KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

A Security Framework for Shared Networked Embedded Systems

Jef Maerien

Supervisor:
Prof. dr. ir. W. Joosen

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

19 June 2015

A Security Framework for Shared Networked Embedded Systems

Jef MAERIEN

Examination committee:
Prof. dr. C. Vandecasteele, chair
Prof. dr. ir. W. Joosen, supervisor
Prof. dr. ir. C. Huygens
Prof. dr. ir. B. Preneel
Dr. S. Michiels
Prof. dr. D. Hughes
Prof. dr. J. Davis
Prof. dr. C. Blondia
(University of Antwerp)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

19 June 2015

© 2015 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Jef Maerien, Celestijnenlaan 200A box 2402, B-3001 Heverlee (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Preface

This work would not have been possible without the aid and support of many. I would like to take this opportunity to express my gratitude towards:

- My promotor Wouter Joosen for giving me the opportunity to do this PhD.

- My daily coaches Danny Hughes, Christophe Huygens and Sam Michiels for all the feedback you provided over the years.

- The other members of my examination committee: Carlo Vandecasteele, Bart Preneel, Jesse Davis, and Chris Blondia for the interesting discussion.

- The Agency for Innovation by Science and Technology in Flanders (IWT) for funding four of the six years of research.

- The project office, system group, and secretariat, for their much appreciated help and administrative support.

- My colleagues in the Networked Embedded Systems task force, for the fruitful collaborations.

- My colleagues at DistriNet and at the Computer Science department, for the interesting meetings at and after work.

- The PhD reception networking society, for the regular break from research.

- My friends, for distracting my mind during our pool/board/card/computer game evenings.

- My parents, Brigitte and Staf, for enabling and encouraging me to follow my dreams and ambitions.

- My brother Jan, aunt Maria, and all other aunts, uncles, nieces, and nephews for their continued interest and support.

- My girlfriend Tatiana, for more things than I can possibly write in this dissertation.

Jef Maerien
June 2015

Abstract

Networked embedded systems are slowly gaining more acceptance in mainstream applications. This causes a shift in the paradigms regarding research and development. While classic research often focused on single-purpose and single-owner networks, we are seeing that modern applications no longer adhere to these paradigms. Rather embedded networks will consist of devices owned by many different parties, and these devices will need to be accessible by many different users. As such, there is a clear need for security in these networked embedded systems. This thesis aims to answer two questions: (1) what are the necessary abstractions required by the different stakeholders to express their security policies and requirements, and (2) what is the minimal required infrastructure to secure all crucial multi-party interactions, with as little overhead as possible.

While classic approaches for both embedded and back-end systems provide a useful starting point, we do see significant gaps in the work for both abstractions and security infrastructure. Current related work in networked embedded systems provides several abstractions to model the underlying platform or the data produced, but none to actually model the security policies or requirements. There are some generic security policy declaration languages, which provide a way to express data, but do not provide the necessary abstractions to enable systems to operate. While looking at available infrastructure, we see that related work mainly proposes point solutions for specific types of interactions, but does not provide a holistic infrastructure securing all crucial multi-party interactions. Second, related work for resource constrained embedded networks still only focus on single-owner networks, while research for multi-owner embedded networks always uses resource intensive algorithms to ensure security.

The two key contributions presented in this thesis are: (1) a set of abstractions that provides the stakeholders with the necessary means to express their security policies and requirements, and (2) a security middleware that secures the multi-party interactions driven by the previously mentioned policies. These contributions together from the SecLooCI security middleware framework. To

develop the abstractions, we defined three stakeholder roles: the application owner, the platform owner, and the network owner. For each of these three roles, we designed a set of abstractions to express their relevant security requirements and policies. These abstraction then drive the security middleware, which consists of five sub-systems, each of which secures one of the 5 key multi-party interactions: (1) network join, (2) application deployment, (3) node service usage, (4) application communication, and (5) application monitoring. These five sub-systems, together with the policy abstractions, form the SecLooCI framework, which provides the necessary functionality for all stakeholders to secure their systems, and offers clear points of modification to allow application specific customisation.

These contributions are validated on state-of-the-art constrained embedded platforms, in the form of an integrated prototype implementation. This implementation is validated through a number of representative scenarios derived from industry collaborations in the domain of smart logistics and smart offices. Additionally these contributions were published into multiple peer-reviewed venues. These results show that the SecLooCI middleware is able to secure multi-party ecosystems on current constrained embedded platforms.

Beknopte samenvatting

Genetwerkte ingebedde systemen worden langzaam aan meer en meer gebruikt in mainstream applications. Dit zorgt voor een verschuiving in de paradigma's met betrekking tot onderzoek en ontwikkeling. Terwijl klassiek onderzoek zich vooral richt op networken met één eigenaar en één functie, zien we dat moderne applicaties deze paradigma's niet meer volgen. Genetwerkte ingebedde systemen zullen daarentegen bestaan uit genetwerkte platformen van vele verschillende eigenaars, die door meerdere verschillende gebruikers gebruikt zullen worden. Dit leidt duidelijk tot een grote noodzaak voor beveiliging. Deze thesis probeert dan ook de volgende twee vragen te beantwoorden: (1) wat zijn de noodzakelijk abstracties voor de verschillende belanghebbenden om hun beveiligingsrichtlijnen en vereisten uit te drukken, en (2) wat is de noodzakelijke infrastructuur om alle cruciale multi-party interacties te beveiligen, met zo weinig mogelijk belasting van de platformen.

