1,q} = st^s_{i,q} - 1$*;*
4. *Output the signature* σ_x*.*

- Vrfy*: On input of the public key pk as above, a message* $x \in \{0,1\}^*$*, and a signature* $\sigma_x = (s_1, \ldots, s_v)$ *proceed as follows:*

  1. *Compute the n-bit hash of the message* $h(x)$*;*
  2. *Compute the checksum* φ*, representing the number of zeros in* $h(x)$*'s binary representation;*
  3. *Do the following, iff for every ith bit of* $h(x)||\varphi$*, which is 1, the signature contains a value* s_i *such that* $v_i = f^q(s_i)$ *for some* $q < \mathfrak{T}$*. Otherwise, output 0;*
  4. *Update the state to* st^r_{i+1} *by replacing* v_i *with* s_i *for all* $v_i = f^q(s_i)$*;*
  5. *Set the public key to* $pk = st^r_{i+1}$ *and output 1.*

Example 3 *Let us assume that Bob has a message x that Alice agrees to sign. Let* $sk = (k_1, \ldots, k_7)$ *be Alice's private key,* $pk = (u_1, \ldots, u_7)$ *the public key, and* $\mathfrak{T} = 3$*. Alice signs the message x with* $h(x) = 1011$ *as follows*[1]*. She computes the 3-bit (i.e.,* $\lfloor log(4) \rfloor + 1$*) checksum* φ *representing the number of 0's in* $h(x)$*. That is* 001*. Then, she appends it to* $h(x)$*. The resulting string is* $h'(x) = h(x)||\varphi = 1011001$*. Since Alice's initial state is* $st^s_1 = (2, 2, 2, 2, 2, 2, 2)$*, the signature of x is* $\sigma_x = (s_1, s_3, s_4, s_7) = (f^2(k_1), f^2(k_3), f^2(k_4), f^2(k_7))$*. Alice sends the tuple* (x, σ_x) *to Bob and updates her state to* $st^s_2 = (1, 2, 1, 1, 2, 2, 1)$*.*

On receipt of the message x and its signature, Bob first computes $h'(x) = h(x)||\varphi = 1011001$*. He accepts the message as authentic, since for each ith bit of* $h'(x)$ *that is 1, the signature* σ_x *contains a parameter* s_i *such that* $f(f^2(k_i)) = f(s_i) = u_i$*. Bob changes his public key (respectively his new state* st^r_2*) to* $(s_1, u_2, s_3, s_4, u_5, u_6, s_7)$*. In case of an invalid signature message, Bob drops it and keeps his old state without changing it.*

4.4.3 Security evaluation

Adversary model In Chapter 3, adversary models for remote programming mechanisms are described. Now, for evaluating the security of the proposed signature scheme, two additional classes of adversaries are considered: *weak* and *strong* adversaries. A

[1] Assume that x is the first message which Alice signs for Bob.

weak adversary can interfere with the communication, e.g., to catch, drop, modify, resend and insert new packets. Moreover, assume that he/she can partition any part of the network by blocking the communication between honest sensor nodes e.g., by running a selective jamming attack. During this period, he/she can perform any operation on the sensor nodes. However, the duration of the network-partition is bounded. That is, it cannot be expanded over *consecutive* software update processes and signing operations. By a strong adversary, it is meant a weak adversary with two additional capabilities. Firstly, in contrast to the weak adversary, this adversary can perform jamming attacks such that the duration of the network partition is unbounded. That is, he/she can jam the network for any number of *consecutive* software update processes and signing operations. Secondly, he/she can physically compromise sensor nodes to capture secret key information. Finally, the strong adversary is classified into two groups: *internal* strong adversary and *external* strong adversary. An *internal* strong adversary is able to obtain the maintenance plan for the WSN to be attacked whereas an *external* strong adversary is not. The maintenance plan shows e.g. the exact time schedule for the software update operations.

Remarks: Since the verifiers store only the public key pk and the public state information st_i^r, node corruptions reveal no secret information. Therefore, w.l.o.g. node compromise attacks are not considered while evaluating the security of the introduced signature scheme against the weak adversary and the strong adversary.

Furthermore, the following observation needs to be noticed: the proposed scheme is an extension of Merkle's one-time signature to a stateful-verifier $\mathcal{T}$-time signature scheme by means of a simple key (state) management. The verifier updates its public key (and also its state) after each valid signature verification. Similarly, the signer updates its private key after each signature generation. This allows the proposed scheme to sign every message with a different public and private key pair providing the security of the one-time signature for each message. However, in cases where the adversary can for example block some sensor nodes from receiving t *consecutive* signature messages, the public keys of those sensor nodes would not be updated. Thus, the adversary may use the keys disclosed in the previous t signatures to output a forgery for those sensor nodes whose states (i.e., public keys) are not updated[1].

[1] Hence, the proposed scheme is referred to as a *stateful-verifier* scheme.

Such an attack can be seen as an existential forgery under a t-known-message attack on the proposed scheme with a *fixed* key pair where the adversary first obtains signatures on t messages of the sender's choice which are *independent* of the hash function. Then, it tries to forge a signature on a new message of its choice. In the adversary model described above, the strong adversary can perform t-known-message attacks whereas the weak adversary cannot. Hence, in the following analysis, the *internal* and *external* strong adversaries are not considered separately. However, a distinction between them is necessary while evaluating the security of the proposed remote programming mechanism in Chapter 6.

Security against the weak adversary Not considering the t-known-message attack, the security of the proposed scheme is based on the difficulty of inverting one-way function f and the collision resistance of the hash function h. Given a signature on a message x, to forge a signature on a message x', the weak adversary Eve (E) is confronted with at least one position where a bit in $h(x)\|\varphi_x$ is 0 and 1 in $h(x')\|\varphi_{x'}$, unless $h(x) = h(x')$.

This is immediately true if a position $h(x)$ is 0 and 1 in $h(x')$. If there is no such position, there must be a position where the bit is 1 in $h(x)$ and 0 in $h(x')$. Then, the number of 0 in $h(x')$ is higher than in $h(x)$ and therefore $\varphi_{x'} > \varphi_x$. Then a position that changes from 0 to 1 can be found in $\varphi_{x'}$.

However, this requires E to compute the inverse of the one-way function for this position what E cannot do better than to a negligible probability, otherwise it obviously contradicts the second pre-image resistance of the hash function h. Considering the brute-force attack, inverting a one-way function of n-bit requires $O(2^n)$ one-way function evaluations. Therefore, independently from the value of $\mathcal{T}$ defining the upper number of allowed signature operations, the probability of forging a signature is 2^{-n} which is negligible in n.

Security against the strong adversary The security of the proposed scheme is validated against the t-known-message attack when the messages are chosen independent of their hash values. Firstly, some notation is introduced:

Table 4.1: Set of messages with distinct signatures for $n = 3$. $h(x)$, φ_x, and σ_x denote the hash, the checksum, and the signature of the message x, respectively.

x	$h(x)$	φ_x	σ_x
x_1	0 0 0	1 1	(s_4, s_5)
x_2	0 0 1	1 0	(s_3, s_4)
x_3	0 1 0	1 0	(s_2, s_4)
x_4	0 1 1	0 1	(s_2, s_3, s_5)
x_5	1 0 0	1 0	(s_1, s_4)
x_6	1 0 1	0 1	(s_1, s_3, s_5)
x_7	1 1 0	0 1	(s_1, s_2, s_5)
x_8	1 1 1	0 0	(s_1, s_2, s_3)

Given a public key $pk = (u_1, \ldots, u_l)$, a private key $sk = (k_1, \ldots, k_l)$, and a signature message $\sigma = (s_1, \ldots, s_l)$, we call u_i the public key parameter, k_i the private key parameter, and $s_i = k_i$ the signature parameter.

Proposition 4.4.1 *Given a $\mathcal{T}$-time signature scheme with a n-bit hash function h. The probability that a signature for a randomly selected message can be forged after having obtained t signatures for uniformly at random chosen messages is $\leq \frac{2^s - t}{2^n}$, where $s \leq n$ denotes the number of distinct signature parameters (i.e., private key parameters) disclosed with those t signatures excluding the checksum part*[1].

PROOF (Sketch): The basic intuition is given by an example. Consider the signature space shown in Table 4.1 for $n = 3$. Furthermore, assume that the strong adversary obtains the signatures σ_{x_2} and σ_{x_5} for the messages x_2 and x_5 chosen independently from their hash outputs. These signature messages reveal three distinct signature parameters (thus private key parameters), namely $s_1 = k_1$, $s_3 = k_3$, and $s_4 = k_4$. There are obviously at most 2^s messages (i.e., x_1, x_2, x_5, and x_6) for which the adversary may generate a valid signature for their hash parts by having the s keys (i.e., k_1 and k_3). However, since t of those messages (i.e., x_2 and x_5) have been already seen, the probability of generating a valid signature for a randomly selected message is $\leq \frac{2^s - t}{2^n}$, when the checksum of the signature is not considered. Obviously, this probability may be even smaller if the checksums of the signatures are considered. For example, the

[1] If $n = 2^m - 1$, the checksum bits are uniformly distributed and the signature including the checksum part can be treated as a hash function that outputs values of length $n + \lfloor \log n \rfloor + 1$.

adversary would not be able to generate the signatures for messages x_1 or x_6 in that case. For simplicity, this case is not considered in the analysis and it is assumed that the probability of a forgery is $\frac{2^s-t}{2^n}$.

Claim 4.4.2 *Given the $\mathcal{T}$-time signature scheme with an n-bit hash function h. Considered only the $h(x)$ part of the signatures, after receiving t signatures, the adversary knows on average $n - \frac{n}{2^t}$ distinct private key parameters.*

Consider a sequence of hash values and let E_i be the random variable describing the first hash value in the sequence where the ith bit equals 1. In a hash value, the probability for every bit to be 0 is $\frac{1}{2}$. Thus, the probability that a bit is in t consecutive hash values 0 is $\frac{1}{2^t}$. For every 1 in the signature a private key parameter leaks. Thus the number of leaked private key parameters after seeing t hash values is $n - \frac{n}{2^t}$.

From Proposition 4.4.1 and Claim 4.4.2, it can be seen that the proposed scheme provides a $\frac{n}{2^t}$-bit security when the strong adversary obtains t signatures. Table 4.2 summarizes the security levels with respect to t. Since $\mathcal{T}$ defines the number of signature generations, where the private keys are derived from an initial private key generated by Gen, one may determine the appropriate value for $\mathcal{T}$ to achieve a desired level of security by using Table 4.2.

Remark: If a high security against strong adversaries is needed, the variant described in the next section can be used. This variant updates the unused keys in every signature, so that no future signature provides any information for changing a 0 bit to a 1 bit in a past signature. For this variant, the strong adversary has no advantage compared to the weak adversary.

Table 4.2: Security level of the proposed stateful-verifier $\mathcal{T}$-time signature scheme against t-known-message attacks for $n = 128$.

t	Security level
1	2^{128}
2	2^{32}
3	2^{16}
4	2^{8}

Efficient security against the strong adversary If security against the strong adversary is needed, the following efficient partial key update can be used. The motivation is that only the keys where no pre-image was revealed must be updated to ensure that the pre-images will not be revealed by future signatures.

The sign algorithm is modified in the following way:

1. Generate uniformly at random $l/2 + \alpha$ secret strings $\kappa_j \leftarrow \{0,1\}^n$ for $1 \leq j \leq l/2 + \alpha$ for a small constant α;

2. Compute one-way chains $u_j = f^{\mathcal{T}}(\kappa_j)$ for $1 \leq j \leq l/2 + \alpha$, where $f^{\mathcal{T}}(k)$ denotes the $\mathcal{T}$-fold composition of $f(k)$ with itself;

3. Compute the signature σ for $x, u_1, \ldots, u_{l/2+\alpha}$ according to Sig;

4. If the number of 0 bits in σ is greater than $l/2+\alpha$, start again from 1. Otherwise, replace k_ℓ by κ_j, where the jth 0-bit appears on the ℓth position of $h(x)||\varphi_x$'s binary representation[1] and set $st^s_{i+1,\ell} = (\mathcal{T}-1)$, where i denotes as in the algorithm description the sequence number of the signature;

5. Append all κ_j for $u_j \leftarrow \{0,1\}^n$ to the signature.

In the same way, the verifier checks the signature on $x, u_1, \ldots, u_{l/2+\alpha}$ and if the check succeeds it replaces the state st^r_{i+1} on the positions where $h(x)||\varphi$ is 0 as described above.

Thus, signer and verifier will with every signature replace all states. Therefore, all signatures base on different hash images and the adversary gains no knowledge by obtaining more than 1 signature when attacking a verifier with an old state.

The cost in terms of signature length is $l/2 + \alpha$, thus, the signature length roughly doubles.

4.4.4 Performance evaluation

Let n be the security parameter. Then, the signature size on average is $n \cdot \frac{l}{2}$ bits, where $l = n + \lfloor log(n) \rfloor + 1$. The public key size is $l \cdot n$ bits. Signature generation and verification require one hash computation with h. Signature generation and verification require on average $\frac{\mathcal{T}}{2} \cdot \frac{l}{2}$ and $\frac{l}{2}$ computations with f, respectively.

[1] These are exactly the positions where the pre-images of the respective verifier state is not released.

The main drawback of the proposed scheme is the large signature size. It is possible to apply the trade-off between the signature size and computational effort as described in [120, page 466]. The basic idea is to sign j-bit of information instead of only one-bit. This reduces the signature size roughly by factor j at the cost of a factor 2^j increased verification time. In the following, the key idea for $\mathcal{T} = 1$ is described.

Again, let x be the message to be signed and $h(x) = x_1 || \dots || x_q$ be its hash in binary representation where each x_i is z-bit long. Compute the integer $x = (q \cdot 2^z) - (x_1 + \dots + x_q)$ and represent it as a binary string $b = b_1 || \dots || b_r$ where b_1 is padded with zeros when necessary. Then, form the binary string $x' = x_1 || \dots || x_q || b_1 || \dots || b_r$. The public key $pk = (u_1, \dots, u_{q+r})$ and the private key $sk = (k_1, \dots, k_{q+r})$ are generated as previously proposed.

The signature is $(s_1, \dots, s_{q+r})$ where $s_i = f^{x_i}(k_i)$ for all $i \in \{1, \dots, q\}$ and $s_i = f^{b_i}(k_{q+i})$ for all $i \in \{1, \dots, r\}$. This grouping reduces the signature size by factor z. However, the signature verification requires g computations with f where $1 \leq g \leq (2^z - 1)$ instead of only one. Therefore, one should consider the requirements of the application before trading the signature size at the cost of the verification time.

5

Broadcast Authentication Schemes and Design Choices

As analyzed in Chapter 3.2.2, one of the fundamental threats to a remote programming is the modification of the software updates during their dissemination. Adversaries might tamper with the software updates and transmit them so as if they were sent by the monitoring device. In order to mitigate such attacks, sensor nodes must be provided with security means providing two main properties: firstly, sensor nodes must be able to verify the origin of the software updates. They should accept only those updates that have been sent by the monitoring device. Secondly, sensor nodes must be able to verify the content of the software updates. They should accept only those updates that have not been manipulated during their dissemination. The first security property is referred to as data origin authentication (or message authentication) [7]. The second property is called data integrity [7]. Data origin authentication provides both data integrity and authenticity.

Software updates are broadcast from a single source to a large number of receivers. In fact, broadcast of data is widely used in praxis. Some examples are the Pay-TV and video-conferencing applications as well as the interactive group games [121]. Due to its wide use in praxis, broadcast security has been addressed by many researches [121]. Hence, there are several broadcast authentication schemes available in the literature. The goal of this chapter is to determine the most appropriate scheme for remote programming. For this, first of all, several schemes from the literature are presented. Subsequently, they are evaluated in terms of the code size as well as the storage, communication, and computation requirements. Notice that this chapter naturally overlaps with [121, 122] to some degree. The analysis performed in this thesis considers the characteristics of remote programming applications.

This chapter is organized as follows: first of all, the assumptions and the notation are introduced. Subsequently, a short taxonomy on the existing broadcast authentication schemes is provided. After that, several schemes are described and evaluated to determine those ones that can be used for protecting the authenticity of software updates.

5.1 Preliminaries and assumptions

This section describes the network model as well as the assumptions.

Network model The network model to be used in the evaluations is composed of a single sender S (i.e., a base station or a sensor node acting as a source node within the WSN) and multiple receivers $r_1, \ldots, r_y$. Figure 5.1 illustrates the considered network model.

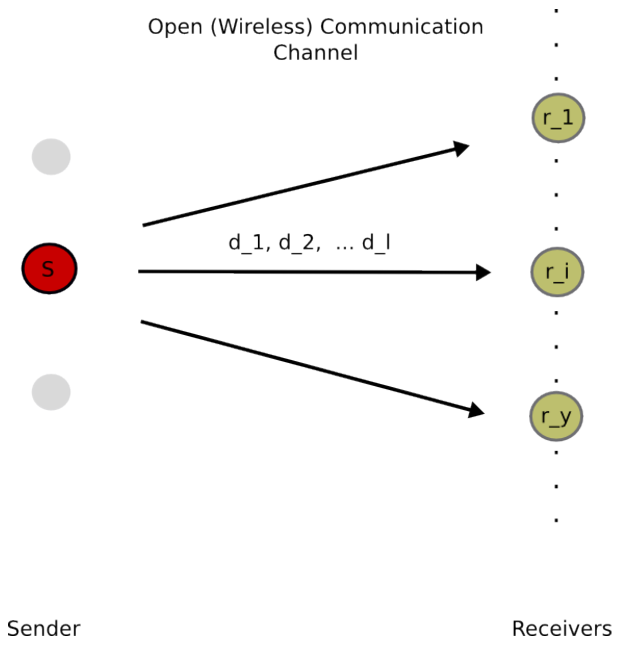


Figure 5.1: Network model to be used in the evaluations of broadcast authentication schemes. - The network is composed of a single source S and y receivers. Source transmits the authenticated data $d_1, d_2, \ldots$ to the receivers via broadcast.

Broadcast data The data D to be disseminated is large (i.e., software update composed of hundreds of packets). It does not fit into one packet of the underlying communication medium. Hence, it is transmitted in l data chunks $d_1, \dots, d_l$.

Security requirements A broadcast authentication scheme for remote programming must be secure against node compromise attacks. In other words, the adversary should not be able to fool a receiver even after compromising λ nodes/keys. A scheme satisfying this security requirement is referred to as λ-secure [123].

Code size A broadcast authentication scheme is assumed to be efficient in terms of code size if it can be implemented using the primitives provided in the code size optimized toolbox (see Chapter 4). Otherwise it is assumed to be inefficient.

Communication overhead Communication overhead is the total size of all authenticators that the sender needs to transmit for authenticating the broadcast data D.

Computation overhead Computation overhead for the sender is the total number of operations that the sender performs for authenticating the broadcast data D. Computation overhead for the receiver is the total number of operations that the receiver performs for verifying the authenticated data D.

Storage overhead Storage overhead of the sender is the size of all key material that need to be stored for authenticating the broadcast data D. Storage overhead of the receiver is the size of all key material that need to be stored for verifying the authenticated data D.

5.2 Taxonomy of the broadcast authentication schemes

Figure 5.2 summarizes the taxonomy of the existing broadcast authentication schemes [121]. Schemes that can even mitigate the adversaries with unlimited computation power are referred to as perfectly secure (also known as unconditionally or information-theoretically secure). Schemes, that can be secure only for a reasonable time, are referred to as computationally secure. The length of this time depends on the computation power of the adversary as well as the underlying mathematical problem. Such

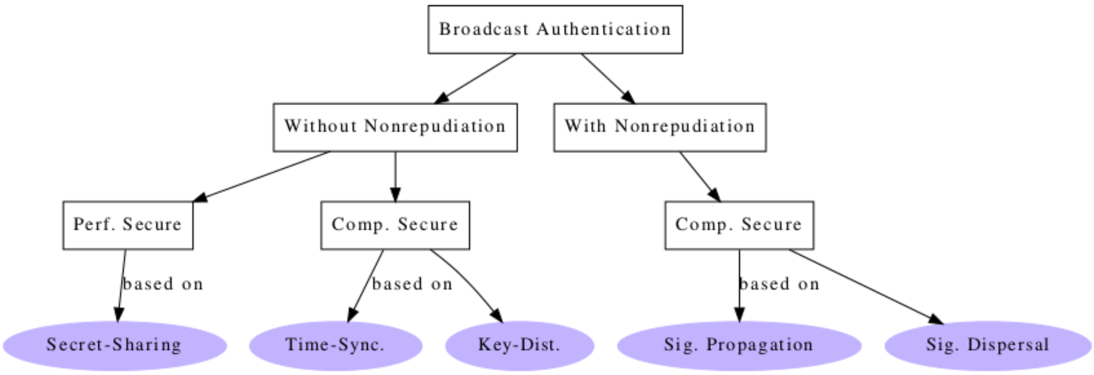


Figure 5.2: Taxonomy of broadcast authentication mechanisms - Broadcast authentication schemes are divided into two groups: schemes with non-repudiation and schemes without non-repudiation. Schemes without non-repudiation are based on either information-theoretic problems (such as secret sharing) or computational problems. Schemes based information-theoretic problems is called perfectly secure schemes. Computationally secure schemes without non-repudiation are based on either time-synchronization or key distribution. Finally, broadcast authentication schemes with non-repudiation are based on digital signatures. They are in turn based on signature propagation or signature dispersal.

schemes are parameterized such that they cannot be broken in a reasonable time with a reasonable success probability.

Perfectly secure broadcast authentication schemes are based on information-theoretic problems such as secret sharing. A secret sharing scheme is a method that enables to share a secret among multiple participants: a secret is divided into a number of parts, called shares. Each participant (i.e. receiver) is given a share such that less than requisite number of shares provide no information about the secret [7]. However, if a sufficient number of receivers collude, they can reconstruct the secret that must be known only to the sender. Hence, broadcast authentication schemes based on secret sharing do not provide non-repudiation[1].

Computationally secure broadcast authentication schemes can be based on symmetric cryptography as well as asymmetric cryptography. In the former case, there is a need for an asymmetry between the authentication and verification keys. That is, the receivers must be able to verify authenticators without being able to create new

[1] A third-party cannot say if a given authenticator has been created by the sender or by a sufficient number of colluding receivers.

authenticators [121]. Two methods are widely used to implement this property. The first one is the delayed disclosure of the authentication keys. Schemes relying on this method need to maintain a loose time synchronization between the sender and the receivers. Hence, they are referred to as time synchronization based schemes. The second method provides the required asymmetry by separating the authentication and the verification keys. That is, the authenticators are verified with a subset of the keys required for the authentication. The receivers cannot generate a valid authenticator since they do not have all the keys required for the authentication. Schemes relying on this method are referred to as key distribution-based schemes. Notice that regardless how the secret-key asymmetry is implemented, broadcast authentication schemes relying on symmetric keys do not provide non-repudiation.

Finally, broadcast authentication schemes can be based on digital signatures. Such schemes inherit the security properties of the underlying signature. Hence, they provide non-repudiation. Broadcast authentication schemes based on digital signatures are further divided into two subclasses: schemes based on signature propagation and signature dispersal [121].

5.3 Evaluation of the existing schemes

5.3.1 Schemes without non-repudiation of origin

Schemes belonging to this class are divided into two subclasses (see Figure 5.2): perfectly secure schemes and computationally secure schemes. In the following, several works relevant to each class are presented.

5.3.1.1 Perfectly secure schemes

DFY scheme Security of the DFY (Desmedt, Frankel and Yung) scheme [124] is based on a (λ, n) threshold secret sharing method similar to [125] . The sender authenticates the broadcast data using a secret key. The receivers verify the received broadcast data using their individual secret shares. Fooling a receiver requires to recover the secret key of the sender. However, this is not possible if $\lambda - 1$ or fewer receivers collude. Notice that the key recovery depends on the number of colluders (i.e, known shares) rather than the available computation power. Hence, the DFY scheme is (perfectly) λ-secure.

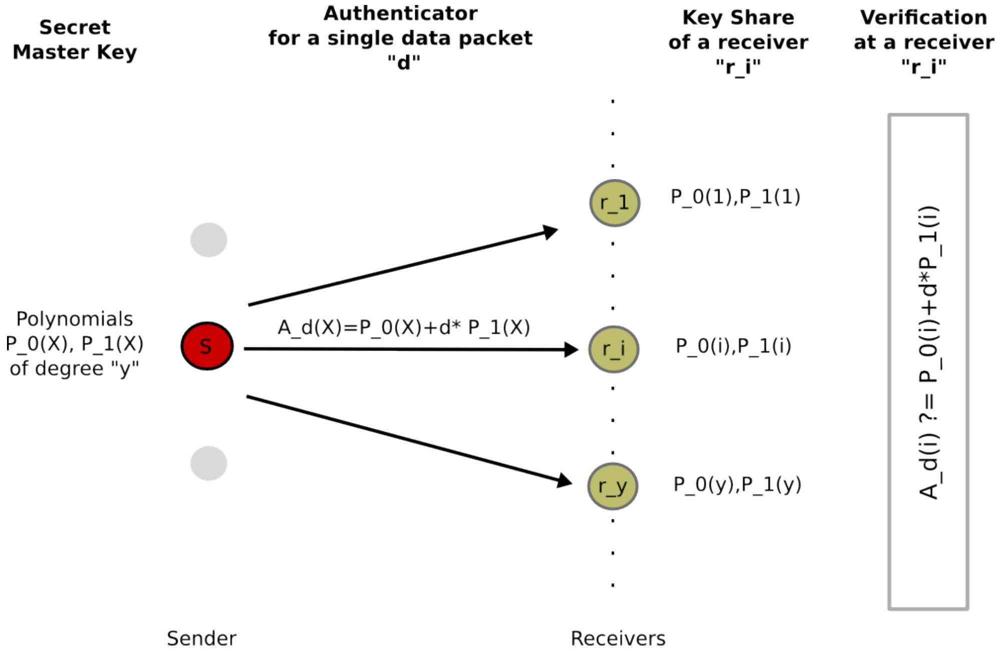


Figure 5.3: DFY scheme - A data packet d is authenticated for broadcast as follows: The sender chooses two polynomials, $P_0(X)$ and $P_1(X)$, of degree y, where y is the number of the receivers. Each receiver r_i is given two shares, $P_0(i)$ and $P_1(i)$, securely. For authenticating the data packet d, the sender computes the authenticator polynomial $A_d(X) = P_0(X) + d \cdot P_1(X)$. It is broadcast to the receivers together with the data packet. Any receiver, r_i, verifies the data packet by checking if $A_d(i) = (P_0(i) + d \cdot P_1(i))$ holds.