Hoewel de klassieke kijk op zowel ingebedde systemen als serversystemen een nuttig startpunt biedt, zien we toch significante lacunes in het gerelateerde werk voor zowel abstracties als beveiligingsinfrastructuur. Het huidige gerelateerde werk in genetwerkte ingebedde system biedt verschillende abstracties aan die toelaten om het onderliggende platform of de geproduceerde data en context te modelleren, maar bieden geen abstracties aan die toelaten de beveiligingsvoorschriften of vereisten te modelleren. In een server omgeving zijn er verschillende generische talen die toelaten om algemene beveiligingsvereisten te encoderen, maar deze bieden geen specifieke abstracties en modellen aan om genetwerkte ingebedde systemen te beheren. Als we kijken naar de beschikbare infrastructuur, zien we dat het huidige werk vooral puntoplossingen aanbiedt, terwijl we nood hebben aan een hollistische architectuur die alle multi-partij interacties beveiligd. Daarnaast focust het huidige gerelateerde werk in systemen met beperkte hulpbronnen zich vooral op networken met slecht één eigenaar, terwijl onderzoek naar ingebedde netwerken met meerdere gebruikers en partijen typisch altijd gebruik maakt van dure algoritmen.

De twee onderzoeksbijdragen, die deze thesis presenteert, zijn: (1) een set abstracties die de verschillende belanghebbenden de noodzakelijke instrumenten geeft om hun beveilingsbeleid en -vereisten te modelleren, en (2) een beveiligingsmiddleware die de multi-partij interacties beveiligd op basis van de vermeldde vereisten. Deze bijdragen vormen samen het SecLooCI beveiligingsraamwerk. Om de abstracties te ontwikkelen, identificeerde we de drie rollen die de belanghebbenden kunnen aannemen: (1) de applicatie eigenaar, (2) de platform eigenaar, en (3) de netwerk eigenaar. Voor elk van deze drie rollen ontwikkelden we een set van abstracties dat hen toelaat hun beveiligingsvereisten en beleid te modelleren. De beveiligings middleware bestaat uit vijf sub-systemen, die elks een van de vijf cruciale multi-partij interacties beveiligd: (1) verbinden met een netwerk, (2) installeren van een applicatie, (3) gebruiken van diensten, (4) applicatie communicatie, en (5) opvolging van middelenverbruik. Deze vijf subsystemen samen met de beleidsabstracties vormen het SecLooCI raamwerk, dat de noodzakelijke functionaleit aanbiedt aan alle belanghebben om hun systemen te beveiligingen, en hun toelaat om het systeem aan te passen aan hun specifieke situatie door middel van duidelijk gedefinieerde uitbreidingspunten.

Deze bijdragen zijn gevalideerd door van middel een geintegreerde implementatie van een prototype, academische publicaties, en via onderzoeksprojecten. Het prototype toont aan dat de beveilingsmiddleware kan geinstalleerd worden op moderne ingebedde hardware. De resultaten en principes van dit onderzoek zijn academisch gevalideerd door middel van meerdere publicaties met beoordeling door vakgenoten. Tot slotte zijn deze resultaten gepresenteerd binnen meerdere onderzoeksprojecten in het domein van de intelligente logistiek en het intelligente kantoor. Dit toont aan dat het SecLooCI beveiligingsraamwerk een bruikbaar en noodzakelijk instrument is ter beveiliging van toekomste multi-partij ingebedde ecosystemen.

Contents

Abstract iii

Contents vii

List of Figures xiii

List of Tables xvii

1 Introduction 1

1.1 Networked Embedded Systems 2

1.2 The Embedded Device 3

1.3 The Embedded Application 5

1.4 Open Issues 6

1.5 Problem Statement 8

1.6 Contributions 10

1.7 Overview of Thesis 11

2 Context 13

2.1 Use Cases 13

2.1.1 Smart Office 14

2.1.2 Smart Logistics 15

2.2 Stakeholders 16

2.2.1 Platform Owner (PO) 17

2.2.2 Network Owner (NO) 18

2.2.3 Application Owner (AO) 18

2.3 Sensor Network Application Lifecycle 18

2.4 Problem Statement 23

2.4.1 Security Management Abstractions 23

2.4.2 Secure Node System 26

2.5 Requirements 29

2.5.1 Attacker Model 29

2.5.2 Non Functional Requirements 31

2.6 Summary 32

3 Related Work **35**

3.1 Management Abstractions 35

3.1.1 Security Management Abstractions 35

3.1.2 Application and System Abstractions 37

3.2 Secure Node Systems 39

3.2.1 Secure Network Initialisation 39

3.2.2 Secure Application Deployment 43

3.2.3 Secure Application Communication 44

3.2.4 Secure Application Management 46

3.2.5 Secure Service Usage 48

3.3 Summary 49

4 Architecture **51**

4.1 End-User Tool 51

4.2 Application Owner Server 53

4.3 Platform Owner Server 55
4.4 Network Owner Gateway 56
4.5 Embedded Node 58
4.5.1 LooCI Node Architecture 59
4.5.2 Securing the Node Architecture 61
4.6 Example Scenario 66
4.6.1 Policy Declaration 66
4.6.2 Application Creation / Acquisition 68
4.6.3 Network Setup 68
4.6.4 Application Instantiation 69
4.6.5 Application Enactment 70
4.6.6 Application Runtime 70
4.6.7 Application Removal 72
4.7 Summary 72

5 Security Management Abstractions 75

5.1 Platform Owner Model 75
5.2 Application Owner Model 79
5.3 Network Owner Model 82
5.4 Evaluation and Discussion 83
5.4.1 Logistics Scenario 83
5.4.2 User Effort 84
5.4.3 Modeling Language Evaluation 86
5.4.4 Verification Evaluation 87
5.4.5 Discussion 88
5.5 Summary 90

6 Security Middleware 91

6.1 Underlying Platform and Assumptions 91
6.1.1 Underlying Platform 92
6.2 Secure Network Initialisation 95
6.3 Secure Application Deployment 99
6.4 Secure Application Management 103
6.4.1 Authentication 104
6.4.2 Authorisation 105
6.4.3 User Management Service 107
6.4.4 Implementation Details 108
6.5 Secure Application Communication 111
6.6 Monitoring and Enforcement 113
6.7 Integrated Security Framework 117
6.8 Summary 119

7 Case Studies 121

7.1 Smart Logistics Scenario 121
7.1.1 Test Scenario and Measurements 121
7.1.2 Summary 124
7.2 Smart Office Deployment 125
7.2.1 Applications 125
7.2.2 Security Requirements 127
7.2.3 Evaluation Goal 127
7.2.4 Metrics 128
7.2.5 Observations 134
7.3 Summary 135

8 Discussion 137

8.1 Non Functional Requirements Discussion 137

8.1.1 Evolvability of Infrastructure 137
8.1.2 Evolvability of Software 138
8.1.3 Heterogeneity of Infrastructure 139
8.1.4 Transparency of Heterogeneity 139
8.1.5 Transparency of Security 140
8.1.6 Flexibility of Communication 140
8.1.7 Flexibility of Security 141
8.2 Security Discussion 142
8.2.1 Outside Attacker 142
8.2.2 Network Attacker 143
8.2.3 Physical Attacker 144
8.2.4 Internal Attacker 145
8.2.5 Other Threats 145
8.2.6 Summary 146
8.3 General Discussion 146
8.3.1 Trust Requirements 146
8.3.2 Privacy 147
8.3.3 Energy 148
8.3.4 Limitations and Trade-offs 150
8.3.5 Security Framework 151
8.4 Summary 152

9 Conclusion 153

9.1 Summary and Contributions 153
9.2 Lessons Learned 156
9.3 Future Work 158
9.3.1 Secure Underlying Environment 158

9.3.2 End-user Support 159
9.4 Outlook 161

Bibliography **163**

List of Figures

2.1 Overview of the three roles present in the NES ecosystem. . . . 17

2.2 Overview of the lifecycle of a typical Networked Embedded System application. 19

2.3 Overview of the problems tackled in this thesis. To create secure networked embedded systems, two components are needed: users must have abstractions to declare their policies, and these policies must be enforced by secure node systems during the lifecycle of the application. This thesis will look at the required abstractions for each role, and at the node security systems to enforce these policies for those data flows that are impacted by sharing. . . . 24

4.1 Architecture of the distributed security infrastructure. 52

4.2 Screenshot of the current implementation of the end-user tool. 52

4.3 Overview of the Application Owner Server. 54

4.4 Overview of the Platform Owner Server. 55

4.5 Overview of the Network Owner Server. 57

4.6 Overview of the Basic Node Architecture. 58

4.7 Overview of the Secure Node Architecture. 60

4.8 Overview of the harbour scenario. 67

4.9 Example Localisation Application. 68

4.10 Network setup data flow. 69

4.11 Component deployment data flow. 70

4.12 Service management data flow. 71
4.13 Application communication data flow. 71
4.14 Service usage data flow. 72

5.1 Platform owner abstraction: node model. 77
5.2 Platform owner abstraction: application owner model. 78
5.3 Platform owner abstraction: network owner model. 79
5.4 Application owner abstraction: application specification model. 80
5.5 Application owner abstraction: deployment specification model. . 81
5.6 Network owner abstraction: network model. 82
5.7 Example of an on-board-unit installed in trucks. 84

6.1 Overview of the LooCI component middleware. 93
6.2 Secure network setup key exchange protocol. 95
6.3 Packet format of the MASY Hello message. Signature is signed, and group key is encrypted with the node unique key, only known to the sensor node and Platform Owner. 98
6.4 Overview of the Sasha Secure application deployment protocol. 99
6.5 Overview of the secure deployment token. The key is optional and used if the application owner required confidential deployment. Token overhead is thus either minimally 56 or 72 bytes. The token is transmitted encrypted using AES128 and the long term node key. The token can be extended with additional binary resource restriction policies. 102
6.6 Overview of the authentication and authorisation of a service access request. 104
6.7 Packet format of a SecLooCI management communication message. 108
6.8 Overview of the Application Management protocol. Colored parts are components added for security: authentication interceptor (orange), authorisation framework (yellow), and user management (green). 108
6.9 Packet format of a SecLooCI application communication message. 112