Protocol description: The DFY scheme is illustrated in Figure 5.3. Key generation and distribution as well as authentication and verification are defined as follows:

- Key generation and distribution: For each packet d to be authenticated, the sender creates two polynomials $P_0(X)$ and $P_1(X)$ of degree λ^1 and chooses a prime number $p \geq 2^{|d|} \gg l$. l denotes the total number of packets to be authenticated. $P_0(X)$ and $P_1(X)$ are the master secret key of the sender. The sender computes a share $(S_{0,i}, S_{1,i})$ for each receiver r_i with $S_{0,i} = P_0(i)$ mod p and $S_{1,i} = P_1(i)$ mod p. The shares are securely distributed to the receivers.

- Authentication: For authenticating a packet d, the sender computes the polynomial $A_d(X) = (P_0(X) + d \cdot P_1(X))$ mod p. The authenticator polynomial $A_d(X)$ is broadcast to the receivers together with packet d.

- Verification: Each receiver r_i verifies the authenticity of a received packet d by checking if $A_d(i) = (S_{0,i} + d \cdot S_{1,i})$ mod p holds. The received packet is considered authentic if the equation holds.

Protocol evaluation: Let $|a|$ denote the length of a in bits.

- Security: The probability that a coalition of up to λ colluding receivers (or an adversary possessing the secret shares of λ receivers) can authenticate a packet correctly is $1/p$. The probability that at least one receiver accepts the created authenticator is λ/p [124]. This is negligible for a sufficiently large p.

- Communication overhead: Each data packet is authenticated with a polynomial which is $((\lambda + 1) \cdot |p|)$ bits long. Hence, the total overhead for the broadcast data D is $(l \cdot (y + 1) \cdot |p|)$ bits.

- Storage overhead: Each of receiver needs to store two shares ($P_0(i)$, $P_1(i)$) which are $|p|$ bits each. The total overhead for each receiver for verifying the broadcast data D is therefore $(l \cdot 2 \cdot |p|)$ bits. The sender needs to store two polynomials ($P_0(X)$, $P_1(X)$) of degree λ for authenticating each data packet. Thus, the total overhead for the sender for authenticating the broadcast data D is $(l \cdot 2 \cdot (y+1) \cdot |p|)$ bits.

[1] λ is an integer denoting the number of the colluding receivers.

- Computation overhead: For generating an authenticator polynomial, the sender is required to perform $O(\lambda)$ multiplications and additions in $GF(p)$. That is approximately $O(\lambda \cdot |p|^2)$ bit operations [124]. Hence, the total number of bit operations performed by the sender for authenticating l data packets is $O(l \cdot \lambda \cdot |p|^2)$. For verifying a data packet, each receiver needs to perform $O(\lambda)$ multiplications. That is approximately $O(\lambda \cdot |p|^2)$ bit operations. The total computation overhead for verifying l data packets for a receiver is then $O(l \cdot \lambda \cdot |p|^2)$ bit operations.

  Remark: To understand these performance results, let us compare them with the performance of the RSA signature mechanism with a k-bit modulus. For authenticating a k-bit message, the prime number p in the DFY scheme must be at least k bits long. Let p be a k-bit prime number.

  Signing a message with RSA takes $O(k^3)$ bit operations. Verifying a message takes $O(k^2)$ bit operations [7]. On the other hand, both, signing and verifying a message with the DFY scheme take approximately $O(\lambda \cdot k^2)$ bit operations. This means that the DFY scheme is about by factor k/λ faster than RSA in signing a message while by factor λ slower than RSA in verifying a message.

- Code size: The DFY scheme uses multi-precision modular arithmetic in $GF(p)$. Multi-precision modular arithmetic cannot be implemented using the primitives provided by the code size optimized toolbox (see Chapter 4). Additional implementations increasing the code size are necessary. Hence, the DFY scheme is assumed to be inefficient in terms of the code size.

Other perfectly secure schemes In DFY scheme, one key pair (i.e., a secret master key and the corresponding shares at the receivers) can be used to authenticate only one message. Sfavi-Naini et al. [126] improved the DFC scheme by allowing the authentication of w messages with $w+1$ keys. Hence, the storage overhead for the sender is halved ($\approx (w+1)/2w$).

Security of DFC scheme depends on the size of p. It must be larger than the number of possible data packets. Hence, it becomes impractical (in terms of key storage and authenticator sizes) if the number of the packets to be authenticated is very large. In the same work, Sfavi-Naini et al. proposed another scheme relying on cover free set systems to address this problem. The security of their scheme is independent of the number of data packets to be authenticated. Hence, it is more efficient than DFC scheme. However, it is only suitable when the number of colluding receivers (λ) is small in comparison to the total number of receivers (y) [126].

5.3.1.2 Computationally secure schemes

Computationally secure schemes are divided into two main classes: schemes relaying on key distribution mechanisms and schemes relying on time synchronization mechanisms.

5.3.1.2.1 Schemes based on key distribution

CGIMNP scheme The simplest way for constructing a (repudiable) broadcast authentication scheme is to share a global key (k) between the sender and the receivers [123]. The sender authenticates each packet (d) by broadcasting the tag $t = MAC_k(d)$. The receivers verify each received packet by checking if $t = MAC_k(d)$ holds. However, this method is not secure. Any compromised receiver (or who has the key k) can fool the receivers without being detected.

This problem can be solved simply by sharing individual keys with the receivers [123]. Each receiver r_i is given a unique key k_i. The sender authenticates each packet (d) by broadcasting the tag $T = t_1||\ldots||t_y = MAC_{k_1}(d)||\ldots||MAC_{k_y}(d)$. Each receiver r_i verifies the received packet (d, T) by checking if $t_i = MAC_{k_i}(d)$ holds. This method is secure. However, it becomes impractical in terms of the tag size when the number of the receivers is large.

The CGIMNP (Canetti, Garay, Itkis, Micciancio, Naor and Pinkas) scheme [127] is a trade-off between these two simple methods. It is more secure than the first approach and more efficient than the second approach. [123] shows that the CGIMNP scheme has the optimal tag length for a broadcast authentication based purely on pseudorandom functions.

Protocol description: The CGIMNP scheme is illustrated in Figure 5.4. Key generation and distribution as well as authentication and verification are defined as follows:

- Key generation and distribution: For achieving n-bit per-packet unforgeability

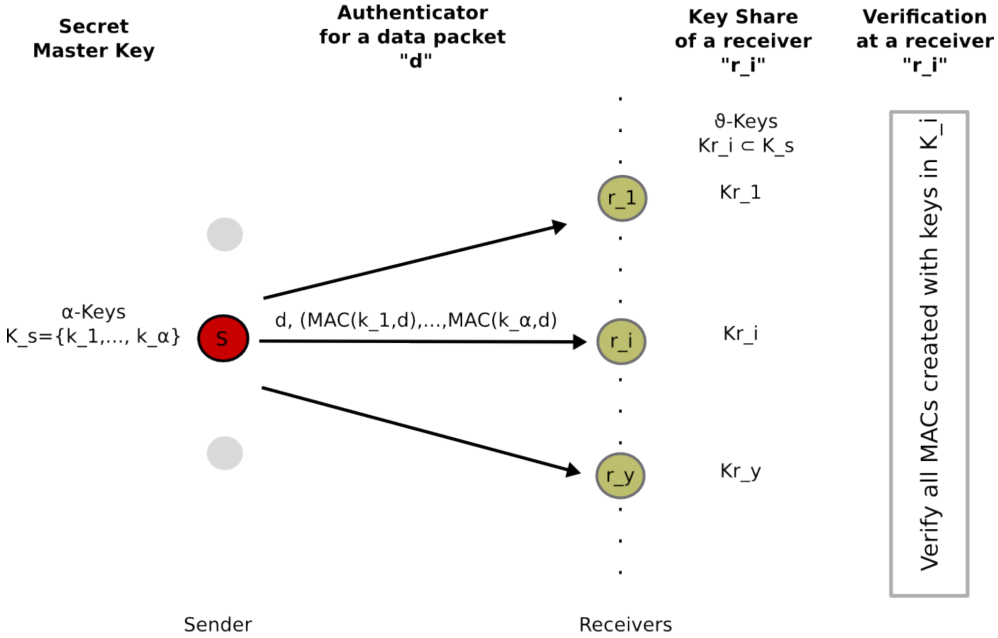


Figure 5.4: CGIMNP scheme - The sender is given a set of α random keys $K_s = \{k_1, \ldots, k_\alpha\}$. Each receiver r_i is given a set of ϑ keys K_{r_i} such that $K_{r_i} \subset K_s$. For authenticating a data packet d, the sender computes an α-bit MAC tag $T = t_1|| \ldots ||t_\alpha$, where t_i is one-bit output of a MAC result computed with key k_i. The sender transmits the tag and the data packet to the receivers. Each receiver r_i can then verify the authenticity of a broadcast packet (d, T) by checking the tag bits which were created using the keys in its subset K_{r_i}.

against a coalition of up to λ colluders[1], the sender is given a set of α keys K_s, where $\alpha = 4e \cdot (\lambda + 1) \cdot \ln(2^n)$. Each receiver r_i is given a subset $K_{r_i} \subset K_s$ composed of $\vartheta = 4e \cdot \ln(2^n)$ keys[2]. Notice that all keys are n bits long. The size of ϑ depends on n, i.e., the desired level of MAC security. The size of α depends on the number of colluders λ as well as ϑ.

- Authentication: For authenticating a packet d, the sender computes an α-bit tag $T = t_1||\ldots||t_\alpha = Trc(MAC_{k_1}(d))||\ldots||Trc(MAC_{k_\alpha}(d))$, where $Trc(\cdot)$ denotes the truncation of a MAC result to a single bit. The packet-tag pair (d, T) is broadcast to the receivers.

- Verification: For verifying a received packet (d, T), each receiver r_i checks the tags which were created using the keys in its key set K_{r_i}.

Protocol evaluation:

- Security: Assume that $\alpha = 4e \cdot (\lambda + 1) \cdot \ln(2^n)$ and $\vartheta = \alpha/(\lambda + 1)$. Then, with probability $(1 - 1/2^n)$, the CGIMNP scheme is n-bit per-packet unforgeable in the case of up to λ compromised keys [127].

- Communication overhead: Each authenticator is α bits. Hence, the total overhead for the entire broadcast data D is $(l \cdot \alpha)$ bits.

- Storage overhead: Each of receiver needs to store ϑ MAC keys. The total overhead for each receiver is therefore $(\vartheta \cdot n)$ bits. The sender needs to α MAC keys. Thus, the total overhead for the sender for authenticating the entire broadcast data D is $(\alpha \cdot n)$ bits.

- Computation overhead: Authenticating a packet requires α MAC operations. Hence, the total overhead for the sender for authenticating the entire broadcast data D is $(l \cdot \alpha)$. Verifying a packet requires ϑ MAC operations. Hence, the total overhead for each receiver for verifying the entire broadcast data D is $(l \cdot \vartheta)$ MAC operations.

[1] Assume that the adversary is given the keys of λ receivers. A broadcast authentication scheme is n-bit per-packet unforgeable if the adversary cannot forge a MAC to fool a receiver without knowing its keys with probability better than $1/2^n$ [127].

[2] Each key is selected from K_s with probability $1/(\lambda + 1)$.

- Code size: The CGIMNP scheme is based on MACs. Hence, it can be implemented using the toolbox provided in Chapter 4. No additional implementation increasing the code size is necessary. Hence, the CGIMNP scheme is assumed to be efficient in terms of the code size.

5.3.1.2.2 Schemes based on time synchronization

TESLA broadcast authentication protocol TESLA (Timed Efficient Stream Loss-tolerant Authentication) [128, 129] is based on a loose time synchronization between the sender and receivers. It works as follows: let $d_{i,j}$ denote the ith packet to be authenticated in an interval j. The sender computes the tag $t_{i,j} = MAC_{k_j}(d_{i,j})$ and broadcasts it together with the packet to the receivers. The key k_j is known only to the sender. The receivers cannot verify the packets received in interval j immediately, since the key k_j is unknown to them. Hence, the receivers store them. After a certain delay (e.g., in the following interval $j + 1$), the sender broadcasts the secret key k_j to the receivers. The receivers verify the authenticity of the packets received in the interval j by checking $t_{i,j} = MAC_{k_j}(d_{i,j})$.

Protocol description: The TESLA protocol is illustrated in Figure 5.5. Key generation and distribution as well as authentication and verification are defined as follows:

- Key generation and distribution: For achieving n-bit per-packet unforgeability, the sender and the receivers are given two one-way functions f_1 and f_2 mapping n-bit inputs to n-bit outputs. The sender picks an n-bit random number k_y and computes a key chain consisting of $y \gg l$ elements[1], $k_{y-x} = f_1^x(k_y)$, where $f_1^x(\cdot)$ denotes x consecutive applications of f_1, i.e. $k_{y-x} = \underbrace{f_1(f_1(\ldots f_1(k_y)))}_{\text{x times}}$.

  Furthermore, the sender divides the transmission time into fixed-length intervals and associates each key of the key chain with one interval (i.e., the sender uses the key k_j for in the interval j). Finally, the sender disseminates the length of the key chain, the interval schedule (interval duration, start time, current interval index j), the key disclosure delay Δ (number of intervals), and the key $k_{j-\Delta}$ to the receivers via a secure channel. The sender keeps k_y secret.

[1] The sender can authenticate at most y packets with such a key chain. A new key chain needs to be created for authenticating more packets.

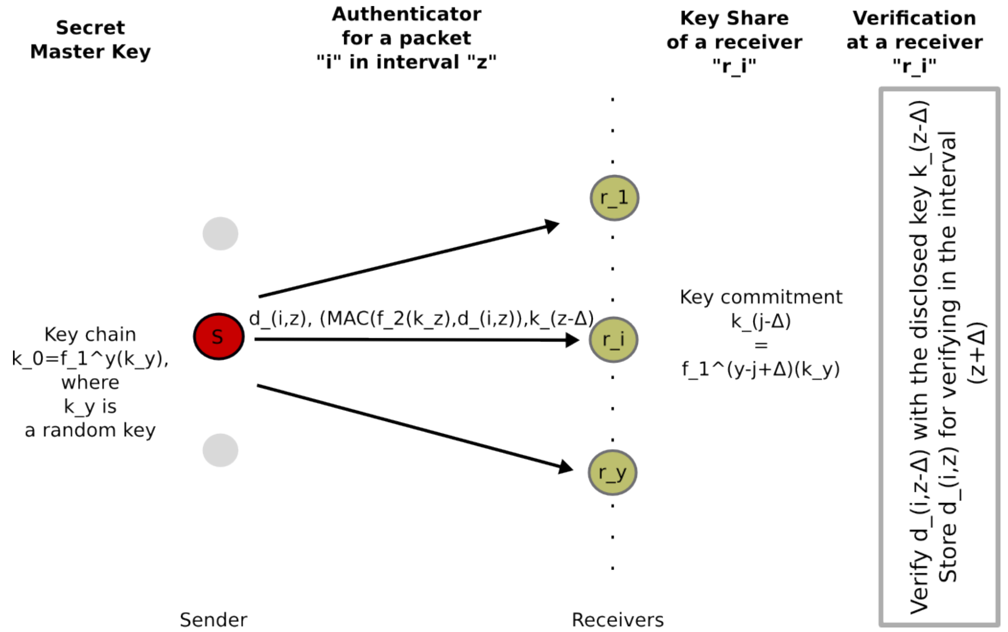


Figure 5.5: TESLA protocol - For authenticating l packets for broadcast: The sender is given a random key k_y. The sender computes a one-way key chain of length $y \gg l$ with $k_0 = f_1^y(k_y)$. The receivers are given the length of the key chain, the interval schedule (interval duration, start time, current interval index j), the key disclosure delay Δ (number of intervals), and the key $k_{j-\Delta} = f_1^{y-j+\Delta}(k_y)$. The sender computes the tag $t_{i,z} = MAC_{f_2(k_z)}(d_{i,z})$ for each data packet $(d_{i,z})$ to be broadcast in interval $z > (j + \Delta)$. The sender broadcasts the triple $(d_{i,z}, t_{i,z}, k_{z-\Delta})$ to the receivers. The receivers verify the packets $d_{i,z-\Delta}$ by checking $t_{i,z-\Delta} = MAC_{f_2(k_{z-\Delta})}(d_{i,z-\Delta})$. Packets $d_{i,z}$ received in interval z are stored for to be verified later in interval $z + \Delta$.

- Authentication: For authenticating the packets $d_{i,z}$ in interval $z > (j + \Delta)$, the sender computes the MAC tag $t_{i,z} = MAC_{f2(k_z)}(d_{i,z})$ and broadcasts the triple $(d_{i,z}, t_{i,z}, k_{z-\Delta})$ to the receivers.

- Verification: For verifying a broadcast packet $(d_{i,z}, t_{i,z}, k_{z-\Delta})$ received in the interval z, the receivers must wait Δ intervals until k_z gets disclosed. Once k_z is disclosed, the receivers verify the packets by checking $t_{i,z} = MAC_{f2(k_z)}(d_{i,z})$.

  Notice the receivers accepts the packets authenticated with a safe key only. A safe key is a key, which is known only to the sender. Consider a triple $(d_{i,\pi}, t_{i,\pi}, k_{\pi-\Delta})$ received in the interval π. Firstly, the receiver verifies the legitimacy of the disclosed key $k_{\pi-\Delta}$ by checking $k_{j-\Delta} = f_1^{\pi-j}(k_{\pi-\Delta})$. Subsequently, based on the time synchronization, the receiver estimates the latest possible time interval π' that the sender could be in. If $\pi' < \pi + \Delta$, then the packet is safe.

Protocol evaluation:

- Security: Considering safe packets, n-bit k_y and n-bit pseudorandom functions f_1 and f_2, TESLA protocol is n-bit per-packet unforgeable.

- Communication overhead: Each authenticator is composed of an n-bit tag. Additionally, in each of Δ periods, a key is disclosed. Hence, the total overhead for authenticating the entire broadcast data D is $(n \cdot (l + \lceil l/\Delta \rceil))$ bits.

- Storage overhead: Each of receiver needs to store an n-bit commitment to the key chain. The receivers can verify the received packets only with a delay of Δ interval. Hence, in a setting, where β packets per interval are being broadcast, each receiver needs to store $(n \cdot (1 + \beta\Delta))$ bits in total. The sender needs to store an n-bit key k_y.

- Computation overhead: Authenticating a packet $d_{i,j}$ requires the sender to perform one MAC operation. Moreover, computing the current element of the key chain k_j requires $(\lceil y/\beta \rceil - j)$ operations with f_1. Additionally, one computation with f_2 is performed to compute the mac key $f_2(k_j)$ for each period j. The number of operations[1] required for authenticating l packets is then

$$T_{(f_1,f_2)} = \underbrace{(y'(l'+1) - l'(l'+1)/2)}_{f_1 \text{ computations}} + \underbrace{l'}_{f_2 \text{ computations}} ,$$

where $l' = \lceil l/\beta \rceil$ and $y' = \lceil y/\beta \rceil$. Hence, the total overhead for the sender for authenticating the entire broadcast data is l MAC computations and $T_{(f_1,f_2)}$ computations with a one-way function.

Verifying a packet requires the receivers to perform one MAC evaluation. Moreover, one computation with f_1 is required to verify the disclosed key in each interval. Similarly, one computation with f_2 is required to compute the current MAC key. Hence, the total overhead for each receiver for authenticating the entire broadcast data is l MAC evaluations and $2l'$ computations with a one-way function.

- Code size: TESLA protocol requires a loose time synchronization between the sender and the receivers. Time synchronization cannot be implemented using the primitives provided by the code size optimized toolbox (see Chapter 4). Additional implementations increasing the code size are necessary. Hence, TESLA is assumed to be inefficient in terms of the code size.

Other schemes based on time synchronization The Chained Stream Authentication Protocol (CSA) [131, 132] is very similar to TESLA. The main difference is that CSA sends only one packet in each interval (i.e., $\beta = 1$). Moreover, the key disclosure delay is only one interval (i.e, $\Delta = 1$). That is, a key used in the interval j is disclosed in the next interval $j + 1$. Hence, the receivers need to store the received packets only one interval. Therefore, CSA is superior to TESLA in terms of the storage overhead. However, CSA authenticates each packet with a different key, while a single key in TESLA is used for authenticating β packets. Hence, TESLA is more efficient (by factor β) than CSA in terms of the communication overhead.

μTESLA [133] is designed for WSNs. There are two main differences between the two schemes. Firstly, μTESLA uses the elements of the key chain as the MAC key.

[1] This overhead might be significantly reduced by slightly increasing the storage overhead: [130] shows that any element of a key-chain of length y can be computed with $log(y)$ computations by storing $log(y)$ elements of the key chain.

Hence, μTESLA requires only one pseuderandom function f_1 instead of two (f_1 and f_2) as in TESLA. Secondly, the keys are disclosed in a special packet instead of in the data packets as done in TESLA.

BAP (Broadcast Authentication using Cryptographic Puzzles) was proposed in [134]. BAP maintains the secrecy of the keys by hiding them in a cryptographic puzzle instead of delayed key disclosures. BAP puzzles are broadcast before the data packets to ensure on-the-fly packet verifications. BAP assumes that the adversary cannot solve the puzzles faster than the receivers. However, this assumption is too strong in particular for WSN applications[1].

The main challenge common to all methods above is the secure transmission of the key commitment ($k_{j-\Delta}$). In TESLA, it is signed with a public key signature scheme. However, using public key signatures in WSNs is impractical. Hence, μTESLA proposes to use an authenticated channel. The authenticated channel is established with a pre-shared secret key between the sender and the receivers. However, this method does not scale well in large networks. As a solution, Liu et al. [135] proposed a method based on multiple key chains instead of only one. It requires neither a public key signature nor an authenticated channel. Later, it was improved to support multiple senders and also to mitigate DoS attacks in [136].

Bootstrapping the receivers with a key chain of infinite length is another approach solving the key commitment problem. [137] shows how to build such a key chain with constant storage and computational requirements. It is based on chameleon functions [138] and can be used to authenticate a practically infinite number of packets. Hence, using such a key chain removes the dependency on the public key signatures, authenticated channels as well as multiple key chains.

5.3.2 Schemes with non-repudiation of origin

Non-repudiation is a security property provided by digital signatures. Hence, a natural way of implementing a non-repudiable broadcast authentication scheme is to use it in conjunction with digital signatures. The simplest approach is to sign all broadcast packets. However, this approach is impractical because digital signatures are typically resource hungry. Hence, more efficient approaches are required.

[1] The adversary might have e.g. a laptop to solve a puzzle while sensor nodes have a microcontroller.

5.3.2.1 Schemes based on signature propagation

GR scheme The GR (Gennaro, Rohatgi) scheme [12] requires to sign only one packet of the entire broadcast data. This amortizes the large overhead of the signature over several packets. Each packet is authenticated with its predecessor. The very first packet is authenticated with the signature. Since each packet verifies the following one, non-repudiation of the signature is spread thorough the entire broadcast data.

Two broadcast authentication methods are described in [12]. The first one is to authenticate broadcast data that the sender knows entirely in advance. The second one is to authenticate broadcast data which is unknown to the sender advance. Software updates are known to the sender entirely in advance. Thus, this thesis describes only the former method which is referred to as off-line signing in [12].

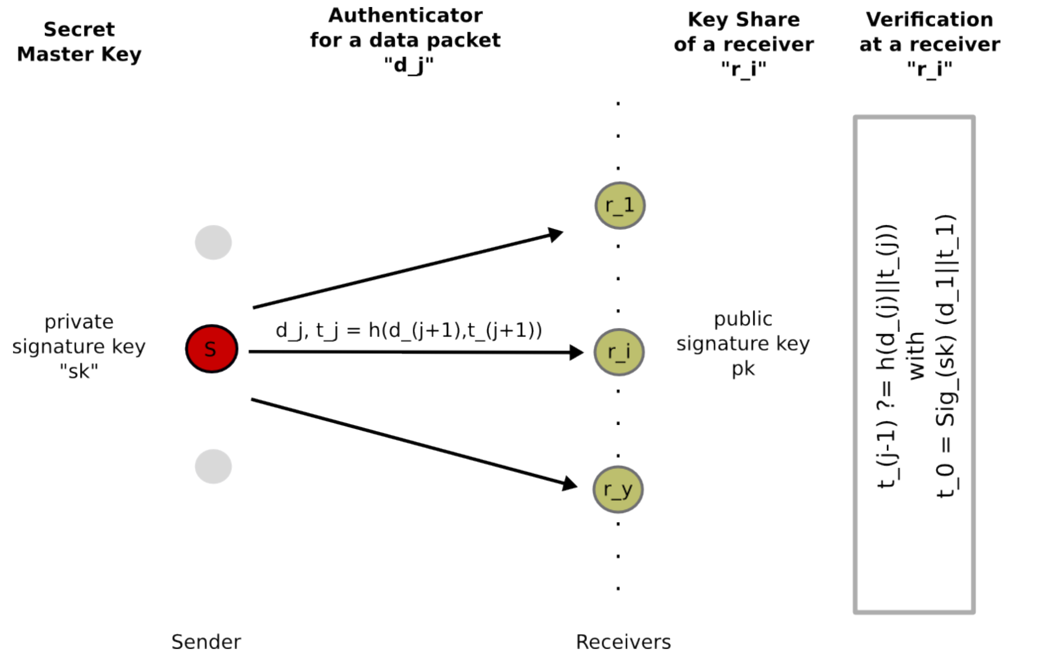


Figure 5.6: GR scheme (off-line case) - For authenticating the data $D = d_1, \ldots, d_l$ for broadcast: The sender generates a key pair pk, sk for the signature mechanism. The public key pk is given to the receivers securely. The sender keeps the private key sk secret. The sender broadcasts the packet-tag pairs $(d_i, t_i = h(d_{i+1}||t_{i+1}))$ to the receivers. The receivers store the tag t_i until the following packet (d_{i+1}, t_{i+1}) gets received and verified. Notice that t_0 is verified with the signature and $t_l = 0x00\ldots$.

Protocol description: The GR scheme is depicted in Figure 5.6. Key generation and distribution as well as authentication and verification are defined as follows:

- Key generation and distribution: The sender and the receivers are given a secure signature scheme $\mathrm{S} = (\mathrm{Gen}, \mathrm{Sig}, \mathrm{Vrfy})$ and a collusion resistant hash function h. For achieving n-bit per-packet unforgeability, the sender generates a public-private key pair using the key generation algorithm $(pk, sk) \leftarrow \mathrm{Gen}(1^{\mathrm{n}})$. The public key pk is given to the receivers. The sender keeps the private key sk secret.

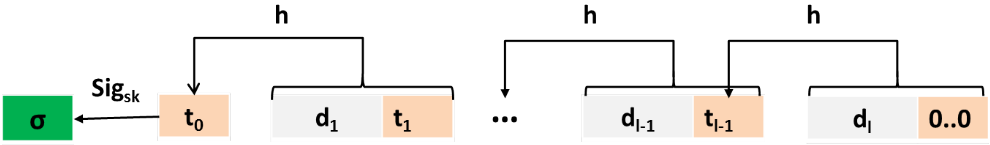


Figure 5.7: Authenticator chain in GR scheme (off-line case) - Authenticator chain, $t_0, \dots, t_l$, is computed backwards with $t_i = h(d_{i+1}||t_{i+1})$. The head of the chain, t_0, is authenticated with the signature $\sigma = \mathrm{Sig_{sk}(t_0)}$.

- Authentication: As shown in Figure 5.7, authenticators t_i of each packet are computed recursively as follows:

$$t_i = h(d_{i+1}||t_{i+1}), \tag{5.1}$$

where $t_l = 0x00\dots$ and $i = (l-1), \dots, 0$. Subsequently, t_0 is signed with a signature scheme as

$$\sigma = \mathrm{Sig_{sk}(t_0)}. \tag{5.2}$$

Finally, the sender broadcasts $(\sigma, t_0), (d_1, t_1), \dots, (d_l, t_l)$ in this order.

- Verification: The receivers first verify the signature, $\mathrm{Vrfy_{pk}(t_0}, \sigma)$. A valid signature ensures the authenticity of t_0. Similarly, each valid tag t_i ensures the authenticity of the subsequent packet as $t_i = h(d_{i+1}||t_{i+1})$. Notice that, each tag needs to be stored until the packet, that it authenticates, gets verified.

Protocol evaluation:

- Security: If the underlying signature scheme and the hash function are secure, then the off-line signing is secure [12]. For n-bit tags, it is n-bit per-packet unforgeable. In other words, for a sufficiently large n, the probability of generating a fake authenticator, which is valid for a receiver, is negligible.

- Communication overhead: Each packet is authenticated with an n-bit hash value. Additionally, one signature packet needs to be broadcast at the beginning. Hence, the total overhead for the broadcast data D is is $(|\sigma| + l \cdot n)$ bits.
- Storage overhead: Each receiver needs to store a public key and an authenticator t_i for verifying the next packet d_{i+1}. Hence, the total overhead for the receivers for verifying the entire broadcast data D is $(|pk| + n)$ bits in total. The sender needs to store only the private key sk.
- Computation overhead: The sender performs one signature and l hash computations. The receivers perform one signature verification and l hash computations.
- Code size: This approach can be used in combination with the $\mathcal{T}$-time signature described in Section 4.4. Hence, this approach is very efficient in terms of the code size.

Other schemes based on authenticator chains The GR scheme is not robust against packet losses. Contrary to the GR scheme, the EMSS [128] approach tolerates the packet losses. The tolerance is obtained by redundancy. That is, authenticators of multiple (randomly chosen successors) packets are added to each packet. Hence, the authenticator chain stays verifiable even if some packets are lost. EMSS [128] determines the redundant authenticators randomly. A deterministic approach is proposed in [139]. The number of authenticators added to each packet is referred to as redundancy degree [121]. A^2Cast [140] improves EMSS by dynamically adapting the redundancy degree to the actual packet-loss ratio. Several other improvements [141, 142, 143] were proposed by the authors of A^2Cast. The main drawback of all those schemes is the requirement on a feedback channel for obtaining the actual packet-loss ratio. Providing such a channel for WSNs is not only a challenging task, but also increases the code size and the implementation complexity.

5.3.2.2 Schemes based on signature dispersal

All schemes presented above assumes the signature packet is received correctly. However, due to the packet losses, signature packets might be received only in part. In such a case, traditional approaches require the retransmission of the signature packet. This is not very efficient if multiple signatures are broadcast during the data transmission. Signature dispersal is a technique that tolerates the packet losses by dispersing the signature in multiple packets. The main idea is to recover the signature information even if its some parts are lost. In a remote programming scenario, only one signature needs to be broadcast at the beginning. Hence, the potential gain from implementing a signature dispersal would likely to be less than the computational and code size overhead caused. For this reason, this thesis skips the further discussion of the signature-dispersal based schemes. The reader is referred to [121, 122] for a more detailed overview.

5.4 Design decisions

Perfectly secure schemes The main disadvantage of the DFY scheme [124] is the key storage size. Receivers are required to use different keys for verifying different packets. This results in a very large storage overhead when they are used in an application like remote programming. Software updates in general are composed of several hundreds or even thousands of packets. Thus, storing an individual key for each packet becomes impractical for resource constrained platforms such as sensor nodes. Furthermore, a secure channel is required for transmitting the secret keys (shares) to the receivers. This requirement limits its use in WSNs. [126] improves the DFY scheme by reducing the key storage size by a factor of 2. However, the penalty for this improvement is the poor performance. Moreover, there is still a need for a secure channel to refresh the secret keys (shares) of the receivers.

CGIMNP scheme The main drawback of the CGIMNP scheme is that the number of colluding users must be known in advance. Furthermore, the size of the key storage as well as the authenticators increases linearly with the number of the compromised receivers. Hence, this scheme becomes impractical to be used in WSNs when a strong level of security is required. For example, each sensor node needs to store 603 MAC keys for obtaining 80-bit per-packet security. It is 1206 keys for the sender. Moreover, this increases linearly with the number of compromised nodes. Using 1206 keys at the sender produces 1206 tag bits for each packet. A scheme with such a large communication overhead is obviously not practical for the remote programming scenario. The main advantage of the CGIMNP scheme is that it is very efficient in terms of the code size.

Schemes based on time synchronization The main drawback of the schemes based on delayed key disclosure is the dependency on a time synchronization. Contrary to the typical workstations and servers, sensor nodes are resource limited devices without Internet connection and complicated clock implementations. Hence, it is challenging to keep the clock of the receivers synchronized with the sender's clock. Once the clock of the receivers varies from the sender's clock more than a threshold value, all the schemes introduced above become insecure. Moreover, implementing a time synchronization protocol naturally increases the code size. Hence, broadcast authentication mechanisms based on time synchronization are not suitable for the remote programming of WSNs when the code size is important.

GR scheme The GR scheme is very efficient in terms of the code size as well as the computation, communication, and storage overheads. However, its main drawback is that verifications are sequential. That is, a packet d_i can be verified only if the previous packet d_{i-1} has already been received. Furthermore, if a packet is lost, all subsequents packets cannot be verified anymore. This drawback widely prohibits its use in streaming applications (video, audio, etc.) in the Internet.

However, remote programming applications have different characteristics and requirements than streaming applications in the Internet. Most importantly, the order of the received packets in a software update is not as critical as in a video or audio streaming. A small delay in receiving a few tens of packets might impact the quality of a video or audio stream significantly. However, a small delay in receiving the packets of a software update can be tolerated in a remote programming application. What important here is that the receivers must have received the complete list of packets at the end of the dissemination process.

Conclusions Based on the observations above, this thesis proposes to use an improved version of the GR scheme for authenticating the software updates. The basic GR scheme is not robust against packet losses at all. As described in the following chapter, this thesis improves the packet-loss tolerance of the GR scheme by applying multiple authenticator chains instead of only one. Moreover, this thesis proposes to use the CGIMNP scheme for verifying the signature packet. This is necessary to mitigate the DoS attacks based on fake signatures[1]. The CGIMNP scheme is based on MAC operations only. Hence, DoS mitigation is obtained without increasing the code size.

[1] The CGIMNP scheme is not efficient in terms of the communication and storage overheads when a high-level of security is required. However, it can be used as a DoS filter against fake signatures with small security parameters.

6

Authenticating Software Updates for Remote Programming

Protecting the authenticity as well as the integrity of the software updates is an essential security requirement for the remote programming protocols. Otherwise, adversaries may program the sensor nodes with malicious and corrupted software updates or deplete their limited power by launching DoS attacks (see Chapter 3).

This chapter presents an efficient and code-size optimized approach, originally proposed in [3], for protecting the authenticity and integrity of the software updates. It relies on an improvement of the GR scheme for the offline signing [12] (see Section 5.3.2.1) as many previous approaches [72, 73, 74, 75] do.

It differs from the previous approaches in two aspects. Firstly, it exploits the stateful-verifier $\mathcal{T}$-time signature scheme to minimize the code size requirement. Secondly, it defeats DoS attacks better than the previous approaches by using multiple hash chains at multiple granularities, namely at packet and page levels.

6.1 Preliminaries and assumptions

This section describes the network model as well as the assumptions.

Remote programming tool The approach presented in this chapter is independent of the underlying remote programming protocol. It can be applied to any remote programming protocol. Deluge [144] is a de-facto standard among the remote programming tools. Hence, this thesis uses Deluge as the underlying programming tool in implementing the prototype of the proposed approach.

Deluge disseminates a software update as follows. Firstly, the software update is divided into fixed-size pages. The size of the pages is chosen such that a page can fit into the available RAM on the sensor nodes. Subsequently, each page is divided into fixed-size packets. The size of the packets depends on the underlying network protocol. Finally, the software update is transmitted packet-wise in a page-by-page mode to all nodes in the WSN. Deluge disseminates the pages strictly in a sequential order. That is, sensor nodes start receiving a page P_i only if the previous page P_{i-1} has been completely received. However, the packets of a page can be received in any order.

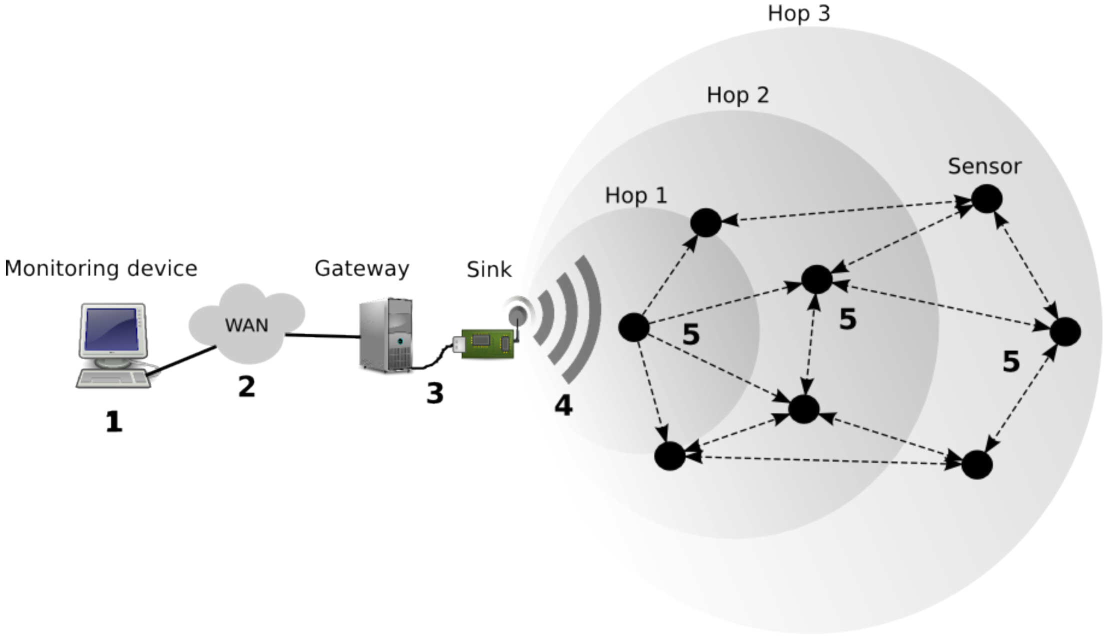


Figure 6.1: Network model for remote programming - The new version of the software is authenticated and delivered to the gateway of the WSN via WAN (1,2). The gateway node divides the software update into packets and disseminates them to the sensor nodes via the sink node (3,4). Sensor nodes at each hop propagate the received packets to the sensors at other hops (5).

Network model Figure 6.1 depicts the generic remote programming architecture considered in this thesis (see Section 1.2.1). The new version of the software is authenticated and delivered to the gateway of the WSN via WAN (1,2). The gateway divides the software update into a sequence of data packets and broadcast them to the nodes at the first hop through the sink node (3,4). Finally, the receivers at each hop deliver the received packets to the nodes at the next hop recursively up to the receivers at the last hop (5).

Assumptions The monitoring device as well as the gateway is a powerful device, for example a workstation. Software updates to be disseminated are known to the monitoring device in advance. Sensor nodes are equipped with a non-volatile memory (e.g., EEPROM) that keeps its content even if the power is removed. Sensor nodes are not tamper resistant. Finally, sensor nodes are bootstrapped with the key material required by the presented approach in the pre-deployment phase of the WSN (or before the very first remote programming operation).

6.2 Proposed approach

The proposed approach is based on three security primitives: the $\mathcal{T}$-time signature scheme, a hash tree, and multiple hash (authenticator) chains. The signature is used to bootstrap the security of the software update. The hash three is used to propagate the security of the signature to the first page of the software update. Finally, hash (authenticator) chains are used to propagate the security of the signature to the remaining pages of the software update.

The use of a hash tree in a combination with multiple hash chains was first proposed in [74] to prevent DoS attacks. Later, this approach was improved in [75] by storing a hash tree of hash chains rather than a hash chain of hash trees. Finally, it was improved by Ugus et al. [3] in terms of communication overhead by applying two hash functions h_1 and h_2 with output of different lengths.

Let $SU_{Auth} = P_0||P_1||P_2||\dots||P_y$ denote the resulting software update after a software update SU is authenticated. Then, P_0 is the signature page carrying a $\mathcal{T}$-time signature[1]. This page is used to bootstrap the security of the software update. The

[1] The $\mathcal{T}$-time signature scheme is an appropriate choice for the resource restricted sensor devices due to its efficiency. However, any secure signature scheme can be used.

second page P_1 carries the packets of the hash tree. Finally, the pages P_2 to P_y carry the actual software update.

Key generation and distribution as well as authentication and verification are defined as follows:

6.2.1 Key generation and distribution

The sender (i.e., the monitoring device) and the receivers (i.e., the sensor nodes) are given the $\mathcal{T}$-time signature scheme and a collusion resistant hash function h. The sender generates a public-private key pair (pk, sk) for the signature. The public key pk is given to the receivers. The sender keeps the private key sk secret. Furthermore, two security parameters κ and ρ are given to the monitoring device and the receivers. They represent the output lengths of two hash functions h_1 and h_2. These operations are performed in the pre-deployment phase of the sensor nodes (i.e., before the very first remote programming operation).

6.2.2 Authentication

Authenticating a software update SU is performed in three steps: partitioning the plain software update, constructing the hash chains and the hash tree, and finally signing the software update.

Partitioning the software update The software update SU is divided into pages and the pages are divided into packets. The page and packets sizes are chosen according to the underlying remote programming tool (see Section 6.1). Recall that the optimal page size depends on the RAM available on the sensor nodes. The optimal packet size depends on the underlying network protocol.

Starting from P_2, the data of the plain SU is filled into the packets leaving κ bits in each packet for later addition of the hash chains. Moreover, ρ bits space is reserved in the last packet of each page except the last page. The last page P_y will be shorter than pages P_2 to P_{y-1}.

Example 4 *Consider a remote programming tool with the page size*[1] *of three packets and with the packet length of 5 bits. Figure 6.2 illustrates the partitioning of a page for*

[1] After addition of κ and ρ bits spaces.

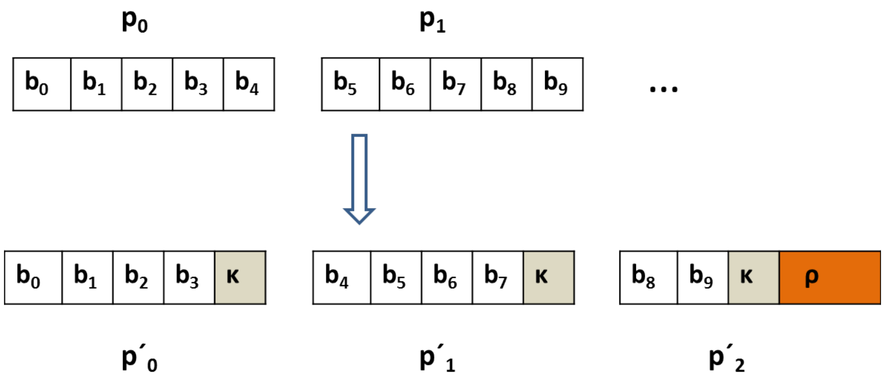


Figure 6.2: Partitioning a software update - Assume that the page size is three packets and the packet size is 5 bits. Furthermore, assume that $\kappa = 1$ and $\rho = 2$. Then, a page is partitioned as follows: the first two packets, p'_0 and p'_1, reserve κ bits space for later addition of the hash chains. The last packet of the page p'_2 reserves an additional ρ bits space.

$\kappa = 1$ *and* $\rho = 2$. *In the first two packets,* p'_0 *and* p'_1, κ *bits space is reserved for later addition of the hash chains. The last packet of the page* p'_2 *reserves an additional* ρ *bits space.*

Constructing the hash chains and hash trees Firstly, the hash (authenticator) chains between the pages from P_{y-1} to P_1 are created. Later, the hash tree page P_1 is formed. Finally, the signature page P_0 is constructed. Notice that the last page P_y does not contain any hash value.

Let $p_{i,j}$ denote the jth packet of the ith page and h_1 and h_2 be two hash functions with output lengths κ and ρ, respectively ($\kappa < \rho$). To construct the hash (authenticator) chains, the hash of the jth packet from the ith page, $h_1(p_{i,j})$, is appended to the jth packet of the $i - 1$th page, $p_{i-1,j}$. Moreover, the hash of the ith page, $h_2(P_i)$, is appended to the previous page P_{i-1}. This is done for all pages in the reverse order, namely $P_y, \ldots, P_2$.

Example 5 *Consider a software update partitioned as described in Figure 6.2 above. Figure 6.3 shows how to construct the hash chains between two pages (*P_i *and* P_{i-1}*) of the partitioned software.*

After constructing the hash chains, the hash tree needs to be formed. It is done using the κ-bit hash values stored in the packets of the page P_2: Firstly, the hash values

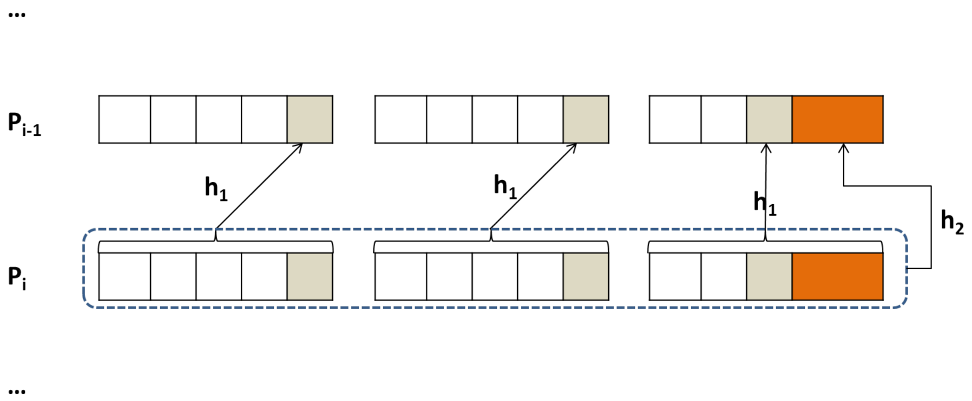


Figure 6.3: Constructing hash chains on a partitioned software update - Hash chains between two pages (P_i and P_{i-1}) of a software update, that was partitioned as described in Figure 6.2, are created as follows: the κ-bit hash of the jth packet of the ith page is appended to the jth packet of the $i-1$th page. This is done for all packets. Additionally, the ρ-bit hash of the ith page is appended to the last packet of the $i-1$th page.

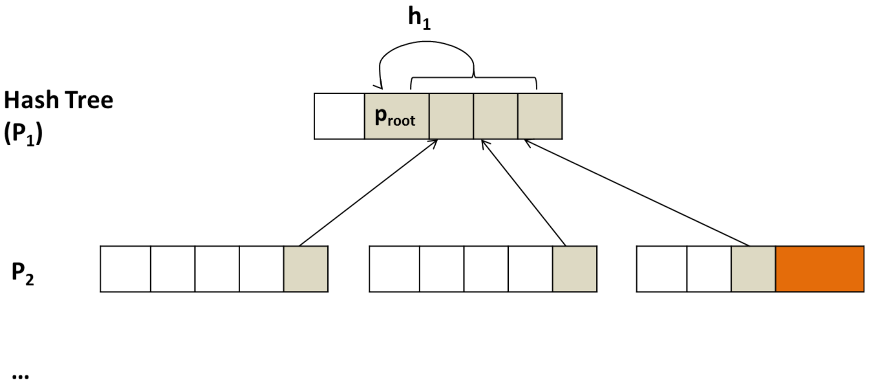


Figure 6.4: Constructing the hash tree on a partitioned software update - Consider a software update that has been processed by adding hash chains as described in Figure 6.3 above. Each hash tree packet at level i is formed by adding the κ-bit hash values of w packets at level $i-1$. In this example $w = 5$. Hence, the entire hash tree can be presented in a single packet.

of the packets of P_2 are set as the leaves of the hash tree. Let us assume that the leaves are located at the ith level of the tree. Each packet at the $i-1$th level of the hash tree is formed by computing the κ-bit hash value of w packets at level i. Here, w is the number of hash values which can fit a single packet. This is done recursively up to the root packet of the hash tree.

Example 6 *Consider a software update that has been processed by adding hash chains as described in Figure 6.3 above. Figure 6.4 shows the construction of the hash-tree page using P_2 for $w = 5$.*

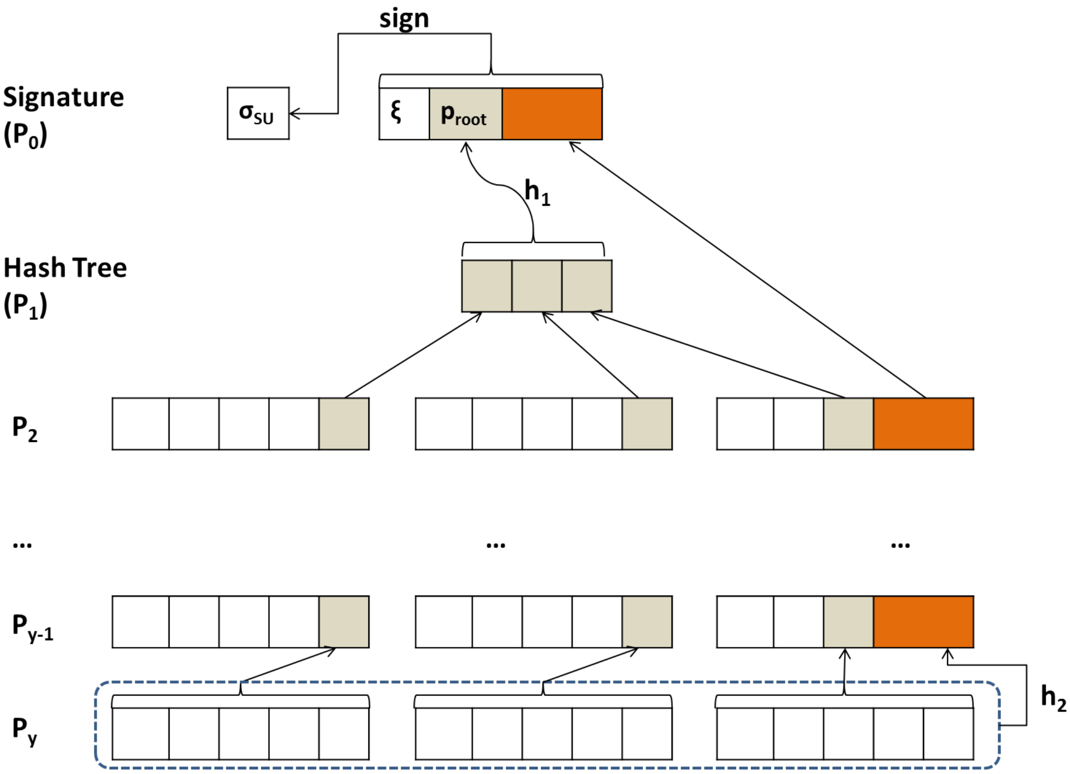


Figure 6.5: Graphical illustration of the software update authentication - Firstly, starting from the last page P_y, the hash (authenticator) chains between the pages are created. Later, the hash tree page P_1 is formed. Finally, the signature page P_0 is constructed. Notice that the last page P_y does not contain any authenticator.

Signing the software update Let $p_{adv} = (p_{root}, h_2(P_2), \xi)$ be the advertisement message for the software update, where p_{root} is the root of the hash tree, $h_2(P_2)$ is the κ-

bit hash value of the first data page following the hash tree, ξ is some update information that contains e.g. the version number of the new software. p_{adv} is signed with the $\mathcal{T}$-time signature scheme. The signature σ_{SU} together with the advertisement packet p_{adv} constitute the signature page P_0. The authenticated software update $SU_{Auth} = P_0, \ldots, P_y$ is disseminated page-by-page where the transmission starts with P_0 and ends with P_y. Figure 6.5 illustrates the entire authentication process of a software update graphically.

6.2.3 Verification

After receiving the signature σ_{SU} and the advertisement packet p_{adv}, the receivers first verify the signature. A valid signature ensures the authenticity of $p_{adv} = (p_{root}, h_2(P_2), \xi)$. Similarly, the packet p_{root} ensures the authenticity of the packets of the hash tree down to the leaves. The leaves of the hash tree are the beginning elements of the hash chains authenticating the remaining pages from P_2 to P_y.

The sensor nodes accept the software update only if the signature is valid and the version number of the update (which is stored in ξ) is greater than the current version of that software. After the signature page, the hash tree is disseminated. The root of the hash tree p_{root} is authenticated by the signature. The root packet is used to verify w packets from the first level of the hash tree. Similarly, each packet $p_{i,j}$ at the ith level verifies w-leaves, $p_{i+1,1}, \ldots, p_{i+1,w}$, at level $i + 1$. This holds recursively until the last level packets (i.e., leaves) of the hash-tree. The leaves of the hash-tree are composed of the hash values of all q packets of P_2, namely, $h_1(p_{2,1}), \ldots, h_1(p_{2,q})$.

Each packet $p_{2,j}$ is verified immediately with the hash value stored in the corresponding leaf of the hash tree which has been received in P_1. Notice that the jth packet of the ith page $p_{i,j}$ contains the hash value of the jth packet of the $i + 1$th page $h_1(p_{i+1,j})$. Hence, starting from the leaves of the hash tree, the authenticity of the packets received in the following pages can be easily verified recursively. Additionally, after a page P_i is received completely, its hash value is compared with the hash value $h_2(P_i)$ received in page P_{i-1}. Once the complete SU_{Auth} is verified successfully, the sensor nodes update their public keys (i.e., their states) according to the $\mathcal{T}$-time signature scheme (see Section 4.4.2).