6.10 Flowchart of the decision tree of policy interpretation at sending and receiving application messages. 112

6.11 Overview of the monitoring and enforcement component. . . . 114

6.12 Overview of the Secure Node Architecture. 117

7.1 Example Localisation Application. 122

7.2 Overview of the applications of the DistriNet smart office deployment. 126

List of Tables

5.1 User effort to enact the system. Case one is a limited deployment of 3 assignments, case two is a larger deployment with 201 assignments. 85

5.2 User effort to change the system. 86

5.3 Overview of the sizes of different descriptors. 87

6.1 LooCI and Contiki memory usage. 94

6.2 Secure application deployment memory overhead. 103

6.3 Overview of the permissions of the different users with the different parties and their applications. 106

6.4 Secure application management memory overhead. 110

6.5 Secure application communication memory overhead. 113

6.6 SecLooCI framework memory overhead. The overhead of the security suite is comparable to the overhead of the non-secure middleware, but clearly still within the possibilities of a low power sensor node. 119

6.7 Detailed SecLooCI framework memory overhead. A significant amount of ROM and RAM is used for the encryption algorithms. Hardware implementations could reduce this overhead. 119

7.1 Overview of the SecLooCI framework overhead. Daily overhead calculated assuming one incoming and outgoing message per minute. Table shows that processing overhead is comparable to one day of operation. Transmission overhead is equal to about 19 days of operation. 124

7.2 Hourly application usage. 130

7.3 Hourly management usage. 131

7.4 Monitoring overhead per hour. 132

7.5 Operating system overhead. 133

Chapter 1

Introduction

The past 25 years have seen a transformation in computing devices from big immovable machines that often had very limited connectivity, to devices that can be carried everywhere, used anytime and continuously communicate with many other systems all over the world. This evolution will not stop. Computing devices are becoming even more ubiquitous, and computing platforms are being embedded in virtually every thing we know and see, connected with other devices locally in the network, and globally across the Internet. Current electronics, such as TVs, fridges, washing machines, smoke detectors, etc. are being equipped with network capabilities, allowing more integration between the digital and real world. Devices are being deployed inside buildings [68] and even inside humans [44], to monitor their current condition, and in some cases to even enact changes in the environment, such as for example a pace-maker. This is the domain of Networked Embedded Systems (NES) research, or what is often called the Internet of Things.

Clearly there are vast opportunities for these tiny computing devices, but with these vast opportunities also come new threats. Being able to turn off a coffee machine remotely can be useful, someone else being able to remotely turn on my coffee machine is much less so. In case of a coffee machine, the risk is low. However, the technology to turn on a coffee machine remotely, is very similar to remotely telling a pacemaker to deliver a mortal shock [61]. Similarly, being able to tell when someone is in my house when I am away is useful. Someone else learning that I am not at home, makes my home suddenly a very interesting target for burglars. There is a need to be able to limit and control access to these devices and the data they produce.

The remainder of this introduction lists some interesting current application

domains for networked embedded systems. The characteristics of the different devices and applications in such environments are then classified, followed by an overview of the main open issues, to conclude with the problem statement and contributions of this thesis.

1.1 Networked Embedded Systems

Networked embedded systems (NES) are used in a wide variety of areas. This section lists a few areas often identified as key areas for NES applications namely domotics, healthcare, research and industry.

A first application area with particular interest for end-users is **domotics** [123, 92]. The rise of smart phones and wireless networks enabled the creation of many gadgets which connect to these smart phones and computing infrastructure. Recent examples are smoke detectors, weight scales, radios and video cameras. The level of integration is currently pretty basic. All applications are created and deployed pretty much independently of each other, and often require custom software and equipment. But it is already possible to wirelessly interact with some electronics, and perform some basic kind of actuation. These current stove-pipe systems will start integrating with each other to a single integrated system. This can then offer a consistent end-user experience, and enables connecting different systems with each other to increase our quality of life.

NES also have a large potential to change **healthcare** [87, 44]. Past research has often suggested to have people wear monitoring equipment, that continuously tracks their body temperature, heart rate, blood pressure, movement, etc. The first generation of devices that perform these functions are coming to market, mostly for wear during sport activity such as the Polar and TomTom sports watches. However, the added value for continuous measurement could be great [109]. Research has shown that many problems can be detected much earlier if only the patient had been properly monitored. The early onset of many problems or illnesses are often invisible to the patient, but can be clearly seen in long term monitoring data. Especially for the elderly or disabled, this monitoring equipment has high promises. They enable an early detection and warning system for detecting falls, or cardiac arrest, allowing for life saving intervention to be triggered. Such technology will not only save a lot of suffering, but also allow the elderly to be more independent for longer, reducing cost and again increasing quality of life.

These networked embedded systems do not only change our daily life, but also the way we do **research**, especially of our environment [12]. Wireless battery powered sensors change the way researchers can observe the world. As every

good engineer knows: to measure is to know. But also as physicists know, to measure is to change. In the past, sensors were bulky and intrusive, prohibiting the easy, fast, and inconspicuous deployment needed to monitor nature and wildlife in their natural and unimpeded state. Small sensors have enabled researchers to measure many phenomena with previously unknown accuracy and breadth, with minimal disturbance of the environment [93, 33, 85]. The prototypical example in the past, is the monitoring of bird colonies at the Great Duck Island [76]. Here researchers deployed many small sensors in birds nests. This provided continuous long term measurements, without having to go to the island and disturb the birds, risking altering their behaviour. Many other research deployments have been done since, and many more will be done.