6.2.4 Remarks and improvements

Improvements and remarks to the basic approach described above are given in the following.

Key refreshment Due to the employed $\mathcal{T}$-time signature scheme, the number of software authentications are limited with $\mathcal{T}$. Hence, $\mathcal{T}$ should be selected sufficiently large to cover the number of secure remote programming (SRP) operations required during the lifetime of the sensor nodes. If the chosen $\mathcal{T}$ was too small, refreshment may become necessary. Let us assume that sensor nodes were initialized for i SRP operations (i.e., $\mathcal{T} = i$). For updating the key pair, the monitoring device first generates a new key pair (pk', sk') each of half-size as described in Section 4.4.3. The new key pair is then disseminated together with the ith software update SU_i. After the SU_{Auth_i} is received and verified successfully, all sensor nodes extract pk' from SU_{Auth_i} and replace the expired public key.

Authenticating the signatures: The packets of the pages $P_1, \ldots, P_y$ can be verified on-the-fly. However, the packets of a signature cannot be verified on-the-fly. All packets of a signature need to be received before its verification. Re-transmission of the signature is required if the received signature is invalid. The adversary might exploit this feature to deplete the sensor nodes' power by sending multiple invalid signature messages. The signature page needs to be authenticated using a broadcast authentication scheme such as the CGIMNP scheme [127] (see Section 5.3.1.2.1) to mitigate such DoS attacks. This allows sensor nodes to detect the corrupted packets. Hence, the sensor nodes require the retransmission of the corrupted packet only instead of the whole signature page P_0.

The signature page is authenticated using the CGIMNP scheme as follows. The monitoring device generates a random key pool K_s. The entire key pool is known only to the monitoring device. All sensor nodes are given a set of ϑ keys selected at random from K_s. These operations are done in the pre-deployment phase of the sensor nodes.

Later, after the deployment of the sensor nodes, the signature page of a software update is authenticated as follows. The monitoring device computes an α-bit MAC for each packet of the signature page using the keys of K_s with $\alpha = |K_s|$. Let mac_i denote the α-bit MAC of a packet p_i. The monitoring device transmits all packets together with their MACs. On receiving a packet-mac pair (p_i,mac_i), each sensor node verifies the one-bit MACs at the ϑ respective positions of the mac_i with its ϑ keys. If all ϑ respective positions pass the MAC verification, the received packet p_i is authentic. Otherwise, the received packet is not authentic. The receivers drop such packets and request their retransmissions. Once the entire signature page is received, sensor nodes proceed with its verification. The main reason for choosing the CGIMNP scheme is its scalability for large WSNs as well as it small code-size requirement (see Section 5.3.1.2.1). The size of the key pools does not depend on the number of the sensor nodes. It depends only on the desired level of security and the number of node corruptions.

6.3 Security evaluation

Recall the three basic security requirements on the remote programming:

- Authenticity: Sensor nodes must accept only those software updates that have been delivered by the monitoring device.
- Integrity: Sensor nodes must accept only those software updates that have not been altered during their dissemination. Any modification made on the updates must surely be detected by the receiving sensor nodes.
- DoS resilience: Considered the DoS attacks exploiting the expensive signature verifications and the online packet modifications, the SRP mechanism must limit the effect of the adversary aiming at exhausting sensor nodes' resources (i.e., storage and power)[1].

In the following subsections, the presented approach is analyzed with respect to these security requirements.

6.3.1 Authenticity

Sensor nodes accept a new software update SU only if the signature message σ_{SU} is valid. Hence, the adversary needs to generate a valid signature for its own software to be

[1] Notice that general DoS attacks targeting at the sensor nodes' capability for receiving or sending a message are out of the scope of this work and therefore not considered.

able to fool the sensor nodes with it. Similar to the t-known-message attacks described in Section 4.4.3, the adversary might exploit the signature parameters disclosed in previous t *consecutive* SRP operations to sign a malicious SU. Let us refer to this attack as *t-known-software-update* attack. To perform this attack, the adversary obtains the signatures on t software updates of the monitoring device's choice while preventing the sensor nodes from receiving those t software updates. After that, the adversary tries to forge a signature on a new software update of its own choice which is valid for all sensor nodes which have not received those t consecutive software updates. Obviously, to be able to perform such a t-known-software-update attack, the adversary needs to know the exact starting time of each SRP operation to obtain the signatures and to prevent the sensor nodes from receiving them. For evaluating the security of the $\mathcal{T}$-time signature scheme, two classes of adversaries are considered: *weak* and *strong* adversary (see Section 4.4.3). In order to represent this new situation, the strong adversary is further classified into two subclasses: the *external* strong adversary and the *internal* strong adversary. It is assumed that only the *internal* strong adversary can perform the *t-known-code-image* attacks.

Security against the weak adversary Not considering the known-software-update attacks, after every successful SRP operation, the monitoring device and the sensor nodes update their private and public keys. This means, each new software update is signed by a new key pair. The success probability of forging the proposed $\mathcal{T}$-time SRP mechanism is therefore independent from the value of $\mathcal{T}$. Forging the signature on a malicious software update requires to forge the signature for the advertisement packet p_{adv}. As shown in Section 4.4.3, the probability of this is 2^{-n} which is negligible in n.

Security against the strong adversary Considering the known-code-image attack, the remaining effort to forge a signature after having obtained t consecutive signatures was given in Table 4.2. Therefore, if security against the internal strong adversary is needed, the secure variant of the $\mathcal{T}$-time signature scheme (described in Section 4.4.3) needs to be used. This variant adds an additional transmission overhead by doubling the signature length. The security against the strong adversary in this case is 2^{-n}. It is negligible in n and independent of the number of software updates that the adversary may intercept.

6.3.2 Integrity

Once the signature page P_0 is verified, the remaining problem is to propagate its security to the remaining pages of the software update. This is achieved by combining a hash tree with multiple hash chains throughout all pages of the update $P_1, \ldots, P_y$ (see Figure 6.5). As described in Section 6.2.3, the root packet of the hash tree is verified by the signature. The root packet verifies the packets of the hash-tree at the first level. The packets at the first level verify the packets at the second level. This holds until the leaves of the hash tree. The leaves of the hash tree verify the packets of the page P_2. Similarly, the packets of a page P_i verify the packets of the following page P_{i+1}. As a result, any malicious packet p' received during the dissemination of the hash-tree is detected by the sensor nodes by checking $h_1(p') = h_1(p)$. Similarly, any malicious packet $p'_{i+1,j}$ received during the dissemination of pages $P_2, \ldots, P_y$ is detected by the sensor nodes by checking $h_1(p'_{i+1,j}) = h_1(p_{i+1,j})$. Moreover, once a page P_i is received completely, its hash value is compared with $h_2(P_i)$ received in the previous page P_{i-1}. This additional authenticity check at the page granularity allows to reduce the communication overhead. The reason for this is that ρ-bit security is achieved by appending only a ρ-bit hash value to each page rather than to all packets. Packets are appended only a κ-bit hash value with $\kappa \ll \rho$.

6.3.3 DoS resilience

DoS attacks are considered in two classes: DoS attacks exploiting (online) packet modifications and DoS attacks exploiting fake signatures.

DoS resilience to packet modifications The approaches proposed in [72, 73] suffer from the following problems. [73] allows the sensor nodes to store the packets received in a page without checking their authenticity. They are authenticated (implicitly) by verifying the authenticity of the corresponding page. However, since the sensor nodes cannot identify the faulty packet in case of an unauthenticated page, retransmission of the whole page is required in such a case. The adversary can make use of this vulnerability to mount DoS attacks. Since modifying a single packet makes the sensor nodes drop the entire page received, the adversary depletes sensor nodes' power by modifying packets during the dissemination of a software update. [72] removes this problem by verifying the software updates on a per packet basis. That is, each packet p_i is authenticated by the previous packet p_{i-1}. However, since the ordered packet delivery is not guaranteed by Deluge, the sensor nodes need to store all packets received out-of-order for the later verification. The adversary can make use of this vulnerability to exhaust e.g., the sensor nodes' buffer capacity by transmitting a large number of out-of-order bogus packets.

The approaches presented in this thesis as well as in [75] solve the problems above by using multiple hash chains. Each packet of a page is authenticated with an individual hash chain (see Figure 6.5). This allows sensor nodes to check the authenticity of a received packet p_i (independently from its order) on-the-fly by using the hash value $h_1(p_i)$ received in the ith packet of the previous page. However, this flexibility is slightly limited while verifying the packets of the hash tree page. That is, a packet at level i can be verified on-the-fly only if all packets at the previous level $i-1$ have been received.

DoS resilience to fake signature messages As described in Section 6.2.4, the signature page is authenticated with the CGIMNP scheme [127]. The security of the CGIMNP scheme depends on the number of the compromised nodes λ and the number of keys ϑ known to each sensor node. The strong adversary can compromise the sensor nodes whereas the weak adversary cannot. Hence, the security of the presented approach is analyzed for each class of adversary separately:

Security against the weak adversary: Each sensor node verifies ϑ bits of the received α-bit MAC using its ϑ keys. A single bit MAC is 2^{-1}-per-packet forgeable. Hence, the probability of breaking a MAC without knowing the ϑ keys is $q = 2^{-\vartheta}$. Moreover, this probability is independent of K_s. Hence, bootstrapping the sensor nodes with ϑ keys (i.e., $|K_s| = \vartheta$) provides ϑ-bit level of per-packet security against the weak adversary.

Security against the strong adversary: Let us assume that the adversary can corrupt up to λ sensor nodes. Then, choosing the security parameters as $\alpha = |K_s| = (\lambda + 1) \cdot \vartheta$ and $\vartheta = 4e \cdot \ln(2^q)$ provides q-bit per-packet security.

6.4 Overhead analysis

The presented approach is analyzed in terms of the communication, storage, and computation overheads. The notation used in the remainder of this chapter is summarized

Table 6.1: The notation.

Notation	Description
$\lvert SU\rvert$	size of the plain software update in bytes
$\lvert payld\rvert$	payload size of a packet in bytes
$\lvert page\rvert$	page size in bytes
$\lvert\sigma_{SU}\rvert$	size of the $\mathcal{T}$-time signature in bytes
$\lvert pk\rvert$	size of the public key in bytes
p	number of packets per page
z	parameter for trading the signature size with verification time for the proposed $\mathcal{T}$-time signature scheme (see Section 4.4.4)
n	bytes level of security against malicious software updates
q	bytes level of security against per-packet forgery attacks for the signature page P_0 when up to λ sensor nodes are compromised
κ	bytes level of security against per-packet forgery attacks for pages $P_2 \ldots, P_y$
ρ	bytes level of security against per-page forgery attacks for pages $P_2 \ldots, P_y$

in Table 6.1.

6.4.1 Communication overhead

Let $SU_{Auth} = P_0||P_1|| \ldots P_{y-1}||P_y$ denote an authenticated software update. The communication overhead consists of the overheads for the pages $P_2, \ldots, P_y$, the hash tree page P_1, and the signature page P_0. Notice that $\kappa \leq \rho < |payld|$.

Code pages: No hash value needs to be appended to the last page. Hence, it is omitted. Each page P_i is composed of p packets and carries $(p \cdot \kappa + \rho)$ bytes hash value in total. Thus, the total communication overhead for pages P_2 to P_{y-1} is

$$\varrho \cdot (p \cdot \kappa + \rho) \tag{6.1}$$

bytes. Here,

$$\varrho = \left\lceil \frac{|SU| - p \cdot |payld|}{p \cdot (|payld| - \kappa) - \rho} \right\rceil \tag{6.2}$$

denotes the number of the data pages excluding the last one.

Hash tree page: The hash tree page P_1 is composed of the packets of all hash-tree levels except the root packet p_{root}. Each leaf packet carries the hash values of $c = \lfloor \frac{|payld|}{\kappa} \rfloor$ packets of P_2. Hence, the hash-tree has $ht_l = \lceil \frac{p}{c} \rceil$ leaves. The total communication overhead of the hash tree is therefore at most

$$|payld| \cdot \sum_{i=1}^{\lceil log_2(ht_l) \rceil} 2^i \tag{6.3}$$

bytes.

Signature page: The signature page P_0 is composed of the quadruple (p_{root}, $h_2(P_2)$, σ_{SU}, ξ). ξ remains the same as in the standard Deluge. Hence, it is not part of the security overhead and is omitted in the analysis. The size of a $\mathcal{T}$-time signature on average is

$$|\sigma_{SU}| = \frac{l \cdot n}{2z} \tag{6.4}$$

bytes, where $l = (8n + \lfloor log_2(8n) \rfloor + 1)$ and z is the parameter for trading the signature size with the verification time (see Section 4.4.4). Due to the authentication with the CGIMNP scheme, each packet of the signature page leads to an additional overhead of

$$|mac| = \left\lceil \frac{4e \cdot (\lambda + 1) \cdot \ln(2^{8q})}{8} \right\rceil \tag{6.5}$$

bytes.

The total communication overhead of the signature page is therefore

$$|\mu| + |mac| \cdot \left\lceil \frac{|\mu|}{|payld| - |mac|} \right\rceil \tag{6.6}$$

bytes, where $|\mu| = (|\sigma_{SU}| + \rho + |payld|) = (|\sigma_{SU}| + |h_2(P_2)| + |p_{root}|)$.

In summary, the total communication overhead for authenticating a software update SU is

$$\begin{aligned} Comm_{ohead} &= (|\mu| + |mac| \cdot \left\lceil \frac{|\mu|}{|payld| - |mac|} \right\rceil \\ &+ (|payld| \cdot \sum_{i=1}^{\lceil log_2(ht_l) \rceil} 2^i) \\ &+ \varrho \cdot (p \cdot \kappa + \rho)) \end{aligned}$$

bytes, where $l = (8n + \lfloor log_2(8n) \rfloor + 1)$, $|\sigma_{SU}| = \frac{l \cdot n}{2z}$, $c = \lfloor \frac{|payld|}{\kappa} \rfloor$, $ht_l = \lceil \frac{p}{c} \rceil$, $|\mu| = (|\sigma_{SU}| + \rho + |payld|)$, $\varrho = \lceil \frac{|SU| - p \cdot |payld|}{p \cdot (|payld| - \kappa) - \rho} \rceil$, and $|mac| = \lceil \frac{4e \cdot (\lambda+1) \cdot \ln(2^{8q})}{8} \rceil$.

6.4.2 Storage overhead

The storage overhead is caused by the public key pk required for verifying the signature messages, the ϑ keys required for verifying the MACs authenticating the packets of P_0 and the hash values required for verifying the packets of the code pages.

The size of the public key of the $\mathcal{T}$-time signature is

$$|pk| = (l \cdot n) \tag{6.7}$$

bytes, where $l = (8n + \lfloor log_2(8n) \rfloor + 1)$. The size of the ϑ MAC verification keys is

$$|\vartheta| = n \cdot 4e \cdot \ln(2^{8q}) \tag{6.8}$$

bytes. Finally, each node needs to have a buffer b of size

$$max(|(\sigma_{SU}, p_{root}, h_2(P_2), \xi)|, \varrho \cdot (p \cdot \kappa + \rho), |\vartheta|) \tag{6.9}$$

bytes to be able to store the signature message and the hash values needed to verify the code pages.

The buffer b is allocated on the data memory (RAM). The public key and the authentication keys need to be stored persistently. Hence, they are stored on the non-volatile EEPROM memory. EEPROM makes the key material available even after a battery change or a reset. However, access to EEPROM is more energy consuming than RAM. The same MAC keys need to be accessed several times during the verification of the packets of the signature page P_0. Therefore, keeping the ϑ authentication keys in buffer b during the reception of P_0 reduces the energy consumption. Only one signature verification is needed in each secure remote programming operation which is in general performed rarely. Hence, the public key is kept in EEPROM to save valuable RAM space.

6.4.3 Computation overhead

The verification of a $\mathcal{T}$-time signature requires on average $\frac{l}{2} + 1$ (fixed-length) hash computations (see Section 4.4.4). Additionally, ϑ MAC evaluations are required to check the authenticity of the packets of the signature page P_0.

The transmission of the hash-tree page P_1 requires $\sum_{i=0}^{\lceil log_2(ht_l) \rceil} 2^i$ hash computations. Finally, verifying each data page from P_2 to $P_{\varrho-1}$ requires $p + 1$ hash computations.

In summary, the computation overhead for authenticating a software update is

$$(\frac{l}{2}+1)+(\sum_{i=0}^{\lceil log_2(ht_l)\rceil} 2^i)+\varrho\cdot(p+1) \tag{6.10}$$

hash computations and

$$4e\cdot\ln(2^{8q})\cdot\left\lceil\frac{|\mu|}{|payld|-|mac|}\right\rceil \tag{6.11}$$

MAC computations, where $l = (8n+\lfloor log_2(8n)\rfloor+1)$, $|\sigma_{SU}| = \frac{l\cdot n}{2z}$, $c = \lfloor\frac{|payld|}{\kappa}\rfloor$, $ht_l = \lceil\frac{p}{c}\rceil$, $|\mu| = (|\sigma_{SU}| + \rho + |payld|)$, $\varrho = \lceil\frac{|SU|-p\cdot|payld|}{p\cdot(|payld|-\kappa)-\rho}\rceil$, and $|mac| = \lceil\frac{4e\cdot(\lambda+1)\cdot\ln(2^{8q})}{8}\rceil$.

6.5 Choice of security parameters

A higher level of security results in higher overhead. Hence, the general design principle is to achieve the sufficient level of security with the smallest possible overhead.

There are five security parameters to determine in the presented approach: the number of the software authentications allowed with one key pair ($\mathcal{T}$), the output size of the hash functions h_1 and h_2 (i.e. κ and ρ), the frequency of the key refreshment, and finally the parameters α and ϑ of the CGIMNP scheme.

For n-bit security, the output length of the hash function used in the $\mathcal{T}$-time signature as well as the output length of h_2 must be n bits[1].

$\mathcal{T}$ should be large enough to cover the number of secure remote programming (SRP) operations required during the lifetime of the sensor nodes. No key refreshment is required when the weak adversary (WA) or external strong adversary (ESA) is considered. However, in case of the internal strong adversary (ISA), keys would need to be refreshed in each SRP operation as described in Section 4.4.3.

The security parameters κ and ρ (i.e., the output size of the hash functions h_1 and h_2) determine the level of protection against the packet modifications and malicious software insertion attacks. Since the packets have to be modified on-line during the dissemination of a software update, 32-bit security is reasonable against packet modifications. Therefore, the outputs of h_1 are truncated into 32 bits ($\kappa = 32$). The security against the malicious software update insertion attacks should be the same as

[1] Both hash functions need to be preimage- and second-preimage resistant. Choosing n-bit outputs for both functions provides n bits security.

Table 6.2: Proposed security parameters.

$\mathcal{T}$-time signature				
Sec.	WA or ESA		ISA	
level	$\lvert f\rvert = \lvert h\rvert$	$\mathcal{T}$	$\lvert f\rvert = \lvert h\rvert$	$\mathcal{T}$
2^{-128}	128	> 30	128	> 30 (Sec. 4.4.3)
2^{-80}	80	> 30	80	
2^{-64}	64	> 30	64	
2^{-32}	32	> 30	32	
Broadcast authentication for the signature page P_0				
	WA		ESA or ISA	
$\lambda = 1$	α	ϑ	α	ϑ
2^{-9}	9	9	136	68
2^{-10}	10	10	150	75
$\lambda = 2$				
2^{-9}	9	9	204	68
2^{-10}	10	10	225	75

the security of the $\mathcal{T}$-time signature. Thus, the outputs of h_2 are truncated into n bits ($\rho = n$).

Finally, the security against the DoS attacks on the signature page P_0 depends on the per-packet security of the CGIMNP scheme. 10-bit level of security against DoS attacks is reasonable. Choosing $\vartheta = 76$ and $\alpha = |K_s| = \vartheta \cdot (\lambda + 1)$ provides 2^{-10} security against the strong adversary (internal or external) compromising up to λ sensor nodes. Choosing $\alpha = \vartheta = 10$ provides the same level of security against the weak adversary which cannot compromise the sensor nodes. Table 6.2 shows the proposed security parameters to be used with the presented approach to achieve a desired level of security against WA, ESA, and ISA.

6.6 Prototype implementation

The prototype implementation requires a fixed-length hash function f (for the $\mathcal{T}$-time signature), a second pre-image resistant hash function h (for the hash chains), a keyed hash function (MAC) (for the CGIMNP scheme). A preimage resistant hash function h can be used as the fixed-length hash function f. MACs can be computed with the

Table 6.3: Memory footprints for the secure remote programming using the $\mathcal{T}$-time signature scheme.

Module	ROM	RAM
SHA1	2048B	108B
HMAC-SHA1	2392B	116B
Signature verification (with $\mathcal{T}$-time signature)	2384B	1932B
Broadcast authentication (with CGIMNP scheme)	2528B	201B

HMAC algorithm using h (see Section 4.2.2). For the prototype implementation, the hash function SHA1 was used.

Table 6.3 shows the ROM and RAM usage of the security modules on a TelosB sensor platform. All implementations were done in nesC language for TinyOS-2.X [78, 145]. All the security primitives are built from the hash function SHA1. We believe that the memory footprints can be further reduced by using a block cipher as the underlying building block for the implementations of the required primitives as described in Chapter 4.

6.7 Comparison with the previous approaches

Only Seluge [75] provides a comparable security level against DoS attacks exploiting invalid signature messages and online packet modifications from the previous approaches [72, 73, 74, 75]. Therefore, the efficiency of the presented approach is compared with Seluge. Comparisons are done in terms of the code size (ROM consumption) as well as the communication, storage, and computation overheads[1].

Comparisons are performed with the following security parameters: 80-bit security for the malicious software update protection ($n = \rho = 80$), 32-bit security for online packet modifications ($\kappa = 32$), and 10-bit security for the multicast authentication (see Section 6.5). Furthermore, 80-bit symmetric keys are assumed to be equivalent to 160-bit elliptic curve keys (see Table 3.4).

[1] Deluge's negative acknowledgment packets are not considered in the analyses.

Table 6.4: ROM and RAM consumptions required for securing Deluge.

Work	ROM	RAM
Presented approach based on $\mathcal{T}$-time signature	480B	1824B
Seluge [75] based on ECDSA	13520B	1396B

6.7.1 Code size

The presented approach uses the stateful-verifier $\mathcal{T}$-time signature while Seluge exploits the ECDSA signature scheme implemented in TinyECC [69]. Both schemes use SHA1 as hash function for verifying the integrity of the received software update packets. The code size for SHA1 is not presented in [75]. Hence, the code size from our prototype implementation is used as the reference value.

The presented approach occupies 2392B ROM and 1932B RAM on the TelosB platform. Compiling Seluge on the same platform occupies 15568B ROM and 1504B RAM. Excluding the RAM and ROM consumption of SHA1 from the both approaches gives the actual ROM and RAM consumptions required for securing Deluge. Results are shown in Table 6.4. Our SRP mechanism occupies only 1% of the available ROM (48KB) and 18.24% of the available RAM (10KB) on a TelosB sensor mote. In contrast, Seluge occupies 28.1% of the available ROM and 13.96% of the available RAM for securing Deluge on the same platform.

6.7.2 Communication overhead

There are two main differences between the presented approach and Seluge which affect the communication overhead. Firstly, the signature scheme used in Seluge is ECDSA. Secondly, the presented approach uses two hash functions (h_1 and h_2) of different output lengths while Seluge uses only one hash function. Let h_S denote the hash function used in Seluge and $|h_S(\cdot)|$ its output size.

The length of an ECDSA signature with 160-bit curve parameters is 40B. Hence, the total communication overhead in Seluge is at most

$$40 + (|payld| \cdot \sum_{i=1}^{\lceil log_2(ht_l) \rceil} 2^i) + \varrho \cdot p \cdot |h_S(\cdot)| \tag{6.12}$$

bytes, where $c = \lfloor \frac{|payld|}{|h_S(\cdot)|} \rfloor$, $ht_l = \lceil \frac{p}{c} \rceil$, and $\varrho = \lceil \frac{|CI| - p \cdot |payld|}{p \cdot (|payld| - |h_S(\cdot)|)} \rceil$.

Figure 6.6 shows the communication overheads of both approaches for different software update sizes with the following parameters: $p = 17$, $|payld| = 64$, $|h_S(\cdot)| = 10$, $z = 1$, and $\lambda = 2$. Figure 6.6 implies that the efficiency of both approaches decreases linearly with the size of the software update (SU). However, the coefficient for Seluge is larger. The presented approach is more efficient than Seluge even in case of the ISA, when a new public key needs to be transmitted in each SRP operation. Notice that the communication overhead of the presented approach can be further reduced by choosing $z > 1$ as described in Section 4.4.4.

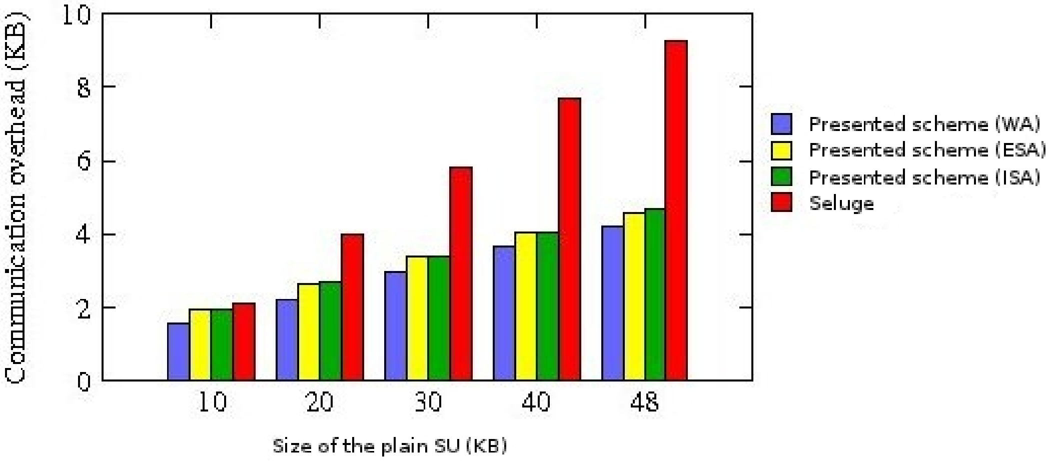


Figure 6.6: Communication overhead - Communication overheads for the presented approach and Seluge [75] for 80-bit security in case of the weak, external strong and internal strong adversaries.

6.7.3 Computation overhead

Verifying a ECDSA signature on the TelosB platform takes 10290ms [69]. One SHA1 computation takes 7.6ms in the reference implementation. The computation overhead for Seluge is therefore

$$\overbrace{1353 \cdot t_{hashvrfy}}^{t_{sigvrfy}} + (t_{hashvrfy} \cdot \sum_{i=1}^{\lceil log_2 ht_l \rceil} 2^i) + \varrho \cdot p \cdot t_{hashvrfy} \tag{6.13}$$

ms in total. Here, $t_{sigvrfy}$ denotes the ECDSA signature verification time and $t_{hashvrfy}$ denotes the execution time for the SHA1 (in ms).

One MAC computation (HMAC-SHA1) takes 29ms in the reference implementation. Hence, one MAC computation is equivalent to 3.8 SHA1 computations. The computation overhead for the presented approach is then computed using the formulas given in Section 6.4.3.

Figure 6.7 shows the computation overheads for both approaches for different software update sizes with the following parameters: $p = 17$, $|payld| = 64$, $|h_S(\cdot)| = 10$, $z = 1$, and $\lambda = 2$. The results show that the presented scheme is (on average) 10s faster than Seluge.

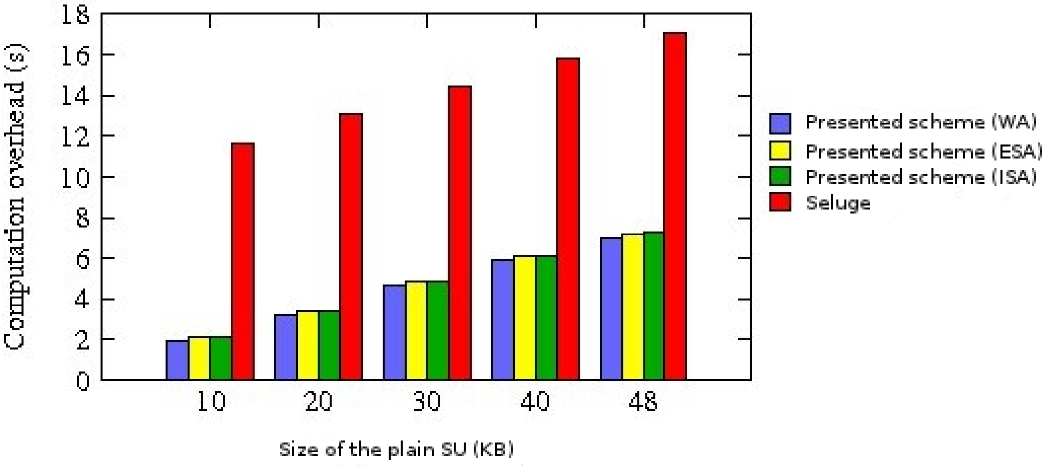


Figure 6.7: Computation overhead - Computation overheads for the presented approach and Seluge [75] for 80-bit security in case of the weak, external strong and internal strong adversaries.

6.7.4 Storage overhead

In the presented approach, one public key and ϑ MAC verification keys need to be stored in EEPROM. The maximum EEPROM consumption for the presented approach is therefore 1.63KB for $\lambda = 2$. Seluge occupies 40B of EEPROM memory for storing the public key.

6.8 Conclusion

This chapter presents a code-size efficient approach for authenticating software updates. It is particularly designed for sensor platforms with a low amount of flash memory (ROM). The presented approach occupies only 480B of ROM. Such a tiny memory requirement allows to implement the remote programming functionality on the sensor platforms while still maintaining space for other applications' program code. The code size requirement of the presented approach is a tremendous improvement compared to the approaches using elliptic curve based signature schemes. For example, ECDSA implementation occupies 13KB code space on the same platform. Such a great improvement is mainly possible due to the stateful-verifier $\mathcal{T}$-time signature scheme presented in Section 4.4. It is a signature scheme that is well suited for situation, such as remote programming, where one signer transmits messages sequentially to a closed group of verifiers.

7

Authenticating Software Updates encoded with Fountain Codes

Software updates need to be broadcast over wireless channels which are prone to packet losses. Several works [11, 52, 53, 54, 55] have proposed to use rateless erasure codes [49, 50, 51] for ensuring an efficient and reliable software dissemination even in highly unreliable environments (see Section 1.2.2.2.1). The main idea is to reconstruct the missing packets from the already received ones or other arriving packets. This is possible if the packets are encoded with rateless erasure codes. However, this great feature comes not for free. Rateless erasure codes introduce new security problems, too. In particular, DoS attacks become more powerful due to the natural error-propagating property of the rateless erasure codes.

This chapter presents efficient and code-size optimized security enhancements for LT codes [49] which are the first practical implementation of the rateless erasure codes. The presented approach was originally proposed in [4, 146]. It is an extension of [3] (see Chapter 6) with LT codes support. In addition to the authenticity protection, the proposed security enhancements reduce the impact of the poisoning attacks (DoS attacks) on the LT codes by allowing (nearly) on-the-fly packet verifications.

7.1 Preliminaries and assumptions

The assumptions as well as the security and network models are the same as those ones described in Section 6.1. The only difference is that the software updates are encoded with security enhanced LT codes. Hence, the underlying remote programming protocol needs to implement the LT-encoder and the LT-decoder algorithms. The security enhanced LT-encoder is given in Algorithm 6. The security enhanced LT-decoder is given in Algorithm 7. Both algorithms are explained in the respective sections.

The following Section gives a short overview of the LT codes which is necessary to understand the proposed security enhancements.

7.2 LT codes

Motivation LT codes are the first practical realization of the rateless erasure codes [49]. They are also known as Fountain codes. LT codes are rateless since the number of encoded packets that can be constructed from a set of source packets is potentially infinite. The receivers can reconstruct the source packets from any subset of the encoded packets that is slightly larger than the number of source packets. Hence, there is no need to inform the sender about missing packets to request their retransmission. This is very beneficial in networks with high packet-loss rates such as WSNs. Moreover, packets can be encoded and decoded very efficient. Both operations require only a small number of XOR operations close to the minimal possible (i.e, number of the source packets). LT codes are also very efficient in terms of the code size as well as the storage requirements. The code size requirement is nearly zero since it requires only XOR operations. Recovering the source packets requires only a slightly larger number of encoded packets than the number of source packets. Hence, the storage requirement is slightly larger than a setting without LT codes. All these advantages make the LT codes an attractive tool for improving the efficiency of the remote programming operations in unreliable WSNs.

The encoding process Let $S = \{p_1, \ldots, p_y\}$ denote the set of source packets to be encoded and $S' = \{p'_1, \ldots\}$ the set of encoded packets. Notice that the set S' is potentially infinite. An encoded packet is generated by choosing d random packets and subsequently by XORing them. d is referred to as the degree of the encoded packet. Each of those d source packets is referred to as a neighbor. That is, an encoded packet of degree d has d neighbors. The degrees are chosen from a degree distribution $\Omega(d)$ uniformly at random.

The encoder (sender) transmits each encoded packet p'_i together with its degree d_i as well as the indices of its neighbors to the receivers (decoders)[1]. Hence, the indices of their neighbors are appended as prefix to the encoded packets. That is,

$$p'_i = (c_i || \bigoplus_{j=1}^{d_i} p_j), \tag{7.1}$$

where N is the set of neighbors, c_i is a coefficient string whose jth bit is set for each neighbor $p_j \in N$. Hence, the length of encoded packets is $|p'| = |p| + y$ bits, where y is the number of source packets. The encoding process is summarized in Algorithm 4.

```
Algorithm 4 LT-encoder
Require: The set of source packets $S = \{p_1, \ldots, p_y\}$ and the degree distribution $\Omega(\cdot)$
Ensure: Encoded packets $p'_i$
1: choose a random degree $d_i$ from the degree distribution $\Omega(d_i)$
2: choose a set of random neighbors $N$ composed of $d_i$ elements (i.e., $N \subseteq S$)
3: $p'_i \leftarrow (c_i || \bigoplus_{j=1}^{d_i} p_j)$, where $c_i$ is the coefficient string whose $j$th bit is set for each
   neighbor $p_j \in N$ (i.e., $j$ is the index of neighbors).
4: return $p'_i$
```

Example 7 *Consider a set of source packets $S = \{p_0, p_1, p_2, p_3\}$ as depicted in Figure 7.1. An encoded packet p'_i is generated as follows: Firstly, a degree $d_i \in \{1, 2, 3, 4\}$ is chosen randomly. Suppose that $d_i = 2$. Subsequently, two distinct packets (i.e., neighbors) are chosen from the set S uniformly at random. Suppose that those are $\{p_0, p_3\}$. Finally, the encoded packet p'_i is computed by XORing all neighbors and appending a coefficient string to the result value as prefix, $p'_i = (1001||(p_0 \oplus p_3))$.*

The decoding process The decoding process is performed merely using XOR operations. Recall that an encoded packet p' of degree d is the XOR of d (distinct) neighbors $N = \{p_1, \ldots, p_d\}$. Hence, XORing p' with one of its neighbor $p_j \in N$ yields a new packet of degree $d - 1$. Repeating this process $d - 1$ times yields the last source packet. The is the main idea behind the decoding process. Notice that the decoder needs to know the neighbors of the encoded packets to be able to decode them. As described in the previous subsection, this information is provided with a coefficient string.

[1] Alternatively, the degree and the indices of the neighbors of an encoded packet can be computed by the receiver using a pseudorandom generator which is seeded with the same value both at the sender and the receivers [49]. However, this method requires to synchronize the seeds.

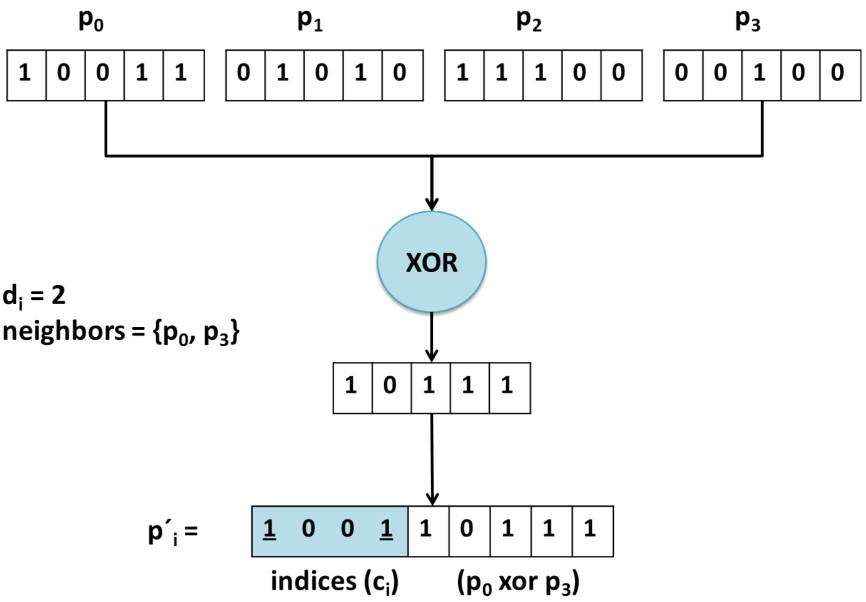


Figure 7.1: Graphical illustration of the encoding process - Given $S = \{p_0, \ldots, p_3\}$, $d_i = 2$ and $N = \{p_0, p_3\}$. Then, the resulting encoded packet is $p'_i = (c_i || (p_0 \oplus p_3)) = (1001||10111)$.

A possible approach to implement the LT-decoder is as follows: two buffers A and B are allocated. The decoder stores the packets of degree $d = 1$ (i.e. source packets) in buffer A. Other packets of degree $d > 1$ (i.e., encoded packets) are stored in buffer B.

On receiving a packet[1], the decoder first checks its degree by computing the Hamming weight of its coefficient string. Let $D(p'_i)$ denote the degree of a packet p'_i. An arriving packet p'_i is a source packet if $D(p'_i) = 1$. In such a case, p'_i is XORed with all of those packets having p'_i as a neighbor and residing in buffer B. Subsequently, all packets, whose degrees were reduced to one, are XORed with the other packets of buffer A. This is done iteratively until no degree one packet is remained. Finally, p'_i is stored in buffer A and all other degree one packets are moved from buffer B into the buffer A. If the degree of the received packet $D(p'_i) > 1$, it is XORed with all of those packets which are a neighbor of p'_i and residing in buffer A. p'_i is stored in buffer B if its reduced degree is still larger than one. Otherwise, it is used to decode the packets of buffer B as described above and finally stored in buffer A. The decoding process is

[1] The decoder drops all duplicate packets $p'_i \in A \cup B$.

summarized in Algorithm 5. The decoder continues to listen for new packets until all source packets are recovered.

7.3 Security challenges

LT codes enable to distribute software updates even in highly unreliable WSNs efficiently. However, they introduce new security challenges, too. The security challenges can be collected in two classes: error propagation and crippled decoding.

7.3.1 Error propagation attacks (poisoning attacks)

When a packet of degree one is received, it is XORed with all of those packets of buffer B one of whose neighbor is the received packet (see step 7 of Algorithm 5). The degree of the XORed packets are then reduced by one. Hence, the decoding process is faced with an important security problem, error propagation[1]: changing even a single bit of an arriving packet of degree one changes the same bit in all of its neighbors residing in buffer B. This allows the adversaries to perform so called poisoning attacks. The adversary needs to tamper only with a few packets of degree one to poison the entire software update. Figure 7.2 illustrates the principle of a poisoning attack.

Notice that the poisoning attack becomes even more dangerous when the receivers (i.e. the sensor nodes) generate new encoded packets as well. This is the case in multihop WSNs. The receivers at each hop encode the packets for the receivers at the next hop. Packets are encoded simply by XORing them. Hence, an error (e.g., introduced by the adversary) in a single packet might propagate to the entire software update. This allows the adversary to stop the entire remote programming operation by modifying a single packet. The main goal of the adversary performing a poisoning attack is to violate the integrity, hence the authenticity, of the software updates.

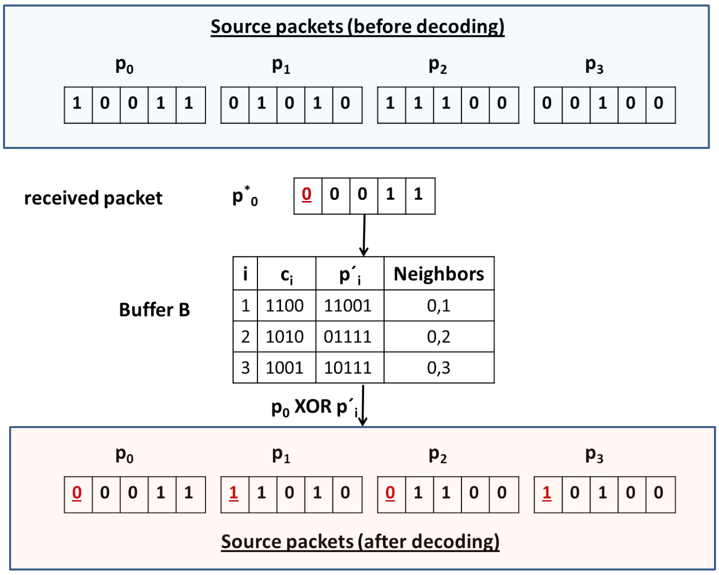
7.3.2 Crippled decoding (coefficient vector attacks)

The decoding process is completed when the buffer A contains all source packets. Hence, the decoding process can be crippled by preventing moving packets from buffer B to A.

[1] Errors might occur due to the unreliable wireless communication (e.g., a bit is flipped from 1 to 0) or they might be introduced by the adversaries (e.g., by tampering with the packets). Checksums might be used to detect errors occurring naturally. However, they cannot be used to detect errors introduced by the adversary. The reason is that no secret key is required for computing checksums.

Algorithm 5 LT-decoder

```
Require: An arriving packet $p'_i$, two buffers $A$ and $B$ and a temporary buffer $Temp = \emptyset$
Ensure: Updated buffers $A$ and $B$
 1: if $p'_i \in A \cup B$ then
 2:   {/* drop duplicate packet */}
 3:   drop $p'_i$
 4:   return $A$, $B$
 5: end if
 6: if $D(p'_i) == 1$ then
 7:   for all $p_j$ in $B$ having $p'_i$ as a neighbor do
 8:     $p_j \leftarrow p_j \oplus p'_i$
 9:     if $D(p_j) == 1$ then
10:       {/* move $p_j$ from buffer $B$ into buffer $Temp$ */}
11:       $B \leftarrow B \setminus \{p_j\}$
12:       $Temp \leftarrow Temp \cup \{p_j\}$
13:     end if
14:   end for
15:   {/* store $p'_i$ in buffer $A$ */}
16:   $A \leftarrow A \cup \{p'_i\}$
17:   if $Temp \neq \emptyset$ then
18:     {/* use the new degree-one packtes for further decoding */}
19:     choose a packet $p'_i \leftarrow p_j \in Temp$
20:     $Temp \leftarrow Temp \setminus \{p_j\}$
21:     jump to step 7
22:   end if
23: else
24:   for all $p_j$ in $A$ that are a neighbor of $p'_i$ do
25:     $p'_i \leftarrow p_j \oplus p'_i$
26:   end for
27:   if $D(p'_i) == 1$ then
28:     jump to step 7
29:   else
30:     {/* store $p'_i$ in buffer $B$ */}
31:     $B \leftarrow B \cup \{p'_i\}$
32:   end if
33: end if
34: return $A$, $B$
```



i	c_i	p'_i	Neighbors
1	1100	11001	0,1
2	1010	01111	0,2
3	1001	10111	0,3

Figure 7.2: Poisoning attack - Consider the source packets given in Figure 7.1. Furthermore, suppose that the current state of buffer B is $B = \{p_0 \oplus p_1, p_0 \oplus p_2, p_0 \oplus p_3\}$. The adversary can poison the entire source packets by modifying a single bit of p_0. Flipping the first bit of p_0 flips the first bit of all decoded packets.

A packet is moved from buffer B to buffer A only if its degree is one after decoding (see step 21 and 28 of Algorithm 5) or if it was received in plain (see step 16 of Algorithm 5). Hence, the adversary can put the decoding process in an infinite loop by blocking the dissemination of the packets of degree one.

A simple way to implement such an attack is to set the degree of a large number of packets being transmitted to a value greater than one, $d > 1$. The degree as well as the indices of the neighbors of a packet are carried in the coefficient string. Hence, such an attack is referred to as coefficient vector attack [4, 146]. The main goal of the adversary is to delay the decoding process as well as to exhaust the capacity of buffer B. Hence, a coefficient vector attack is in fact a DoS attack.

7.4 Security enhancements for remote programming

This section describes the security enhancements of LT codes. The security goals are to mitigate the poisoning and the coefficient vector attacks described in the previous section. More precisely, the following goals are to achieve:

- Authenticity protection: It must be ensured that the LT-decoder stores a packet in buffer A only if it is a source packet. In other words, at the end of the decoding process, the buffer A contains only those packets that have been sent by the monitoring device (i.e., the software update delivered by the monitoring device).

- Integrity protection: The LT-decoder never stores a packet in buffer A if it has been altered. In other words, at the end of the decoding process, the buffer A contains only those packets that are integer.

- DoS protection: The LT-decoder minimizes the impact of the coefficient vector attacks. In other words, the decoder recovers the entire software update when it receives a sufficient number of encoded packets. That is, the decoding process is prevented from being crippled even if the adversary modifies the coefficient vector of some packets.

The authenticity protection implies the integrity protection. It is required to mitigate the poisoning attacks. Verifying the packets (nearly) on-the-fly mitigates the DoS attacks. The terms on-the-fly verification as well as nearly on-the-fly verification are defined as follows [4]:

Definition 3 (On-the-fly verification) *An encoded packet is referred to as on-the-fly verifiable if it can be verified immediately on its arrival at the decoder.*

Definition 4 (Nearly on-the-fly verification) *An encoded packet is referred to as nearly on-the-fly verifiable if it can be verified only after being stored in buffer B for a certain (short) time.*

7.4.1 Mitigating poisoning attacks

The encoding as well as the decoding processes need to be secured to mitigate poisoning attacks. The basic idea behind the secure encoding is to encode the source packets (i.e., the software update) after authentication. Similarly, the basic idea behind the secure decoding is to label a packet as decoded only if it is authentic. The security enhanced encoding and decoding processes are implemented as follows:

Security enhanced encoding The secure encoding process is very similar to the original encoding process described in Section 7.2. The only difference is that the set of source packets is not the packets of the plain software update, but that after authentication.

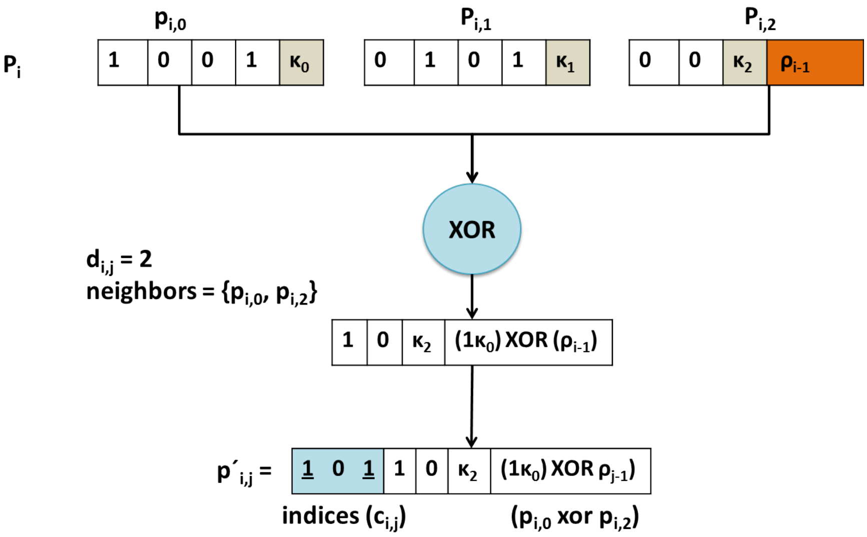


Figure 7.3: Graphical illustration of the secure encoding process - $S = P_i = \{p_{i,0}, p_{i,1}, p_{i,2}\}$ (given in Example 5), $d_{i,j} = 2$ and $N = \{p_{i,0}, p_{i,2}\}$. Then, the resulting encoded packet is $p'_{i,j} = (c_{i,j}||(p_{i,0} \oplus p_{i,2})) = (101||10||\kappa_2||(1||\kappa_0) \oplus (\rho_{i-1}))$.

Suppose that the software update to be encoded is SU. The secure encoding process is executed as follows. Firstly, SU is authenticated as described in Section 6.2 using Ugus et al.'s approach [3]. Let $SU_{Auth} = P_0||P_1||P_2||\dots||P_y$ denote the authenticated software update. Then, the packets of the signature page (P_0) and the hash tree page (P_1) are disseminated in plain. Finally, the packets of the remaining pages are encoded using the LT-encoder as described in Section 7.2. Notice that each page represents a separate set of source packets. Hence, packets are encoded page by page starting from P_2. Algorithm 6 summarizes the secure encoding process. Figure 7.3 illustrates an example.

Security enhanced decoding The security of the decoding process relies on the security of the underlying authentication method, i.e., Ugus et al.'s approach [3]. In this approach, starting from the page P_2, each packet can be verified with an authenticator

Algorithm 6 Security enhanced LT-encoder

```
Require: A page $P_i = \{p_{i,1} \dots, p_{i,q}\}$ of an authenticated software update $SU_{Auth} \setminus \{P_0, P_1\}$ and the degree distribution $\Omega(\cdot)$
Ensure: Encoded packets $p'_{i,j}$ of page $P_i$
1: choose a random degree $d_{i,j}$ from the degree distribution $\Omega(d_{i,j})$
2: choose a set of random neighbors $N$ composed of $d_{i,j}$ elements (i.e., $N \subseteq P_i$)
3: $p'_{i,j} \leftarrow (c_{i,j} || \bigoplus_{l=1}^{d_{i,j}} p_{i,l})$, where $c_{i,j}$ is the coefficient string and $p_{i,l} \in N$
4: return $p'_{i,j}$
```

which has been already received in the previous page. Hence, all packets of degree one are on-the-fly verifiable[1].

The security of the decoding process is based on the on-the-fly verifiable packets. The decoded packets are moved to buffer A only if they have a degree one. All degree-one packets are verifiable due to the underlying packet authentications. Suppose that the decoder has received (or derived) the jth packet of a page j, $p_{i,j}$, of degree $d = 1$. Due to the underlying remote programming protocol (i.e., Deluge), such a packet can be received only if all packets of the previous page $j - 1$ have been received. The jth packet of page $i - 1$ has the authenticator of the jth packet of page i. Hence, the decoder can verify the authenticity of all packets of degree one by checking if $h_1(p_{i-1,j} = h_1(p_{i,j})$ holds. The decoder stores only the authentic packets in buffer A. Algorithm 7 summarizes the security enhanced LT decoding process.

7.4.2 Mitigating coefficient vector attacks

Coefficient vector attacks change the degree of encoded packets to a value greater than one, i.e. $d > 1$. Hence, all such packets need to be stored buffer B. Since the capacity of buffer B is limited, a selection strategy needs to be determined for dropping the packets when the buffer is full. Two strategies were proposed in [4]. The first strategy selects the packet to be dropped randomly. The second strategy proposes to drop a packet with the maximum degree.

Figure 7.4 compares both strategies for 64 source packets. The degree of the encoded packets were chosen between 1 and 8. The adversarial strategy was to change the degree of the encoded packets to the maximum possible value, i.e. 8. The number of packets

[1] The reader is referred to Section 6.2 for a more detailed information on the security of [3].