Not only in society can NES have a vast impact, but also in **industry**, there are vast opportunities for improvements [124, 43]. Live monitoring offers producers and transporters a previously unknown level of information. This information can be used to ensure quality through the entire production process, to more quickly detect and react to failures, and to optimise processes, leading to increased efficiency and decreased costs. A series of industry collaboration projects has provided a set of interesting use cases, such as container monitoring, industry automation, and harbour monitoring, which we will use throughout this dissertation as reference examples.

Clearly, networked embedded systems have vast possibilities in a wide variety of areas. However, what most of these use cases have in common, is that many use cases involve multiple people and multiple different heterogeneous devices working together, and have a need for security. For example, in office domotics solutions many users should be able to interact with the office systems. In healthcare systems, different doctors potentially need to interact with medical devices, in addition to patients themselves being able to read the devices. In industry, many different parties or administrative units often have to work together, and share data provided by these embedded monitoring and control systems. These cases also show large need for security. For domotics, healthcare, and industry, it is crucial that only authorised parties can interact with the node systems, and often only have clear and limited capabilities.

1.2 The Embedded Device

The embedded devices used in the applications can differ widely in many different dimensions.

The first dimension is **computational power**. Devices used in embedded systems span a wide range of computational power. The low end range consists

of the microcontrollers with around 10 MHz computational speed. Typical examples of this class are the Tmote Sky [99], Atmel Raven [3], and many Arduino devices. The next order of magnitude are the cheap microprocessor devices. In this class are the raspberry pi (700 MHz), and many current day smart phones. The next class of devices are the typical servers, which often have multiple GHz of processing speed, and can be embedded in cars, or work as central gateway in homes.

The second dimension is **energy**. Here two classes can be distinguished: limitless power, and limited power. The first class of devices have unlimited power, meaning they are not dependent on a battery, but are connected to the power grid. This allows them to operate indefinitely, but requires a constant physical connection, limiting mobility. The class of limited power contains those devices that are powered by battery. The smaller the battery, the more aware the device and applications have to be of unnecessary power consumption. There are advances on power harvesting [102], which allow these small devices to harvest power from the environment, for example from light, shocks, or temperature variations. This has the potential to significantly increase the lifetime of these wireless mobile devices.

The third dimension is **network connectivity**. Networked embedded systems are by definition networked with other devices. This connection can be wired or wireless. Most research looks at wireless devices, since this allows for greater mobility and flexibility. There are however many different and incompatible types of wireless communication: some currently popular wireless standards are WiFi [55], 3g [24], 4g/LTE [25], Bluetooth [8], Bluetooth Low Energy [40], RFID [116], NFC [117] and 802.15.4 wireless communication [79]. Each of these protocols differ in the range at which devices can communicate, energy cost, and ability to support many devices in a network. For in depth comparison we refer to related work.

The last dimension is **memory capacity**. The devices used in the mentioned applications can have a wide range of possible storage capabilities. On the low range are again the microcontrollers, which often support only about a dozen kilobytes of RAM memory, and about 100 kilobytes of ROM/flash memory. Due to this low storage capacity, it is necessary that these devices can communicate their measured data to other more capable devices for processing and storage. On the high end, we again find the smart phones and servers, which have multiple gigabytes or terabytes of data storage.

This thesis focuses on the minimal and most constrained case: small battery powered, mobile, low memory, and low computational power devices, that communicate with each other over the wireless network. This is the most constrained class, and as such the most difficult class to ensure security. This

class is often referred to as **Wireless Sensor Networks**. These minimalistic devices cost the least, which is naturally the most interesting for producers and buyers of embedded devices. Only if the cost is low enough, will most producers be willing to put additional computing devices in their products, and will consumers be willing to buy those products. Additionally, since this is the border case, we can extrapolate that all systems that are in any way more capable than the resource constrained systems, will be able to support all services that the more resource constrained version can support.

1.3 The Embedded Application

Over the past years, many Networked Embedded Systems (NES) applications have been deployed, mostly for research purposes. These deployments have evolved on different aspects. Previously, deployments were static, isolated entities with very little dynamism, managed and controlled by a single entity. Recently, the integration of smart phones, and increase in scale have caused much more integration and dynamism in these application. This section identifies four dimensions on which these applications have evolved over recent years.

A first dimension is the **single application focus**. Many organisations looked at NES systems as single application environments, where the system is deployed with a single static goal in mind. These systems typically monitor a limited number of variables such as temperature and humidity, or provide access control to a door. Multiple such systems might exist next to each other in a stove-pipe fashion, where each application is isolated from everything else. Networked embedded systems however will contain many different types of devices, that are horizontally integrated. Single embedded nodes and the control infrastructure need to integrate in a single large interconnected infrastructure, to enable highly customised interaction. A good example would be a person entering a building automatically causes his workspace to be adapted to his personal settings.