Algorithm 7 Security enhanced LT-decoder

```
Require: Arriving packet $p'_{i,j}$, two buffers $A$ and $B$ and a temporary buffer $Temp = \emptyset$
Ensure: Updated buffers $A$ and $B$
 1: if $p'_{i,j} \in A \cup B$ then
 2:   {/* drop duplicate packets */}
 3:   drop $p'_{i,j}$ and return $A$, $B$
 4: end if
 5: if $D(p'_{i,j}) == 1$ then
 6:   {/* verify the authenticity of the received packet */}
 7:   if $h_1(p'_{i,j}) \neq h_1(p'_{i-1,j})$ then
 8:     {/* received packet is NOT authentic, drop it */}
 9:     drop $p'_{i,j}$ and return $A$, $B$
10:   else
11:     {/* received packet is authentic */}
12:     for all $p_k$ in $B$ having $p'_{i,j}$ as a neighbor do
13:       $p_k \leftarrow p_k \oplus p'_{i,j}$
14:       if $D(p_k) == 1$ then
15:         {/* move $p_k$ from buffer $B$ into buffer $Temp$ */}
16:         $B \leftarrow B \setminus \{p_k\}$ and $Temp \leftarrow Temp \cup \{p_k\}$
17:       end if
18:     end for
19:     $A \leftarrow A \cup \{p'_{i,j}\}$
20:     if $Temp \neq \emptyset$ then
21:       {/* use the new degree-one packtes for further decoding */}
22:       $p'_{i,j} \leftarrow p_k \in Temp$ and $Temp \leftarrow Temp \setminus \{p_k\}$
23:       jump to step 7
24:     end if
25:   end if
26: else
27:   for all $p_k$ in $A$ that are a neighbor of $p'_{i,j}$ do
28:     $p'_{i,j} \leftarrow p_k \oplus p'_{i,j}$
29:   end for
30:   if $D(p'_{i,j}) == 1$ then
31:     if $h_1(p'_{i,j}) \neq h_1(p'_{i-1,j})$ then
32:       drop $p'_{i,j}$ and return $A$, $B$
33:     else
34:       jump to step 7
35:     end if
36:   else
37:     $B \leftarrow B \cup \{p'_{i,j}\}$
38:   end if
39: end if
40: return $A$, $B$
```

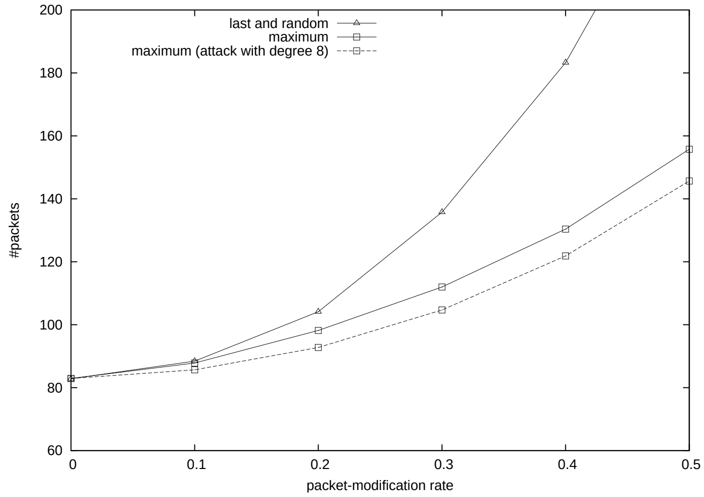


Figure 7.4: Performance of packet dropping strategies [4] - The encoder is given 64 source packets. The degree of encoded packets are chosen randomly between 1 and 8. The adversarial strategy is to change the degree of the packets to 8. Simulation results show that dropping the packets with the highest degree is the proper choice for high packet modification rates. All strategies behave similar when the packet modification rate is small.

needed for recovering the entire set of source packets in both strategies were roughly equal for small packet modification rates (e.g., 0.1). That is, dropping a random packet from the buffer provided no better performance than dropping an arriving packet of degree $d = 8$. The reason for this is the random choice of the degrees at the encoder. That is, the probability of receiving a packet of degree $d = 8$ is equal to the probability that a packet chosen from the buffer randomly has the degree $d = 8$.

As one might expect, the strategy of dropping a packet of degree $d = 8$ performed better than the random strategy for larger packet modification rates. The reason is that in such a case, the probability that a packet of degree $d = 8$ is malicious one is much higher than that is a legitimate one. In conclusion, dropping the packets of maximal degree turns out to be the best strategy to mitigate coefficient vector attacks.

7.5 Overhead analysis

The performance of the security enhanced decoder was analyzed in terms of the communication and storage overheads. Simulations were performed in the Sage mathematics environment [147]. The encoder and the decoder were implemented using Python. The

Degree distribution $\Omega(d)$	32	64
$\Omega(d = 1)$	0.212	0.161
$\Omega(d = 2)$	0.351	0.400
$\Omega(d = 4)$	0.288	0.256
$\Omega(d = 8)$	0.101	0.101
$\Omega(d = 16)$	0.048	0.045
$\Omega(d = 32)$	—	0.037
packets needed (avg)	43.8	82.8
standard deviation	6.8	9.3

Table 7.1: Optimal degree distribution for 32 and 64 source packets (taken from [11]).

results are the average of 1000 executions.

7.5.1 Parameter selection for simulations

Two simulation parameters need to be chosen: the number of source packets and the optimal degree distribution.

Number of source packets The performance of LT codes depends on the degree distribution which, in turn, depends on the number of the source packets to be communicated. The degree distribution proposed by Luby [49] is optimized for a large number of source packets. However, the number of source packets in a remote programming scenario is much smaller. In fact, software updates are disseminated page by page. Hence, the number of source packets is limited with a page size. The page size depends on the amount of RAM available on the sensor nodes. Remote programming protocols such as Deluge [144] set the page size typically to 32 or 64 packets. Hence, the number of source packets in the simulations were set to 32 or 64 packets.

Optimal degree distribution The degree distribution proposed by Luby [49] is not optimal for 32 and 64 packets. An optimal degree distribution for such a small number of packets was proposed in [11, 148]. It is depicted in Table 7.1. In simulations, all packets were encoded using the degree distributions given in Table 7.1.

7.5.2 Communication overhead

The use of LT codes is beneficial when the communication channel is unreliable. In order to determine how beneficial it is, the following experiments were executed. 64 packets were disseminated to a number of receivers ranging from 4 to 100 with LT codes and without LT codes (i.e., in plain with Deluge, see Section 6.1). The unreliability of the communication channel was simulated by dropping a number of arriving packets at each receiver. The number of packets to be dropped were determined according to a desired packet-loss probability ranging from 0.1 to 0.8. The packets to be dropped were chosen randomly and the choices were independent at each receiver.

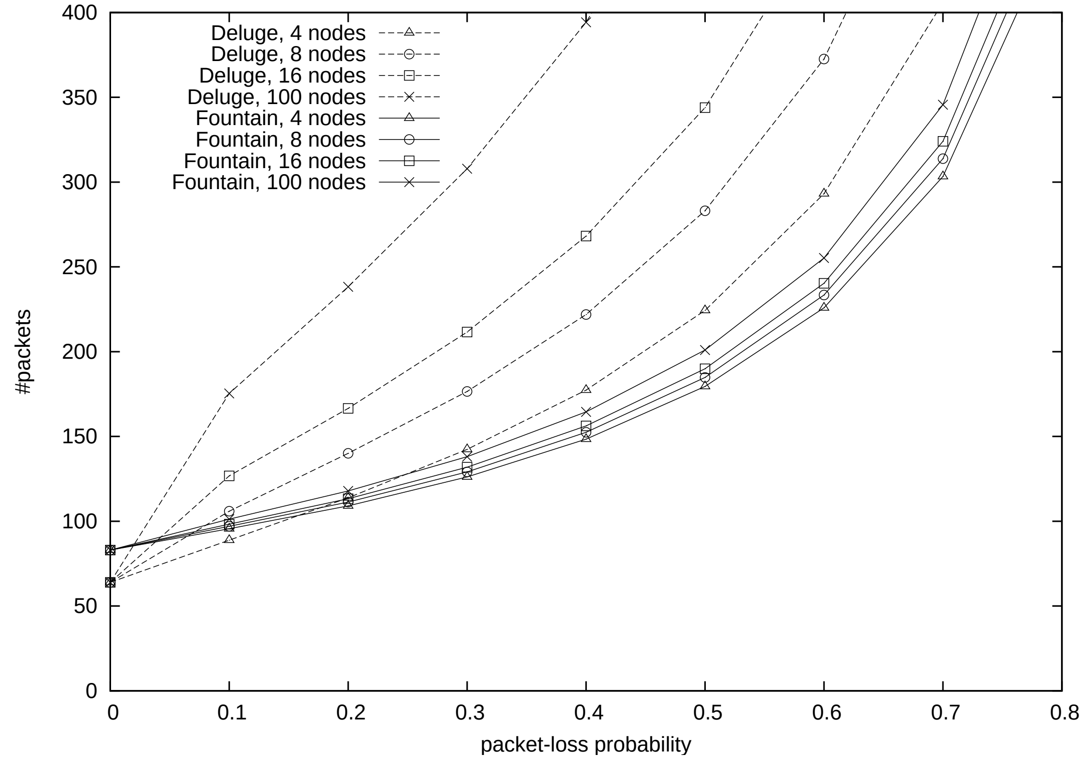


Figure 7.5: Performance of LT codes with respect to packet losses [4] - LT codes are an appropriate choice when the packet-loss rates are high. Moreover, due to the encodings, the use of LT codes in large networks turns out to be the better choice.

Simulation results are shown in Figure 7.5. The number of plain packets that need to be transmitted over an ideal channel (i.e., packet-loss rate is 0) for recovering the entire source packets is 64. It is approximately 83 packets when the LT codes are used. However, the performance of LT codes becomes visible as the packet-loss probability as well as the number of receivers increases. The obvious reason of having a better performance when the packet-loss rate is high is that there is no need for packet retransmissions with LT codes. In case of a plain transmission, each missing packet needs to be retransmitted. The reason of having a better performance when the number of receivers is large is that an encoded packet contains the information of d random packets. Hence, a single encoded packet can be useful for decoding different packets at different receivers. However, in case of a transmission of plain packets with Deluge, each receiver needs to be provided with the packets missing to it. Finally, Figure 7.5 implies that the use of LT codes becomes beneficial when the probability of packet-loss rates is larger than 0.075.

7.5.3 Storage overhead

Each receiver needs to allocate two buffers A und B. The buffer A contains the decoded packets. Hence, the size of the buffer A must be equal to the number of source packets. The buffer B stores the encoded packets. Hence, its size is not obvious and needs to be determined through simulations. The total buffer size $(A+B)$ that need to be allocated at each receiver for recovering 32 and 64 source packets is depicted in Figure 7.6 and Figure 7.7.

Figure 7.6 shows that buffering 44 packets are sufficient to recover all of 32 source packets in 95% of all transmissions. As shown in Figure 7.7, the total buffer size of 83 packets is required to have the same level of success probability for recovering 64 source packets. In the remaining cases, a packet with the maximal degree has to be dropped from the buffer B when the buffer is full.

7.6 Conclusion

The use of rateless erasure codes in disseminating software updates is beneficial in highly unreliable WSNs. However, the performance improvement is not for free. The penalty is that DoS attacks become more effective when the rateless erasure codes are used. This chapter presents security enhancements for the LT codes so that they can be used with the remote programming protocols. In particular, security means to mitigate DoS attacks as well as malicious software update attacks are presented.

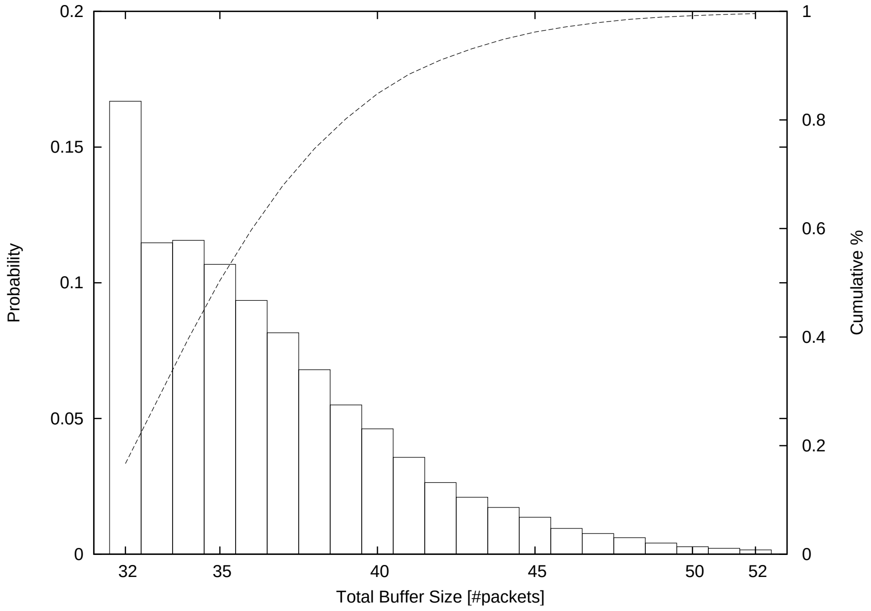


Figure 7.6: LT-decoder buffer size for 32 packets [4] - A buffer size of 44 packets $(A + B)$ is sufficient to recover all 32 source packets in 95% of all transmissions.)

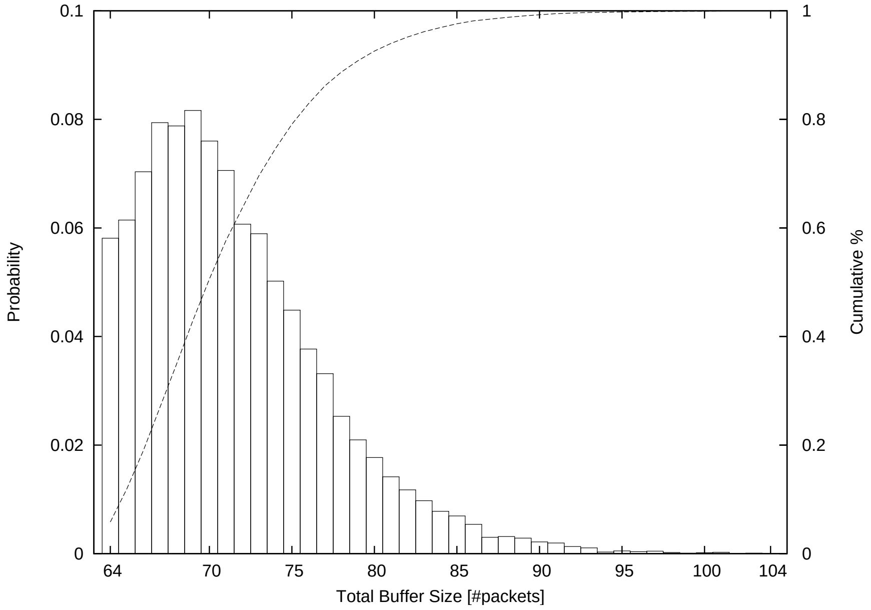


Figure 7.7: LT-decoder buffer size for 64 packets [4] - A buffer size of 83 packets $(A + B)$ is sufficient to recover all of 64 source packets in 95% of all transmissions.

Security against malicious software attacks are realized by authenticating the software updates with Ugus et al.'s approach [3] described in Section 6.2. Security against DoS attacks are realized using an optimal packet dropping strategy.

8

Confidentiality Protection of Software Updates

Protecting the authenticity and the integrity of software updates are the primary security goals in a remote programming scenario. Otherwise, the adversary can program the sensor nodes with his/her malicious software. Protection against DoS attacks are also an essential requirement. Otherwise, the adversary can prevent the WSN from fulfilling its duty by exhausting the resources (i.e., energy or storage) of the sensor nodes. Another important security goal is the protection of the confidentiality of the software updates. The eavesdropping adversaries must be prevented from gaining information on the software updates being disseminated. Otherwise, they might take advantage of this information.

This chapter presents an efficient approach for protecting the confidentiality of the software updates. It was originally proposed in [5]. In this chapter, that approach is applied to the secure remote programming protocol presented in Chapter 6. Moreover, the performance of the presented approach with Synapse++[52] will be presented as a case study.

8.1 Confidentiality protection

Motivation Protecting the authenticity and the integrity of software updates is a mandatory security requirement. Security against DoS attacks is an essential requirement, too. The confidentiality protection is required to prevent the eavesdroppers from gaining information on what is being disseminated. Hence, this security requirement is dependent on the application scenario and the software being disseminated. It is obviously a fundamental requirement if the software update is a military application. A motivating argument in the public sector might be the commercial activities. Consider companies developing WSN applications. Such companies invest money and resources (material and human) for developing those applications. Frequently, software licensing is the only way for those companies to continue with the business. Hence, disseminating software updates without confidentiality protection might endanger the companies making business by selling WSN software licenses.

Challenges and limitations Node compromise attacks are possible if the adversary has physical access to the sensor nodes. If the adversary can have the required contact, he/she needs less than a minute to extract the entire content of the flash memory (ROM) and EEPROM [68]. In such a case, it does not matter whatever encryption method was used during the software dissemination. Hence, protecting the confidentiality of a software running on a sensor node without using a tamper resistant hardware is probably impossible against node compromise attacks. A possible countermeasure would be using a tamper resistant memory unit. However, such solutions based on tamper resistant hardware are out of the scope of this thesis. This thesis considers only software-based solutions. Hence, the proposed approach is not secure against node compromise attacks.

8.2 The approach

Preliminaries and assumptions The assumptions as well as the security and network models are the same as those ones described in Chapter 6. The main difference is that the sender (i.e., the monitoring device) and the receivers (i.e., the sensor nodes) share a secret key k which is used for encrypting and decrypting the software updates. It is assumed that the key is shared during the pre-deployment phase of the sensor nodes (i.e., before the very first remote programming operation).

Figure 8.1 illustrates the security flow for a secure remote programming with confidentiality protection. Firstly, the software update is encrypted by the monitoring

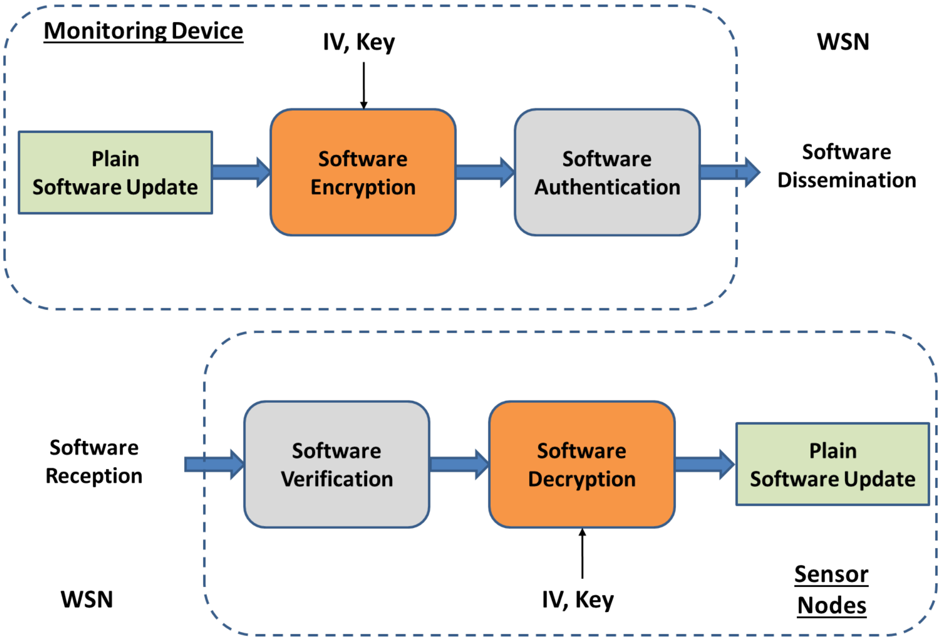


Figure 8.1: Remote programming with confidentiality protection - Schemata of a remote programming operation with encrypted and authenticated software updates: First, the software update is encrypted using an encryption mechanism. After that, the encrypted update is authenticated using a mechanism like the one presented in [3] (see Chapter 6). These operations are performed at the monitoring device. Finally, the encrypted and authenticated software update is disseminated using a remote programming protocol. The sensor nodes perform these operations in the reverse order. That is, first, the authenticity of the received software update is verified. Subsequently, it is decrypted using the decryption algorithm.

device (i.e., the sender of the software update) using the shared secret key k. Subsequently, the encrypted update is authenticated for example as described in Chapter 6. Finally, the encrypted and authenticated software update is transmitted to the sensor nodes. The sensor nodes perform these operations in the reverse order. That is, firstly, the authenticity of the received software update is verified. If it is authentic, the sensor nodes decrypt the received update using the shared secret key k.

The main optimization goal in this thesis is the code size. Hence, an encryption algorithm with a small code size needs to be chosen. More exactly, the decryption method must require a small code size.

8.2.1 Design choices

The software updates are encrypted with a symmetric algorithm, i.e. wit a block cipher. Hence, there are two factors that impact the code size: the choice of the block cipher and the choice of the mode of operation.

Choice of the block cipher The basic building block for any symmetric encryption is a block cipher. Hence, designing an encryption suite must start with choosing a suitable block cipher. The choice mainly depends on the optimization goals to be achieved. The main optimization goals are small memory footprint, high performance as well as low energy consumption. In this thesis, the small memory footprint is the most important goal to achieve.

Most of the radio chips found on modern sensor platforms today provide a hardware encryption module. For example, the Chipcon CC2420 [20] radio chip is equipped with the hardware implementation of AES-128 [105, 106]. Using a hardware module for encryption is obviously the most appropriate way to reduce the code size requirement. Hence, the approach proposed in this thesis assumes that the radio chip provides a hardware encryption module such as AES-128.

Choice of the mode of operation Block ciphers take a fixed-length plaintext and a fixed-length key and produce a fixed-length ciphertext [7]. The length of the plaintext is equal to the length of the ciphertext. It is referred to as block length.

A block cipher cannot encrypt data which is larger than its block length. In order to encrypt arbitrary-length data, block ciphers needs to be used in conjunction with a so called mode of operation. There are several modes of operation such as Electronic Code Book (ECB), Cipher Block Chaining (CBC), Output Feedback (OFB), Cipher Feedback (CFB) and Counter (CTR) [7]. The ECB mode is not secure. Hence, it must not be used [7]. The main difference between the CBC mode and the other modes is that the CBC mode requires the decryption function of the block cipher for decrypting the encrypted data. Decryption with other modes can be done using only the encryption function of the block cipher. Hence, the CBC mode is not an optimal choice when the code size is considered. Furthermore, the CC2420 radio chip implements only the encryption function of AES-128. Hence, the CBC mode cannot be used. As a result, the OFB, CFB and CTR modes turn out to be the only possibilities. The OFB mode was chosen due to its good performance [149].

8.2.2 Software update encryption and decryption

This subsection describes how to extend the secure remote programming described in Chapter 6 with the confidentiality support. This extension requires a slight modification in the software partitioning step described in Section 6.2.2. All other steps, i.e. software authentication and dissemination, remain the same. Hence, in the following, only the software partitioning as well as the software encryption and decryption operations are described.

Software update partitioning with confidentiality support Compared to the partitioning method described Section 6.2.2, there are only two differences. Firstly, an additional n bits space needs to be reserved in the first packet of each page[1]. Secondly, the partitioning is done such that the payload of each packet, excluding the authenticators, is at least n bits long. Depending on the payload size of the underlying network protocol, it can be a multiple of n bits, too. The exception is the last packet of each page. Its payload length can be smaller than n bits.

As an example, consider the software update given in Example 4 (see Section 6.2.2). Figure 8.2 illustrates the partitioning of the software update with confidentiality support.

[1] Hence, n denotes the block length of the block cipher available for the encryption. For AES-128, $n = 128$.

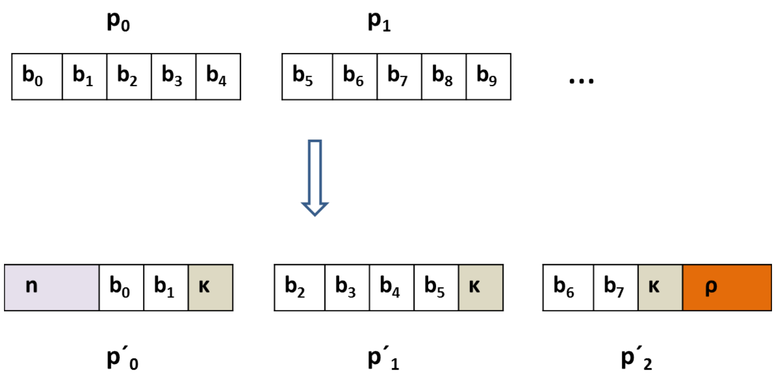


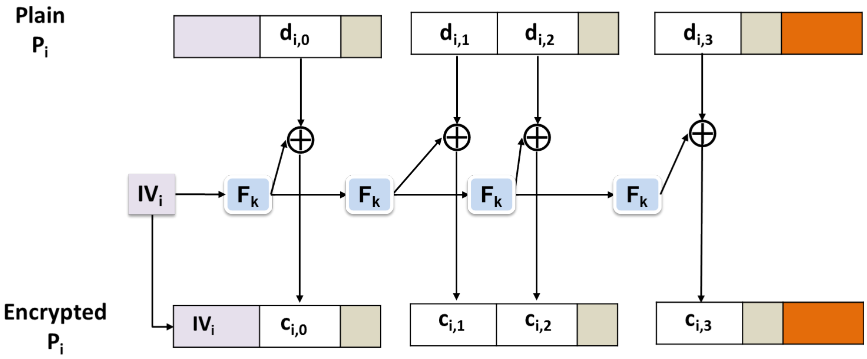
Figure 8.2: Partitioning a software update with confidentiality support - Assume that the page size is three packets and the packet size is 5 bits (see Example 4 in Section 6.2.2). Furthermore, assume that $\kappa = 1$, $\rho = 2$, and $n = 2$. Then, a page is partitioned as follows: in all packets of a page κ bits space is reserved. In addition to that, the first packet p'_0 reserves n bits space and the last packet reserves ρ bits space. Moreover, the packets are partitioned such that the length of the payload of the resulting packets are either n bits or one of its multiples. The last packet is an exception. Its payload length can be smaller than n bits.

Software update encryption Suppose that $\{P_2, \cdots, P_y\}$[1] denotes the pages of a software update after partitioning as described in Figure 8.2 and F is a block cipher of length n (i.e., AES-128). Furthermore, assume that $d_{i,j}$ denotes the jth data payload of the ith page, P_i. The software update is encrypted page by page by applying the encryption function to each n-bit data payload. The resulting encrypted blocks, $c_{i,j}$, are computed as

$$c_{i,j} = d_{i,j} \oplus r_{i,j}, \tag{8.1}$$

where $r_{i,j} = F_k(r_{i,j-1})$ and $r_{i,-1} = IV_i$. The secret key k is a shared key uploaded into the sensor nodes during their deployment. IV_i is an n-bit random initialization vector. It is chosen by the monitoring device and appended to the first packet of each encrypted page as a prefix. Figure 8.3 illustrates the encryption of a page P_i partitioned as in Figure 8.2.

[1] Notice that the signature and the hash tree pages, i.e. P_0 and P_1, are not encrypted. The reason is that they bootstrap the security of the software authentication. Hence, they must be disseminated in plaintext.



Encryption: $c_{i,j} = d_{i,j} \oplus r_{i,j}$

$r_{i,j} = F_k(r_{i,j-1})$ **and** $r_{i,-1} = IV_i$

Figure 8.3: Software update encryption with the OFB mode - Each data block $d_{i,j}$ of length of n is encrypted by cipher F using a key k which is a shared secret key between the sender and the receivers. IV_i is a random bitstring. It is chosen by the monitoring device and called the initialization vector. It needs to be different for each page. For $j \geq 1$, the encryption of block $j-1$ is used as the initialization vector of block j.

Software update decryption with the OFB mode: The decryption operation is identical with the encryption operation. Only the encrypted data blocks are swapped with the plain data blocks:

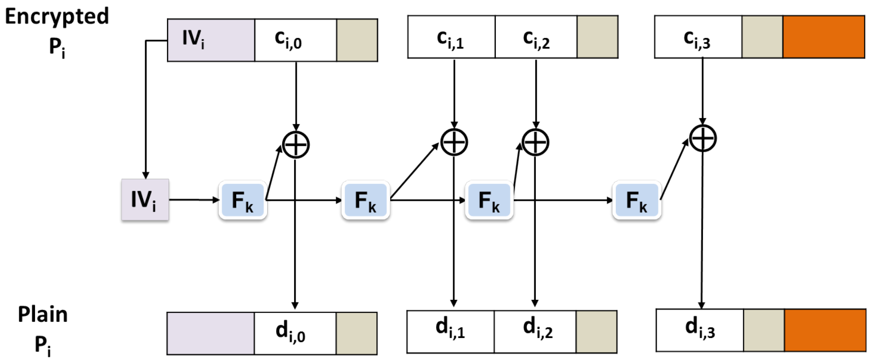
$$d_{i,j} = c_{i,j} \oplus r_{i,j}, \tag{8.2}$$

where $r_{i,j}$ and IV_i are as described above. Figure 8.4 illustrates the decryption of a page encrypted as in Figure 8.3.

Authenticating encrypted software updates The next step after encrypting a software update is its authentication. If the software updates are disseminated using Deluge [144] as described in Chapter 6, the authentication is performed exactly as described in Section 6.2.2.

8.3 Security analysis

The security of the presented approach depends on the underlying block cipher, the OFB mode, and finally the secrecy of the shared key k.



Decryption: $d_{i,j} = c_{i,j} \oplus r_{i,j}$

$r_{i,j} = F_k(r_{i,j-1})$ **and** $r_{i,-1} = IV_i$

Figure 8.4: Software update decryption with the OFB mode - The decryption operation is identical with the encryption operation shown in Figure 8.3. Only the encrypted data blocks are swapped with the plain software data blocks.

The block cipher AES-128 is an encryption standard due to its high level of security and performance. There is no known attack which breaks the security of AES-128 faster than the exhaustive search (i.e., brute force attack) [100, 150]. Hence, the presented approach, based on AES-128, offers an excellent security if the OFB mode is implemented properly and the shared key is unknown to the adversary.

The security of the OFB mode depends on the underlying block cipher and the initialization vector (IV). The underlying block cipher is AES-128 which is secure. Hence, the presented approach remains secure if the initialization vectors are chosen randomly [100].

The adversary can obtain the secret key k by physically compromising a sensor node. Hence, the confidentiality protection cannot be guaranteed in case of node compromise attacks. The impact of such attacks might be reduced by refreshing the shared key periodically by using e.g. a key distribution scheme. However, implementing such schemes not only increases the code size, but also requires the exchange of a large number of packets. Moreover, if the adversary can physically compromise a sensor node, it does not matter whatever key was used during the software dissemination. The adversary can simply obtain the software running on a sensor node by extracting its memory content. Considering this fact, the presented approach offers an efficient compromise between the security level offered and the code size required. In addition, compromising the secret key does not help the adversary with compromising the software authentication which is the primary security goal in a remote programming scenario.

8.4 Case study: Secure Synapse++

Bui et al. [5] proposed security extensions for the remote programming protocol Synapse++ [52]. Confidentiality protection is implemented using the approach presented in the previous section. Software authentication is also performed in analog to the idea described in Chapter 6. However, there are some differences due to the characteristics of the Synapse++.

Synapse++ uses LT codes for disseminating the software updates. Thus, it is quite similar to Deluge's extension with the LT codes support which is described in Chapter 7. Besides some protocol specific differences, the main difference lies in the implementation of the LT-decoder. Synapse++ applies the Gaussian elimination method for decoding. Thus, the software updates are decoded at a much larger granularity (e.g., page) than the packet granularity as done in Algorithm 7. This results in a slightly different software authentication method than in [3, 4]. The advantage of Synapse++ is that the pages of a software update can be verified even if they arrive out-of-order. Its main disadvantage is the weaker security against DoS attacks. Detecting a malicious packet in a page requires to decode that page entirely.

Since Synapse++ uses a different approach for authenticating the software updates, the software partitioning needs to be done in a slightly different way as well. Let $SU_{Auth} = P_0||P_1||P_2||\dots||P_y$ denote the resulting software update after a software update SU is authenticated. In the following, the software partitioning, the software encryption and decryption as well the software authentication for Synapse++ are described.

Partitioning software updates for Synapse++ The P_0 is the signature page. It is used to bootstrap the security of the software update. Hence, it is disseminated without being encrypted and encoded. The remaining pages, P_1 to P_y, carry the actual software update. Hence, they need to be prepared for the later encryption. That is, they need to be partitioned. The software partitioning for Synapse++ requires no ρ bits space reservation in the last packet of each page. However, κ bits space in each packet of each page is reserved for later addition of the authenticators. Additionally, n bits space is reserved in the first packet of each page for storing the initialization vector. Figure 8.5 illustrates the partitioning of a page for $\kappa = 1$ and $n = 2$.

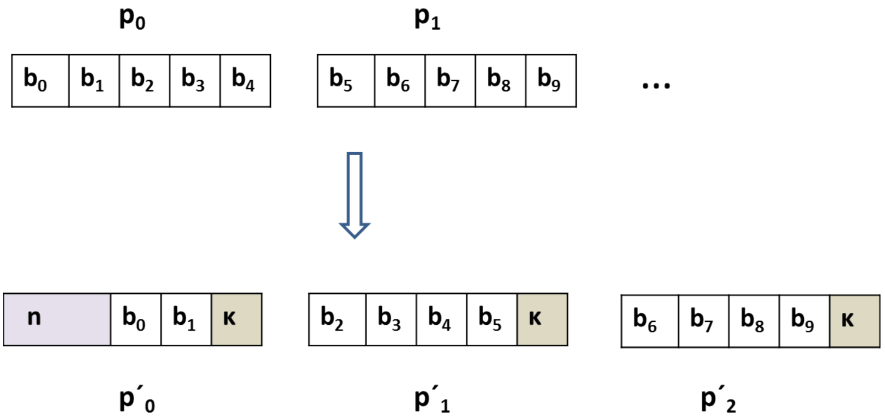


Figure 8.5: Partitioning a software update for Synapse++ - Assume that the page size is three packets and the packet size is 5 bits (compare with Figure 8.2). Furthermore, assume that $\kappa = 1$ and $n = 2$. Then, a page is partitioned as follows: in all packets of a page κ bits space is reserved. In addition to that, the first packet p_0' reserves n bits space as a prefix. Moreover, the packets are partitioned such that the length of the payload of the resulting packets is either n bits or one of its multiples. The last packet is an exception. Its payload length can be smaller than n bits.

Once the software update is partitioned, it is encrypted.

Encrypting software updates for Synapse++ The software encryption and decryption for Synapse++ are done as described in Section 8.2.2 (cf. Figures 8.3 and 8.4).

Authenticating (Signing) software updates for Synapse++ Let $H = h(P_1)||\cdots||h(P_y)$ denote the ρ-bit hash values of the encrypted pages of a software update. The signature σ_{SU} is computed by signing H and ξ using the $\mathfrak{T}$-time signature scheme. Notice that ξ is some update information that contains e.g. the version number of the new software. The signature page is composed of the signature σ_{SU}, the hash values H, and ξ. Figure 8.6 illustrates the entire authentication process of a software update graphically.

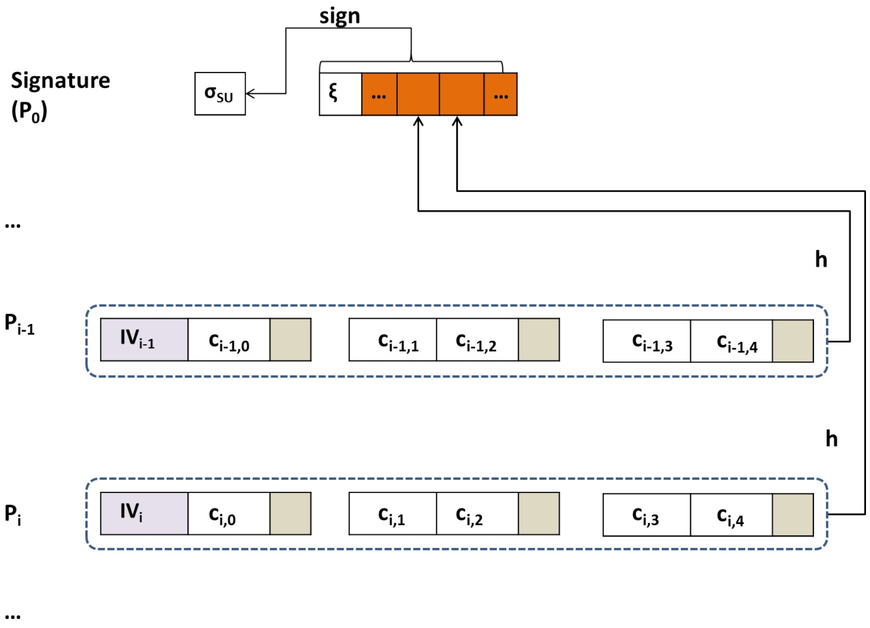


Figure 8.6: Authenticating (Signing) a software update for Synapse++ - Synapse++ signs a software update as follows: The signature σ_{SU} is computed over the hash values of the encrypted pages $H = h(P_1)||\cdots||h(P_y)$ and the ξ using the $\mathcal{T}$-time signature scheme. ξ is some update information carrying e.g. the version number of the new software.

DoS protection for Synapse++ The CC2420 radio chip provides a hardware implementation of the CBC-MAC. Hence, the packets of the software updates for Synapse++ are authenticated with the CBC-MAC against online modifications. Each packet is authenticated with a κ-bit MAC tag which is stored in the κ bits space reserved during the software partitioning. The MAC key k is a shared key and uploaded into the sensor nodes during their deployment. Figure 8.7 illustrates the computation of the MAC tags for the pages of an encrypted software update.

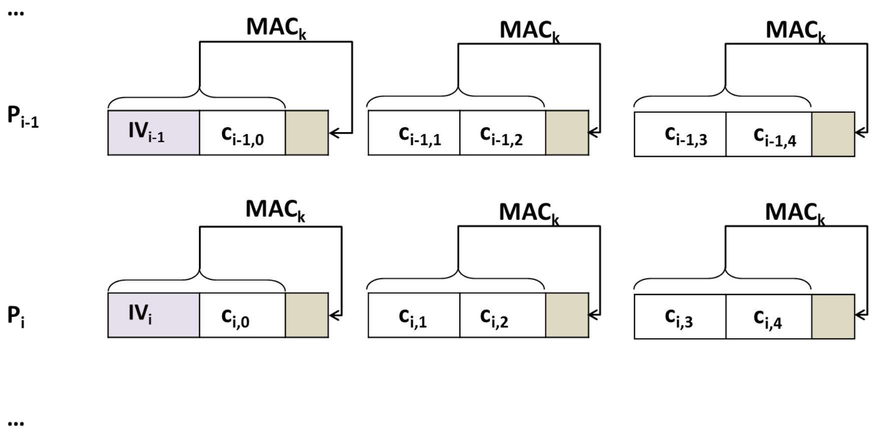


Figure 8.7: DoS protection for Synapse++ - Synapse++ mitigates DoS attacks based on online packet modifications by authenticating the packets with a MAC tag. Each packet is authenticated with a shared secret k.

8.5 Prototype implementation and performance analyze

Bui et al. [5] analyzed the performance of secure Synapse++ with a prototype implementation[1].

8.5.1 Prototype implementation

Bui et al. [5] implemented the secure Synapse++ with the following security parameters. The security parameter ρ against malicious software attacks was chosen to be 80 bits. That is, the length of each hash value of an encrypted page, which are signed with the $\mathfrak{T}$-time signature scheme, is 80 bits. The security parameter κ against DoS attacks, i.e. online packet modifications, was chosen to be 32 bits. That is, the length of each MAC tag is 32 bits. Finally, the software updates are encrypted using AES-128 which provides 128-bit confidentiality protection.

8.5.2 Performance evaluation

Bui et al. [5] compared the performance of the secure Synapse++ with the Synapse++ without the security extensions. The comparisons were performed on a testbed com-

[1] Notice that the prototype implementation and the performance evaluation were not done by the author of this thesis. The author was mainly involved in the design of the security concepts.

posed of 55 TelosB sensor nodes. The testbed was located at the Department of Information Engineering of the University of Padova [5]. The results are the average of 25 executions.

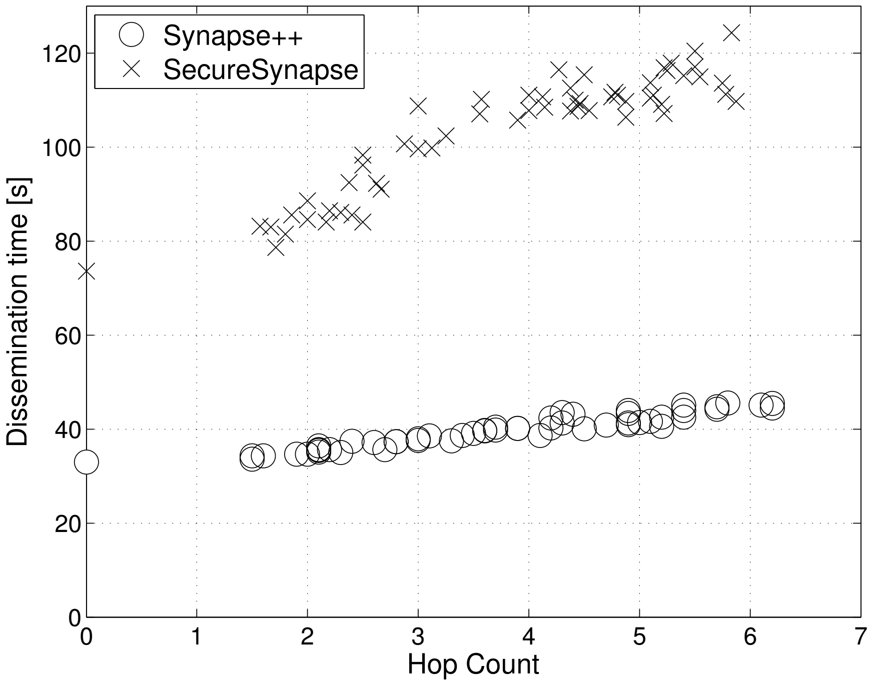


Figure 8.8: Dissemination times of a 10KB software update with the secure and insecure Synapse++ [5] - Synapse++ without security extensions is approximately twice as fast as the secure Synapse++. Increased update size as well as the additional security operations such as signature verification, decryption and MAC computations are the reason of this performance loss.

Figure 8.8 shows the times required for disseminating a 10KB software update with the secure Synapse++ and the Synapse++. The Synapse++ is more than twice as fast as the secure Synapse++. There are several reasons for this performance loss. Firstly, the size of a software update in case of the secure Synapse++ is 465B larger. Secondly, the sensor nodes require approximately 410ms for verifying the received signature. Finally, the secure Synapse++ requires to decrypt as well as to verify each received packet. Hence, the secure Synapse++ disseminates each page roughly 30% slower.

8.6 Conclusion

This chapter presents a simple approach for protecting the confidentiality of software updates. It is based on the hardware implementation of AES which is provided by the radio chip of most modern sensor platforms. The presented approach proposes to share a secret key between the sender (i.e., monitoring device) and the receivers (i.e., sensor nodes). Software updates are encrypted and decrypted using AES in the OFB mode. The motivation behind this simple approach is that it does not matter whatever encryption method is used for protecting a software update, if the adversary can obtain it by physically compromising the sensor nodes. Hence, using sophisticated approaches e.g. for refreshing the keys is likely to help little with solving this main problem. The presented approach on the other hand offers an efficient compromise between the security level in terms of confidentiality and the code size. In addition, compromising the secret key does not help the adversary with compromising the software authentication which is the primary security goal in a remote programming scenario.

9

Conclusion and Future Work

Due to their low cost and self organization capabilities, WSNs are used in many application scenarios such as environmental monitoring, industrial automation, home automation, health monitoring, and military applications. WSN applications need to be maintained after their deployment. Typical maintenance works include updating the software running on the sensor nodes to adapt it to the changing application requirements as well as to remove the software bugs and the security vulnerabilities. Manual software updates would not be practical in particular when the number of the sensor nodes is large or when they are deployed in difficult-to-reach fields. Hence, software updates must be performed ideally over-the-air.

However, programming the sensor nodes remotely over-the-air introduces new challenges, too. These challenges might be partitioned in two main groups: implementation challenges and security challenges. Implementation challenges are due to the limited resources available to the sensor nodes. Sensor nodes are typically equipped with a very limited program memory (Flash memory ROM) and data memory (RAM). Since the remote programming mechanism must share the available memory with the actual application, it needs to be optimized in terms of the code size. Moreover, the limited power available to sensor nodes requires the remote programming mechanism to be efficient, too. Security challenges are due to the typical characteristics of the WSNs. WSNs are often deployed in public and even hostile environments. Hence, the remote programming mechanism must be protected against adversarial interferences. These include protecting the authenticity, integrity, and confidentiality of the software updates. Furthermore, the remote programming mechanism must mitigate DoS attacks aiming at depleting the resources (power and memory) of the sensor nodes.

This thesis proposes a code-size optimized cryptographic toolbox in order to address the implementation challenges. The main idea is to build all security primitives required for a secure remote programming from a single symmetric building block. High performance is achieved by using symmetric algorithms. The code size is reduced due to the code sharing. Chapter 4 describes how such a cryptographic toolbox can be realized using a block cipher. It contains all essential cryptographic primitives required to implement the basic security goals. More precisely, it is composed of a hash function, which is the fundamental primitive for almost all security solutions, a MAC function for protecting message authenticity and integrity, and finally a signature scheme, for providing non-repudiation in addition to authenticity and integrity protection. The signature scheme is a dedicated design for the secure remote programming mechanism.

The security challenges are addressed by proposing several security mechanisms for protecting the authenticity, integrity, and confidentiality of the software updates. Moreover, the security mechanisms are designed to mitigate the DoS attacks.

Chapter 5 analyzes the existing broadcast security mechanisms in terms of their applicability to the remote programming scenario. Based on this analysis, an improved version of the off-line signing [12] has been proposed to authenticate the software updates. The main improvement compared to the original scheme [12] lies in increasing the robustness against the packets losses. It has been realized by applying multiple authenticator chains instead of only one.

Chapter 6 presents an efficient and code-size optimized approach for protecting the authenticity and integrity of the software updates. This approach was originally proposed in [3]. It relies on the improvement of the off-line signing [12]. The code size of the proposed approach is very small mainly due to the code-size optimized cryptographic toolbox including the stateful-verifier $\mathcal{T}$-time signature scheme. DoS attacks are mitigated by using multiple hash chains at multiple granularities. The implementation results show that the the proposed approach occupies only 1% of the available flash memory on a TelosB platform. This is a tremendous improvement compared to the approaches using elliptic curve based signature schemes occupying 28% of the available flash memory on the same platform. Such a great improvement is mainly possible due to the $\mathcal{T}$-time signature scheme presented in Chapter 5.

Chapter 7 proposes security mechanisms for a remote programing protocol using fountain codes. It was originally proposed in [4, 146]. Fountain codes allow for an efficient and reliable software dissemination even in highly unreliable environments. However, the price to pay is the new security problems. In particular, DoS attacks become more powerful due to the natural error-propagating property of the fountain codes. Chapter 7 presents security means to mitigate DoS attacks as well as malicious software update attacks. Security against DoS attacks is realized using an optimal packet dropping strategy. Security against malicious software attacks are realized by authenticating the software updates before their dissemination.

Finally, Chapter 8 presents an efficient approach for protecting the confidentiality of the software updates. It was originally proposed in [5]. It is applied to the secure remote programming presented in Chapter 6 in this thesis. Moreover, its application to Synapse++ [52] is presented as a case study. The proposed approach is based on the hardware implementation of AES available on most modern sensor platforms. A secret key is shared between the sender (i.e., monitoring device) and the receivers (i.e., sensor nodes). Software updates are then encrypted and decrypted using the shared secret key.

The proposed approaches assume that the WSNs are composed of a single type of sensor platform. However, WSNs in practice might contain different types of sensor platforms. The security mechanisms proposed in this thesis cannot be applied directly to such heterogeneous WSNs. Hence, improving the proposed approaches for an efficient remote programming of heterogeneous WSNs remains as the future work.

References

[1] Osman Ugus. **Asymmetric Homomorphic Encryption Transformation for Securing Distributed Data Storage in Wireless Sensor Networks**, 2007. Diploma Thesis. xi, 56

[2] Jens-Matthias Bohli, Alban Hessler, Osman Ugus, and Dirk Westhoff. **Security Solutions for Uplink- and Downlink-Traffic in Wireless Sensor Networks (Sicherheitslösungen für Uplink- und Downlink-Verkehr in drahtlosen Sensornetzen)**. *it - Information Technology*, **52**(6):313–319, 2010. xi, 68

[3] Osman Ugus, Dirk Westhoff, and Jens-Matthias Bohli. **A ROM-friendly secure code update mechanism for WSNs using a stateful-verifier T-time signature scheme**. In *Proceedings of the second ACM conference on Wireless network security*, WiSec '09, pages 29–40, New York, NY, USA, 2009. ACM. xi, 161, 195, 197, 219, 227, 228, 235, 239, 245, 252

[4] Jens-Matthias Bohli, Alban Hessler, Osman Ugus, and Dirk Westhoff. **Security enhanced multi-hop over the air reprogramming with Fountain Codes**. In *IEEE LCN*, Zurich, Switzerland, October 2009. xi, xxv, 219, 225, 226, 228, 230, 232, 234, 245, 253

[5] N. Bui, O. Ugus, M. Dissegna, M. Rossi, and M. Zorzi. **An integrated system for secure code distribution in Wireless Sensor Networks**. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, pages 575 –581, 29 2010-april 2 2010. xi, xxv, 237, 245, 248, 249, 253

[6] I.F. Akyildiz, Weilian Su, Y. Sankarasubramaniam, and E. Cayirci. **A survey on sensor networks**. *Communications Magazine, IEEE*, **40**(8):102 – 114, aug 2002. [cited from page 102]. xxiii, 1, 44

[7] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001. xxiii, 22, 61, 62, 63, 128, 129, 130, 131, 133, 136, 138, 139, 143, 144, 145, 146, 173, 176, 180, 240, 241

[8] J.D. Meier and Alex Mackman and Michael Dunner and Srinath Vasireddy and Ray Escamilla and Anandha Murukan. **Improving Web Application Security: Threats and Countermeasures**. Microsoft Patterns & Practices, 2003. xxvii, 76, 81, 83

[9] Joe Grand. **Practical Secure Hardware Design for Embedded Systems**. In *Proceedings of the Embedded Systems Conference*, San Francisco, California, USA, March 2004. xxvii, 85, 86

[10] **ECRYPT2 Yearly Report on Algorithms and Keysizes, D.SPA.7 Rev. 1.0**. `http://www.ecrypt.eu.org/documents/D.SPA.7.pdf`, July 2009. xxvii, 63, 64, 67, 68, 89, 90, 91

[11] M. Rossi, G. Zanca, L. Stabellini, R. Crepaldi, A.F. Harris, and M. Zorzi. **SYNAPSE: A Network Reprogramming Protocol for Wireless Sensor Networks Using Fountain Codes**. In *Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON '08. 5th Annual IEEE Communications Society Conference on*, pages 188 –196, june 2008. xxvii, 24, 219, 231

[12] Rosario Gennaro and Pankaj Rohatgi. **How to Sign Digital Streams**. In *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '97, pages 180–197, London, UK, 1997. Springer-Verlag. 3, 39, 189, 190, 195, 252

[13] Dargie Waltenegus and Christial Poellabauer. *Fundamentals of Wireless Sensor Netwoeks: Theory and Practice*. A John Wiley and Sons, Ltd., 2010. 7, 8, 46, 48, 49, 51

[14] Luca Mottola and Gian Pietro Picco. **Programming wireless sensor networks: Fundamental concepts and state of the art**. *ACM Comput. Surv.*, **43**(3):19:1–19:51, April 2011. [cited from page 4]. 7, 8

[15] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. **Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with ZebraNet**. *SIGPLAN Not.*, **37**(10):96–107, October 2002. 8, 12

[16] Carl Hartung, Richard Han, Carl Seielstad, and Saxon Holbrook. **FireWxNet: a multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments**. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, MobiSys '06, pages 28–41, New York, NY, USA, 2006. ACM. 9, 12

[17] **Chipcon CC1000 Datasheet,Texas Instruments**, 2006. 9, 47

[18] Richard Beckwith, Dan Teibel, and Pat Bowen. **Report from the Field: Results from an Agricultural Wireless Sensor Network**. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 471–478, Washington, DC, USA, 2004. IEEE Computer Society. 9, 12

[19] Ubisec&Sens Deliverable 4.4. **Validation of prototyping activities**, 2008. 10

[20] **Chipcon CC2420 Datasheet, Texas Instruments**, 2007. 10, 47, 240

[21] C. Manzie, H.C. Watson, S.K. Halgamuge, and K. Lim. **On the potential for improving fuel economy using a traffic flow sensor network**. In *Intelligent Sensing and Information Processing, 2005. Proceedings of 2005 International Conference on*, pages 38 – 43, jan. 2005. 10

[22] Lakshman Krishnamurthy, Robert Adler, Phil Buonadonna, Jasmeet Chhabra, Mick Flanigan, Nandakishore Kushalnagar, Lama Nachman, and Mark Yarvis. **Design and deployment of industrial sensor networks: experiences from a semiconductor plant and the north sea**. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, pages 64–75, New York, NY, USA, 2005. ACM. 10, 12

[23] Vipul Gupta, Matthew Millard, Stephen Fung, Yu Zhu, Nils Gura, Hans Eberle, and Sheueling Chang Shantz. **Sizzle: A Standards-Based End-to-End Security Architecture for the Embedded Internet (Best Paper)**. In *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, PERCOM '05, pages 247–256, Washington, DC, USA, 2005. IEEE Computer Society. 11, 12, 121

[24] Konrad Lorincz, Bor-rong Chen, Geoffrey Werner Challen, Atanu Roy Chowdhury, Shyamal Patel, Paolo Bonato, and Matt Welsh. **Mercury: a wearable sensor network platform for high-fidelity motion analysis**. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 183–196, New York, NY, USA, 2009. ACM. 11, 12, 13

[25] Joseph Polastre, Robert Szewczyk, and David Culler. **Telos: enabling ultra-low power wireless research**. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press. 15, 32, 37

[26] Philip Levis and David Culler. **Mate: a tiny virtual machine for sensor networks**. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 85–95, New York, NY, USA, 2002. ACM. 16