A second dimension is the **level of dynamism** in the systems. Dynamism is the level of change, which can be divided into hardware dynamism and software dynamism. Hardware dynamism is the amount of churn of devices in the network. Building networks can be considered fairly static. Devices are more or less fixed, and stay alive for a long time. Vehicle and other ad-hoc networks are more dynamic, where the neighbourhood of devices is constantly changing. Most deployed systems often start static, but evolve into more dynamic systems when more mobile nodes, such as smart phones, are assimilated in the larger system. Software dynamism is the evolution of the software running in the network. In current networks, the software running on the devices is continuously evolving:

new applications are being installed, existing applications are being upgraded, and old applications are removed. Current embedded systems sometimes ignore this aspect, but many foresee that software evolution will be a vital aspect of any networked embedded system deployment.

A third dimension is the **level of sharing**. The traditional research viewpoint of networked embedded systems is that systems are built, owned and maintained by a single party. Data and services are sometimes offered to other parties, but usually exclusively through back-end web services. However, the use cases show that networked embedded systems are evolving towards ecosystems comprised of devices owned by many parties, where these parties want direct access to the services provided by devices in the network.

A fourth dimension is the **required skillset** of the users of the NES application. In most NES deployments, the construction and maintenance of these systems requires specialists to put all required hardware components together, and then write the correct software for these devices. However to truly unlock the capabilities of these systems, the deployment and management frameworks must reach a level of usability so virtually anyone can create and maintain a NES. This also drives a specialisation, with clearly defined interfaces between different parties involved in the network.

This thesis focuses on multi-application, dynamic and shared systems, since we consider this the most general case. While this case is often simplified in literature, industrial use cases clearly show the need for dynamic systems, that support sharing at the device level. This thesis aims to increase the level of abstraction to allow specialisation, and decrease the complexity of managing the system for specialist network and system administrators. This thesis does not target an average consumer of electronics, but rather tries to create higher-level abstractions allowing domain experts to easily specify their security requirements. This provides another step towards the ultimate end goal of enabling future plug-and-play deployments, where anyone can set up and manage their own smart environment.

1.4 Open Issues

Currently there are still several open issues in the field of networked embedded systems. This section gives a short and general overview of current research issues from the network to the application layer, and some cross-cutting concerns. This list intends to give a brief summary of some of the major issues, and should not be considered complete.

The first issue is **the wireless network**. Current main-stream network solutions such as WiFi are considered impractical to run on embedded networks due to resource constraints and scale. These characteristics have caused renewed interest in optimising the wireless network for this class of devices. This ranges from physical radio optimisation [57], through medium access control protocols [110], to networking [21] and routing protocols [118]. These protocols are currently still in a prototype stage. However, to enable the integrated scenarios as previously described, it is necessary for one or a few common standards to emerge, in order to ensure interoperability at the network level.

The second issue is **the application paradigm**. Developing applications for embedded networks poses many challenges. This drives research to investigate what application development paradigms can be considered most interesting for these systems. The client server application paradigm for example has been optimised for constrained embedded networks in the form of the CoAP protocol standard. Other approaches apply event-based and component-based application paradigms to more easily develop scalable applications.

The third issue is **data storage and processing**. Instrumenting the environment produces vast amounts of continuous data, requiring significant storage space. To make this data useful however, it must be processed. Many researchers are looking at intelligent algorithms to extract knowledge and optimal settings from this vast amount of data. Due to the large variety in environments, there is currently not a single approach defined which works everywhere. It may even be impossible to create common algorithms to analyse and optimise all environment, requiring significant and specific future research for each topic.

The fourth issue is **system and application management**. Current management of large scale systems is often the work of very few, very skilled system and network administrators, who use highly customised scripts and systems to ensure consistency of their assets. Networked embedded systems will bring these large scale systems within our homes. However, most people are not computer experts, yet want their systems to work continuously, be upgraded to the latest version, and have significant control over the operation. How these people can be empowered is still an open issue.

The last and perhaps most important issue is **security**. Security is often cited as the crucial enabler in order for NES to receive widespread adoption in industry as well as the home [101]. Since NES applications will typically gather a lot of sensitive information, disclosure of such information offers clear and big risks regarding safety and privacy. When these systems are used for actuation, attackers can potentially apply lethal damage to people or goods [61]. In order to prevent and mitigate these threats, security must be a key concern in any

NES application. Constrained systems are currently unable to use the common security standards such as TLS. Additionally these standards are often not sufficient to fully secure the wide variety of interactions and systems present in NES applications.

This thesis primarily focuses on the securing Wireless Networked Embedded System applications, given the characteristics described previously. While a lot of related work exists on securing networks of constrained devices, it typically assumes a simplified, static, single-owner environment. As such it is unsuited for the complex multi-actor real-world environments. Since security is considered one of the key enablers of successful real-world applications, this thesis will look into expanding the current state of the art by identifying and creating the required infrastructure to allow the secure shared usage of dynamic constrained Networked Embedded Systems.

1.5 Problem Statement

The previous section looked at the different characteristics and open issues in Networked Embedded System deployments. It specified that this thesis will look at the secure sharing of dynamic constrained NES deployments. This section identifies two main problems that must be solved in order to create these deployments: 1) the different stakeholders must be able to clearly express and encode the necessary policies and security requirements, and 2) the underlying embedded nodes must be able to enforce these policies.

The first problem is the control and management of the security of these embedded nodes. In a static and isolated environment, the owner and builder of the network can often manage all complexity by himself, or can keep the complexity manageable for a small team. The users can manually code their policies, or configure the systems on a fairly low level. The users of these systems need to have significant knowledge of each part of the system, including the security sub system. However in order to enable non-expert users to setup and maintain these systems, higher levels of abstraction are necessary. While this thesis will not specifically target the non-expert, it does attempt to provide the different stakeholders with the necessary abstractions and tools in order for them to easily express their security policies.

The second problem this thesis tackles is the security architecture for embedded networks. While writing down policies is a necessary first step; if the policies cannot be enforced, the system cannot be secure. Related work has proposed significant work on how to secure embedded networks. However, most of these systems look at embedded networks as static isolated systems, owned

and managed by a single party. As touched upon in our application analysis, these assumptions are evolving. More and more Networked Embedded Systems deployments are required to integrate many different systems, cope with both hardware and software dynamism, and enable sharing at a device level. These changes require additions to the security services available on these constrained devices. Several key functions, such as network setup, code deployment, node management, and secure end-to-end application communication need to be revisited to ensure they can be optimally and securely used in these shared environments.

So the two main problems this thesis aims to solve are:

- How can users effectively and easily configure this security system to ensure the executed security matches the user's requirements?
- Which are the minimal system requirements for a node middleware that enables secure, shared usage of these resource constrained nodes based on the previously expressed policies?

To handle these problems, this thesis will 1) define a set of policy abstractions which allow users to easily define their security requirements in policies, and 2) design and implement an embedded node middleware that is able to enforce the policies. These solutions will follow the separation of concerns design principle on both the policy abstractions and the node architecture.

Many different stakeholders have different security requirements, which need to be enforced. To enable this, the different stakeholders must be able to express their requirements into policies. These policies must then drive the decision making of the different security systems, allowing or denying users or applications from performing certain actions. This policy driven security allows the end-users to easily declare and update their requirements, with limited knowledge of the underlying system which enforces their policies.

When developing the embedded node architecture, the security concern must be separated from the functional application concern. This means that the security infrastructure must operate transparently for applications. This allows application developers to purely focus on the creation of their applications, without having to be concerned about the underlying security infrastructure. The security infrastructure must then operate underneath. This separation of concerns simplifies the application creation process, and ensures application developers do not have to develop their own security solutions. Recent revelations have shown that even popular security software can contain errors, even if it was written by security professionals. Requiring application developers to create their own security framework has shown to often create even more security holes.

To create a reusable and extensible environment, we aim to create this system as a security framework. Gamma et al. [38] define a framework as ”a set of cooperating classes that make up a reusable design for a specific class of software.” The frameworks also defines how different systems cooperate, and the thread of control. This allows application developers to create applications faster and easier. As such, our goal is to define a set of modules and systems that together make up a reusable system and design for securing multi-party networked embedded systems. We will implement and prototype this system, and propose clear points of extension, allowing users to customize the framework to adapt to specific use cases.

1.6 Contributions

To tackle the problems listed in the previous section, this thesis presents the SecLooCI infrastructure: a policy driven security infrastructure that enables sharing of constrained networked embedded systems. SecLooCI provides security feature to easily enforce security on resource constrained nodes. The two scientific contributions of SecLooCI are 1) a set of abstractions that allow the different stakeholders to express their policies, and 2) a per-node security middleware that can enforce the previous policies.

The security management abstractions allows application owners, platform owners and network owners to specify their requirements in high level policies. By creating clear abstractions, it is possible to assist and guide the users while entering policies and to verify that the entered policies are consistent and feasible. During the lifecycle of the networked embedded system, these abstractions are enforced by the per-node security middleware.

The per-node security middleware secures the interactions necessary to set up, run and maintain shared NES deployments based on the policies entered by the different parties. More specifically this middleware enables the secure joining of networks, deployment and management of applications, securing application communication and monitoring application and user node usage. The middleware is controlled by lightweight policies, and is transparent for the applications running on top of the middleware. This allows the easy addition of security to any supported application, and allows the application developer to focus on the application logic, while the security developer can focus on offering reusable security systems. This increases security, and promotes reuse.

The SecLooCI middleware provides security on top of the LooCI component middleware. The LooCI middleware offers a loosely coupled semantically typed communication layer and easy application deployment for embedded systems.

The loosely coupled communication layer allows for the addition of the security middleware and lightweight communication system necessary for configuration and management. The LooCI application deployment framework offers the basis for the secure application deployment framework.

All parts are individually evaluated using state-of-the-art constrained hardware, as is the system as a whole. Next these systems have been validated through a use case derived from experiences gained in industrial projects. A final integrated evaluation is performed using a real-world deployment of the security middleware in a smart lab environment.

1.7 Overview of Thesis

The remainder of this dissertation is structured as follows:

- **Chapter 2** details the context of this thesis. It first provides two use cases: the smart office use case and the smart logistics use case. Next, based on these two use cases, it presents the three main roles a stakeholder can take in the shared embedded ecosystem. Then it presents an extended lifecycle for shared networked embedded system applications. It continuous with detailing the two main problems this thesis will tackle: the need for adequate security abstractions which the stakeholders need to express their requirements, and the need of a security middleware to enforce those policies during the five main data flows that involve multiple stakeholders. Finally it lists a set of security and non-functional requirements for this system.
- **Chapter 3** provides an overview of the related work of this thesis. It first looks at the current related work in managing the security and the applications of an embedded network. Next it looks at the current state-of-the-art solutions for each of the five data flows which the security middleware aims to secure.
- **Chapter 4** provides an overview of the SecLooCI network architecture. It identifies the five main architectural elements in this environment, and details for each of them their roles and responsibilities: 1) the end-user tool, 2) the application owner server, 3) the platform owner server, 4) the network owner gateway, and 5) the shared embedded node. Next it provides an overview of the SecLooCI embedded node security middleware architecture, starting from a basic embedded node system. Finally it provides an example smart logistics scenario that shows how the SecLooCI security framework operates during the previously presented lifecycle.