[27] Sam Michiels, Wouter Horré, Wouter Joosen, and Pierre Verbaeten. **DAViM: a dynamically adaptable virtual machine for sensor networks**. In *Proceedings of the international workshop on Middleware for sensor networks*, MidSens '06, pages 7–12, New York, NY, USA, 2006. ACM. 16

[28] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. **Agilla: A mobile agent middleware for self-adaptive wireless sensor networks**. *ACM Trans. Auton. Adapt. Syst.*, **4**(3):16:1–16:26, July 2009. 16

[29] Rahul Balani, Chih-Chieh Han, Ram Kumar Rengaswamy, Ilias Tsigkogiannis, and Mani Srivastava. **Multi-level software reconfiguration for sensor networks**. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, EMSOFT '06, pages 112–121, New York, NY, USA, 2006. ACM. 16

[30] Sam Michiels, Wouter Horré, Wouter Joosen, and Pierre Verbaeten. **DAViM: a dynamically adaptable virtual machine for sensor networks**. In *Proceedings of the international workshop on Middleware for sensor networks*, MidSens '06, pages 7–12, New York, NY, USA, 2006. ACM. 16

[31] René Müller, Gustavo Alonso, and Donald Kossmann. **A virtual machine for sensor networks**. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 145–158, New York, NY, USA, 2007. ACM. 16

[32] Niels Reijers and Koen Langendoen. **Efficient code distribution in wireless sensor networks**. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, WSNA '03, pages 60–67, New York, NY, USA, 2003. ACM. 16, 17, 22

[33] Tom Yeh Rahul Kapur and Ujjwal Lahoti. **Differential Wireless Reprogramming of Sensor Networks**. Technical report, December 2003. 16, 22

[34] Pascal Rickenbach and Roger Wattenhofer. **Decoding Code on a Sensor Node**. In *Proceedings of the 4th IEEE international conference on Distributed Computing in Sensor Systems*, DCOSS '08, pages 400–414, Berlin, Heidelberg, 2008. Springer-Verlag. 16

[35] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, 1999. 17

[36] Rajesh Krishna Panta, Saurabh Bagchi, and Samuel P. Midkiff. **Zephyr: efficient incremental reprogramming of sensor nodes using function call indirections and difference computation**. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 32–32, Berkeley, CA, USA, 2009. USENIX Association. 17

[37] R.K. Panta and S. Bagchi. **Hermes: Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks**. In *INFOCOM 2009, IEEE*, pages 639 –647, april 2009. 17

[38] Jaein Jeong and D. Culler. **Incremental network programming for wireless sensors**. In *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, pages 25 – 33, oct. 2004. 17, 22

[39] J. Koshy and R. Pandey. **Remote incremental linking for energy-efficient reprogramming of sensor networks**. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 354 – 365, jan.-2 feb. 2005. 17, 18

[40] Pedro Marron, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. **FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks**. In *Wireless Sensor Networks*, **3868** of *Lecture Notes in Computer Science*, pages 212–227. Springer Berlin / Heidelberg, 2006. 10.1007/11669463_17. 18

[41] Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Matthias Gauger, Olga Saukh, and Kurt Rothermel. **Management and configuration issues for sensor networks**. *Int. J. Netw. Manag.*, **15**(4):235–253, July 2005. 18

[42] Weijia Li, Youtao Zhang, Jun Yang, and Jiang Zheng. **Towards update-conscious compilation for energy-efficient code dissemination in WSNs**. *ACM Trans. Archit. Code Optim.*, **6**(4):14:1–14:33, October 2009. 18

[43] Wei Dong, Yunhao Liu, Xiaofan Wu, Lin Gu, and Chun Chen. **Elon: enabling efficient and long-term reprogramming for wireless sensor networks**. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '10, pages 49–60, New York, NY, USA, 2010. ACM. 18

[44] N. Tsiftes, A. Dunkels, and T. Voigt. **Efficient Sensor Network Reprogramming through Compression of Executable Modules**. In *Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON '08. 5th Annual IEEE Communications Society Conference on*, pages 359 –367, june 2008. 20

[45] Thanos Stathopoulos, John Heidemann, and Deborah Estrin. **A Remote Code Update Mechanism for Wireless Sensor Networks**. Technical Report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing, November 2003. 22, 23, 25, 26, 29

[46] Jonathan W. Hui and David Culler. **The dynamic behavior of a data dissemination protocol for network programming at scale**. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 81–94, New York, NY, USA, 2004. ACM. 22, 23, 27, 29

[47] Limin Wang. **MNP: multihop network reprogramming service for sensor networks**. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 285–286, New York, NY, USA, 2004. ACM. 22, 23, 26, 29

[48] Charles Wang, Dean Sklar, and Diana Johnson. **Forward Error-Correction Coding**. *Crosslink The Aerospace Corporation magazine of advances in aerospace technology*, **3**(1):1–4, 2002. 23

[49] Michael Luby. **LT Codes**. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, FOCS '02, pages 271–, Washington, DC, USA, 2002. IEEE Computer Society. 23, 24, 219, 220, 221, 231

[50] A. Shokrollahi. **Raptor codes**. *IEEE Transactions on Information Theory*, **52**(6):2551 –2567, june 2006. 23, 219

[51] Petar Maymounkov. **Online codes**, November 2002. New York Univerity Technical Report (TR2002-833). 23, 219

[52] M. Rossi, N. Bui, G. Zanca, L. Stabellini, R. Crepaldi, and M. Zorzi. **SYNAPSE++: Code Dissemination in Wireless Sensor Networks Using Fountain Codes**. *Mobile Computing, IEEE Transactions on*, **9**(12):1749 – 1765, dec. 2010. 24, 219, 237, 245, 253

[53] Andrew Hagedorn, David Starobinski, and Ari Trachtenberg. **Rateless Deluge: Over-the-Air Programming of Wireless Sensor Networks Using Random Linear Codes**. In *Proceedings of the 7th international conference on Information processing in sensor networks*, IPSN '08, pages 457–466, Washington, DC, USA, 2008. IEEE Computer Society. 24, 219

[54] I-Hong Hou, Yu-En Tsai, T.F. Abdelzaher, and I. Gupta. **AdapCode: Adaptive Network Coding for Code Updates in Wireless Sensor Networks**. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 1517 –1525, april 2008. 24, 219

[55] Anthony D. Wood and John A. Stankovic. **Online Coding for Reliable Data Transfer in Lossy Wireless Sensor Networks**. In *Proceedings of the 5th IEEE International Conference on Distributed Computing in Sensor Systems*, DCOSS '09, pages 159–172, Berlin, Heidelberg, 2009. Springer-Verlag. 24, 219

[56] Sze-Yao Ni, Yu-Chee Tseng, Yuh-Shyan Chen, and Jang-Ping Sheu. **The broadcast storm problem in a mobile ad hoc network**. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, MobiCom '99, pages 151–162, New York, NY, USA, 1999. ACM. 24

[57] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. **A reliable multicast framework for light-weight sessions and application level framing**. *IEEE/ACM Trans. Netw.*, **5**(6):784–803, December 1997. 25

[58] Philip Levis and David Culler. **The firecracker protocol**. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, EW 11, New York, NY, USA, 2004. ACM. 26

[59] Philip Levis, Neil Patel, David Culler, and Scott Shenker. **Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks**. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association. 27

[60] Sandeep S. Kulkarni and Mahesh Arumugam. **Infuse: A TDMA Based Data Dissemination Protocol for Sensor Networks**. *IJDSN*, **2**(1):55–78, 2006. 27

[61] Vinayak Naik, Anish Arora, Prasun Sinha, and Hongwei Zhang. **Sprinkler: A Reliable and Energy Efficient Data Dissemination Service for Wireless Embedded Devices**. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, RTSS '05, pages 277–286, Washington, DC, USA, 2005. IEEE Computer Society. 28

[62] Klaus Meier, Alban Hessler, Osman Ugus, Jörg Keller, and Dirk Westhoff. **Multi-Hop Over-The-Air Reprogramming of Wireless Sensor Networks using Fuzzy Control and Fountain Codes**. In *Somsed 2009. Workshop of Self-Organising Wireless Sensor and Communication Networks*, October 2009. 29

[63] Rajesh Krishna Panta". *"Remote reprogramming of wireless sensor networks"*. PhD thesis, Purdue University, May 2010. 29

[64] Guillermo Barrenetxea, François Ingelrest, Gunnar Schaefer, and Martin Vetterli. **The hitchhiker's guide to successful wireless sensor network deployments**. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 43–56, New York, NY, USA, 2008. ACM. 30

[65] Wenyuan Xu, Ke Ma, W. Trappe, and Yanyong Zhang. **Jamming sensor networks: attack and defense strategies**. *Network, IEEE*, **20**(3):41 – 47, may-june 2006. 33

[66] Mingyan Li, I. Koutsopoulos, and R. Poovendran. **Optimal Jamming Attack Strategies and Network Defense Policies in Wireless Sensor Networks**. *Mobile Computing, IEEE Transactions on*, **9**(8):1119 – 1133, aug. 2010. 33

REFERENCES

[67] A. Proa ando and L. Lazos. % bf Selective Jamming Attacks in Wireless Networks. In *Communications (ICC), 2010 IEEE International Conference on*, pages 1 –6, may 2010. 33

[68] Carl Hartung, James Balasalle, and Richard Han. **Node Compromise in Sensor Networks: The Need for Secure Systems**. Technical report, CU-CS-990-05, University of Colorado, Boulder, January 2005. 33, 93, 238

[69] An Liu and Peng Ning. **TinyECC: Elliptic Curve Cryptography for Sensor Networks (Version 1.0)**, 2007. 38, 161, 214, 215

[70] Benedik Driessen, Axel Poshmann, and Christof Paar. **Coparison of Innovative Signature Algorithms for WSNs**. In *WiSec '08: ACM Conference on Wireless Network Security*. ACM, 2008. 38, 163

[71] Ralph C. Merkle. **A certified digital signature**. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 218–238, New York, NY, USA, 1989. Springer-Verlag. 39, 164, 165

[72] Prabal K. Dutta, Jonathan W. Hui, David C. Chu, and David E. Culler. **Securing the deluge Network programming system**. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, pages 326–333, New York, NY, USA, 2006. ACM. 40, 195, 206, 213

[73] Patrick E. Lanigan, Rajeev Gandhi, and Priya Narasimhan. **Sluice: Secure Dissemination of Code Updates in Sensor Networks**. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 53, Washington, DC, USA, 2006. IEEE Computer Society. 40, 195, 206, 213

[74] Jing Deng, Richard Han, and Shivakant Mishra. **Secure code distribution in dynamically programmable wireless sensor networks**. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, pages 292–300, New York, NY, USA, 2006. ACM. 40, 195, 197, 213

[75] Sangwon Hyun, Peng Ning, An Liu, and Wenliang Du. **Seluge: Secure and DoS-Resistant Code Dissemination in Wireless Sensor Networks**. In *IPSN'08: International Conference on Information Processing in Sensor Networks*. IEEE Computer Society, April 2008. 40, 195, 197, 207, 213, 214, 215, 216

[76] Joseph Polastre, Robert Szewczyk, and David Culler. **Telos: enabling ultra-low power wireless research**. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press. 47

[77] **Telos (Rev B) Datasheet, Crossbow**. 48

[78] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. **TinyOS: An Operating System for Sensor Networks**. In Werner Weber, Jan M. Rabaey, and Emile Aarts, editors, *Ambient Intelligence*, pages 115–148. Springer Berlin Heidelberg, 2005. 49, 50, 51, 52, 53, 213

[79] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. **Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors**. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society. 49, 50, 51, 52, 53

[80] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. **MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms**. *Mob. Netw. Appl.*, **10**(4):563–579, August 2005. 49, 50, 51, 52, 53

[81] Anand Eswaran, Anthony Rowe, and Raj Rajkumar. **Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks**. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, RTSS '05, pages 256–265, Washington, DC, USA, 2005. IEEE Computer Society. 49, 50, 51, 52, 53

[82] Qing Cao, Tarek Abdelzaher, John Stankovic, and Tian He. **The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks**. In *Proceedings of the 7th international conference on Information processing in sensor networks*, IPSN '08, pages 233–244, Washington, DC, USA, 2008. IEEE Computer Society. 49, 50, 51, 52, 53

[83] Muhammad Omer Farooq and Thomas Kunz. **Operating Systems for Wireless Sensor Networks: A Survey**. *Sensors*, **11**(6):5900–5930, 2011. 49, 53

[84] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. **The nesC language: A holistic approach to networked embedded systems**. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 1–11, New York, NY, USA, 2003. ACM. 53

[85] Karsten Walther and Jorg Nolte. **A Flexible Scheduling Framework for Deeply Embedded Systems**. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops - Volume 01*, AINAW '07, pages 784–791, Washington, DC, USA, 2007. IEEE Computer Society. 53

[86] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. **TinyDB: an acquisitional query processing system for sensor networks**. *ACM Trans. Database Syst.*, **30**(1):122–173, March 2005. 53

[87] Chris Karlof, Naveen Sastry, and David Wagner. **TinySec: a link layer security architecture for wireless sensor networks**. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 162–175, New York, NY, USA, 2004. ACM. 53

[88] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. **TOSSIM: accurate and scalable simulation of entire TinyOS applications**. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, pages 126–137, New York, NY, USA, 2003. ACM. 54

[89] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. **Avrora: scalable sensor network simulation with precise timing**. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press. 54

[90] Ernst L. Leiss. *A Programmer's Companion to Algorithm Analysis*. Chapman & Hall/CRC, 2006. 56

[91] **AVR035: Efficient C coding for AVR, Atmel**. 56

[92] Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, and Adi Shamir. **Key Recovery Attacks of Practical Complexity on AES-256 Variants with up to 10 Rounds**. In *EUROCRYPT*, pages 299–319, 2010. 64

[93] Osman Ugus, Dirk Westhoff, Ralf Laue, Abdulhadi Shoufuhan, and A. Sorin Huss. **Elliptic Curve Based Additive Homomorphic Encryption in Wireless Sensor Networks**. In *Proceedings of the 2nd Workshop on Embedded Systems Security (WESS) under IEEE/ACM Conference on Embedded Systems Software (EMSOFT)*, WESS '07, Salzburg, Austria, 2007. 68

[94] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. **Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs**. In *CHES*, pages 119–132, 2004. 68

[95] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. **Efficient Elliptic Curve Exponentiation Using Mixed Coordinates**. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology (ASIACRYPT98)*, **1514** of *Lecture Notes in Computer Science*, pages 51–65. Springer Berlin/Heidelberg, 1998. 69

[96] Michael Howard and David LeBlanc. *Writing Secure Code*. Microsoft Press, 2003. 76

[97] **The STRIDE Threat Model, Microsoft**. http://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx. 79

[98] Chris Salter, O. Sami Saydjari, Bruce Schneier, and Jim Wallner. **Toward a secure system engineering methodolgy**. In *Proceedings of the 1998 workshop on New security paradigms*, NSPW '98, pages 2–10, New York, NY, USA, 1998. ACM. 80

[99] Wade Trappe and Lawrence C. Washington. *Introduction to Cryptography with Coding Theory (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005. 132

[100] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC Press, 2007. 133, 135, 137, 145, 146, 148, 150, 158, 160, 244

[101] Bart Preenel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke Universiteit Leuven, February 1993. `http://www.esat.kuleuven.ac.be/~preneel/phd_preneel_feb1993.pdf`. 135, 136, 137, 148

[102] David Chaum, Eugène van Heijst, and Birgit Pfitzmann. **Cryptographically Strong Undeniable Signatures, Unconditionally Secure for the Signer**. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '91, pages 470–484, London, UK, UK, 1992. Springer-Verlag. 137

[103] Ralph C. Merkle. **One way hash functions and DES**. In *Proceedings on Advances in cryptology*, CRYPTO '89, pages 428–446, New York, NY, USA, 1989. Springer-Verlag New York, Inc. 137

[104] Ivan Bjerre Damgård. **A design principle for hash functions**. In *Proceedings on Advances in cryptology*, CRYPTO '89, pages 416–427, New York, NY, USA, 1989. Springer-Verlag New York, Inc. 137

[105] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002. 139, 240

[106] **Specification for the Advanced Encryption Standard (AES)**. Federal Information Processing Standards Publication 197, 2001. 139, 240

[107] Bart Prenel and Paul C. van Oorschot. **MDx-MAC and Building Fast MACs from Hash Functions**. In *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '95, pages 1–14, London, UK, UK, 1995. Springer-Verlag. 146, 151

[108] Xiaoyun Wang and Hongbo Yu. **How to break MD5 and other hash functions**. In *Proceedings of the 24th annual international conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'05, pages 19–35, Berlin, Heidelberg, 2005. Springer-Verlag. 146

[109] Bart Preneel, René Govaerts, and Joos Vandewalle. **Cryptographic hash functions: an overview**. In *Proceedings of the 6th International Computer Security and Virus Conference (ICSVC 1993)*, page 19, New York,NY,USA, 1993. 148

[110] John Black and Phillip Rogaway. **CBC MACs for Arbitrary-Length Messages: The Three-Key Constructions**. *J. Cryptol.*, **18**(2):111–131, April 2005. 149

[111] Tetsu Iwata and Kaoru Kurosawa. **OMAC: One-Key CBC MAC**. In *In Proceedings of the 10th International Workshop on the Fast Software Encryption (FSE)*, pages 129–153, 2003. 149

[112] Mihir Bellare, Joe Kilian, and Phillip Rogaway. **The security of the cipher block chaining message authentication code**. *J. Comput. Syst. Sci.*, **61**(3):362–399, December 2000. 149

[113] John Black and Phillip Rogaway. **A Block-Cipher Mode of Operation for Parallelizable Message Authentication**. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology*, EUROCRYPT '02, pages 384–397, London, UK, UK, 2002. Springer-Verlag. 149

[114] Douglas R. Stinson. *Cryptography: Theory and Practice, Third Edition*. CRC/C&H, 3rd edition, 2006. 150

[115] S. Bakhtiari, R. Safavi-naini, J. Pieprzyk, and Centre Computer. **Cryptographic Hash Functions: A Survey**. Technical report, University of Wollongong, Australia, 1995. 150

REFERENCES

[116] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. **Keying Hash Functions for Message Authentication**. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 1–15, London, UK, UK, 1996. Springer-Verlag. 150

[117] Markus Jakobsson, Tom Leighton, Silvio Micali, and Michael Szydlo. **Fractal Merkle Tree Representation and Traversal**. In *Topics in Cryptology - CT-RSA 2003: The Cryptographers' Track at the RSA Conference 2003*, pages 314–326. Springer, 2003. 165

[118] Michael Szydlo. **Merkle Tree Traversal in Log Space and Time**. In *Advances in Cryptology - EUROCRYPT'04*, pages 541–554. Springer, 2004. 165

[119] Piotr Berman, Marek Karpinski, and Yakov Nekrich. **Optimal trade-off for Merkle tree traversal**. *Theor. Comput. Sci.*, **372**(1):26–36, 2007. 165

[120] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001. 172

[121] Y. Challal, H. Bettahar, and A. Bouabdallah. **A taxonomy of multicast data origin authentication: Issues and solutions**. *Commun. Surveys Tuts.*, **6**(3):34–57, July 2004. 173, 174, 175, 177, 191, 192

[122] Christophe M. A. Tartary. *Authentication for Multicast Communication*. Macquarie University, 2007. 174, 192

[123] Dan Boneh, Glenn Durfee, and Matthew K. Franklin. **Lower Bounds for Multicast Message Authentication**. In *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques: Advances in Cryptology*, EUROCRYPT '01, pages 437–452, London, UK, UK, 2001. Springer-Verlag. 175, 181

[124] Y. Desmedt, Y. Frankel, and M. Yung. **Multi-receiver/multi-sender network security: efficient authenticated multicast/feedback**. In *INFOCOM '92. Eleventh Annual Joint Conference of the IEEE Computer and Communications Societies, IEEE*, pages 2045 –2054 vol.3, may 1992. 177, 179, 180, 192

[125] Adi Shamir. **How to share a secret**. *Commun. ACM*, **22**(11):612–613, November 1979. 177

[126] R. Safavi-Naini and H. Wang. **New results on multi-receiver authentication codes**. In Kaisa Nyberg, editor, *Advances in Cryptology EUROCRYPT'98*, **1403** of *Lecture Notes in Computer Science*, pages 527–541. Springer Berlin Heidelberg, 1998. 180, 181, 192

[127] Ran Canetti, Juan Garay, Gene Itkis, Daniele Micciancio, Moni Naor, and Benny Pinkas. **Multicast Security: A Taxonomy and Some Efficient Constructions**. In *Proc. IEEE INFOCOM'99*, **2**, pages 708–716, New York, NY, March 1999. IEEE. 181, 183, 203, 207

[128] Adrian Perrig, J. D. Tygar, Dawn Song, and Ran Canetti. **Efficient Authentication and Signing of Multicast Streams over Lossy Channels**. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP '00, pages 56–, Washington, DC, USA, 2000. IEEE Computer Society. 184, 191

[129] Adrian Perrig, Ran Canetti, J.D̃. Tygar, and Dawn Song. **The TESLA Broadcast Authentication Protocol**. *RSA CryptoBytes*, **5**(Summer), 2002. 184

[130] Don Coppersmith and Markus Jakobsson. **Almost optimal hash sequence traversal**. In *Proceedings of the 6th international conference on Financial cryptography*, FC'02, pages 102–119, Berlin, Heidelberg, 2003. Springer-Verlag. 187

[131] Francesco Bergadano, Davide Cavagnino, and Bruno Crispo. **Individual Single Source Authentication on the MBone.** In *IEEE International Conference on Multimedia and Expo (I)*, pages 541–544, 2000. 187

[132] Francesco Bergadano, Davide Cavagnino, and Bruno Crispo. **Chained Stream Authentication**. In *Proceedings of the 7th Annual International Workshop on Selected Areas in Cryptography*, SAC '00, pages 144–157, London, UK, UK, 2001. Springer-Verlag. 187

[133] Adrian Perrig, Robert Szewczyk, J. D. Tygar, Victor Wen, and David E. Culler. **SPINS: security protocols for sensor networks**. *Wirel. Netw.*, **8**(5):521–534, September 2002. 187

[134] Patrick Schaller, Srdjan Čapkun, and David Basin. **BAP: Broadcast Authentication Using Cryptographic Puzzles**. In *Proceedings of the 5th international conference on Applied Cryptography and Network Security*, ACNS '07, pages 401–419, Berlin, Heidelberg, 2007. Springer-Verlag. 188

[135] Donggang Liu and Peng Ning. **Multilevel μTESLA: Broadcast authentication for distributed sensor networks**. *ACM Trans. Embed. Comput. Syst.*, **3**(4):800–836, November 2004. 188

[136] Donggang Liu, Peng Ning, Sencun Zhu, and Sushil Jajodia. **Practical Broadcast Authentication in Sensor Networks**. In *Proceedings of the The Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, MOBIQUITOUS '05, pages 118–132, Washington, DC, USA, 2005. IEEE Computer Society. 188

[137] Roberto Di Pietro, Luigi V. Mancini, Antonio Durante, and Vishwas Patil. **Addressing the shortcomings of one-way chains**. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, ASIACCS '06, pages 289–296, New York, NY, USA, 2006. ACM. 188

[138] Giuseppe Ateniese and Breno Medeiros. **Identity-Based Chameleon Hash and Applications**. In Ari Juels, editor, *Financial Cryptography*, **3110** of *Lecture Notes in Computer Science*, pages 164–180. Springer Berlin Heidelberg, 2004. 188

[139] Philippe Golle and Nagendra Modadugu. **Authenticating streamed data in the presence of random packet loss**. In *Proc. of the Symposium on Network and Distributed Systems Security (NDSS)*, pages 13–22, 2001. 191

[140] Y. Challal, H. Bettahar, and A. Bouabdallah. **A2cast: an adaptive source authentication protocol for multicast streams**. In *Computers and Communications, 2004. Proceedings. ISCC 2004. Ninth International Symposium on*, **1**, pages 363 – 368 Vol.1, june-1 july 2004. 191

[141] Yacine Challal, Hatem Bettahar, and Abdelmadjid Bouabdallah. **Hybrid and Adaptive Hash-Chaining Scheme for Data-Streaming Source Authentication**. In Zoubir Mammeri and Pascal Lorenz, editors, *High Speed Networks and Multimedia Communications*, **3079** of *Lecture Notes in Computer Science*, pages 1056–1067. Springer Berlin / Heidelberg, 2004. 191

[142] Yacine Challal, Abdelmadjid Bouabdallah, and Hatem Bettahar. % bf H2A: Hybrid Hash-chaining scheme for Adaptive multicast source authentication of media-streaming. *Computers and Security*, **24**(1):57 – 68, 2005. 191

[143] Yacine Challal, Abdelmadjid Bouabdallah, and Yoann Hinard. **RLH: receiver driven layered hash-chaining for multicast data origin authentication**. *Comput. Commun.*, **28**(7):726–740, May 2005. 191

[144] Jonathan W. Hui and David Culler. **The dynamic behavior of a data dissemination protocol for network programming at scale**. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM. 195, 231, 243

[145] **TinyOS: An open-source operating system designed for wireless embedded sensor networks**, 2007. 213

[146] Jens-Matthias Bohli, Alban Hessler, Klaus Meier, Osman Ugus, and Dirk Westhoff. **Dependable Over-the-Air Programming**. *Adhoc & Sensor Wireless Networks*, **13**(3-4):313 – 340, 2011. 219, 225, 253

[147] W. A. Stein et al. *Sage Mathematics Software*. The Sage Development Team, 2008. 230

[148] Esa Hyytiä, Tuomas Tirronen, and Jorma Virtamo. **Optimizing the Degree Distribution of LT codes with an Importance Sampling Approach**. In *6th International Workshop on Rare Event Simulation, RESIM 2006*, 2006. 231

[149] Johann Großschädl, Stefan Tillich, Christian Rechberger, Michael Hofmann, and Marcel Medwed. **Energy evaluation of software implementations of block ciphers under memory constraints**. In *DATE*, Nice, France, April 2007. 241

[150] Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, and Adi Shamir. **Key Recovery Attacks of Practical Complexity on AES Variants With Up To 10 Rounds**. Cryptology ePrint Archive, Report 2009/374, 2009. 244

Curriculum Vitae

Personal Information

Name:	Osman Ugus
Email:	`osman.ugus@ougus.de`

Education

Diploma in Computer Science	Oct. 2001 – Apr. 2007
Technische Universität Darmstadt	Germany
B.Sc. in Environmental Engineering	Oct. 1994 – Jul. 1998
Istanbul University	Turkey
Primary and High School	1983 – 1994
Nigde High School	Turkey

Employment

Research Assistant	Dec. 2010 – present
HAW Hamburg	Germany
Research Associate	May 2007 – Dec. 2010
NEC Europe Network Laboratories	Germany