- **Chapter 5** presents the first contribution of this thesis: a set of management abstractions that allow the different roles to express their security requirements and model their applications and environments. It then also provides a small evaluation of these management abstractions.
- **Chapter 6** presents the second contribution of this thesis: an embedded node security middleware: the SecLooCI middleware which consists of five subsystems: each of them securing one of the five main data flows crucial to creating shared networked embedded systems: 1) network setup, 2) application deployment, 3) application management, 4) application communication, and 5) service usage. Finally this section presents an overview of the integrated prototype.
- **Chapter 7** provides an evaluation of the SecLooCI framework. It first details the overhead of the different subsystems based on the previously presented smart logistics scenario. Next it presents the real world smart office deployment: it lists the different smart office applications, some metrics provided by the deployment, and some general observations. This evaluation shows that the SecLooCI infrastructure is able to continuously operate on resource constrained embedded nodes and is able to provide the necessary security for enabling secure shared embedded networks.
- **Chapter 8** provides a discussion of the SecLooCI infrastructure. It first goes over the non-functional requirements as presented in Chapter 2, and discusses how the SecLooCI framework meets those requirements. Second it provides a security analysis of the SecLooCI node middleware. Finally, it discusses the SecLooCI framework with regards to the trust requirements, the energy consumption, and the trade-offs and limitations.
- **Chapter 9** finally provides the conclusion of this dissertation. It first summarizes this work and lists the contributions again. Second, it provides some lessons learned during the execution of this work, Next, it provides some interesting avenues of future work, to finally end with an outlook on the future of shared networked embedded systems.

Chapter 2

Context

This section first presents the context and requirements of this thesis. First it lists the two main use cases in the area of networked embedded systems (NES) which have driven the development of this thesis: the smart logistics case, and the smart office case. Both cases have been developed in research project in collaboration with industrial and academic partners. Next this section identifies the three generic roles that can be identified in both case: the platform owner, the node owner, and the application owner. Then this section presents a shared NES lifecycle. From this lifecycle and role identification, the problem statement is explained in detail. Lastly, this section lists the attacker model, and the non-functional requirements of the framework.

2.1 Use Cases

This section presents the two use cases that drive the research presented in the thesis. These cases are the result of multiple projects in collaboration with Flanders and European collaborators from both industry and academia. The first use case is the smart office use case, which has been developed in the frame of the ITEA DiY-SE project, and further worked out using an in-house smart office development project. The second use case is a smart logistics use case, which has been developed in the industrial research projects ICON STADiUM and ICON COMACOD.

2.1.1 Smart Office

Domotics environments offer an interesting use case, especially when applied to office environments. It showcases a multi-party environment, dynamism, and a clear need for security. It is also of particular local relevance as companies in and near Flanders are exploring this interesting application domain, as observed during the ITEA DiY-SE project.

Smart office scenarios typically start with the deployment of sensing nodes in the building, and slowly evolve towards the adaptation of building services and environment depending on the requirements of the people present in the building. For example when a worker enters the building, automatically his office gets lighted and the heating turned on to his preferred temperature, or a mean temperature between all occupants of the space. When no one is using a space, heating and lighting are shut off, potentially providing significant energy gains. Access to certain equipment might need to be monitored or controlled, such as for example opening door and cupboard locks, or using electronics such as coffee machines.

There clearly are multiple parties that need to collaborate with each other. Each office worker has his or her own preferences, requirements and devices. These systems should be able to seamlessly integrate, allowing preferences to be applied across multiple systems. For example a worker should be able to have a single set of preference applying to both his work and office conditions, and interact with different systems using his own smart devices. These systems should not be site specific, since this would greatly increase cost and effort. Hence there is a clear need for collaboration.

Second there is also clear dynamism. Each worker carries his or her own mobile smart phone, that should be able to interact with the different smart systems. These devices come and go into the system frequently. Additionally, these systems will not be built in one shot. Rather there will be a continuous addition and integration of devices when new needs or capabilities arise. These new devices need to be able to seamlessly integrate with existing infrastructure. Finally existing hardware might break down, or contain bugs, requiring hardware or software updates. Applications should be able to handle these upgrades, and functionality should be easily transferable between different devices.

Lastly again there is a clear need for security. Access to office spaces should be controlled, but also monitoring data of the buildings and control of building's actuation systems and electronics must be secure. Intruders should not be able to manipulate lighting and heating, potentially causing increased cost, or decreased working comfort. Electronics usage should also be monitored, to ensure consumption can be attributed to users, or just to create visibility on

resource usage.

2.1.2 Smart Logistics

The logistics industry offers a second excellent use case for shared networked embedded systems. While it is similar to the smart office use case in that it showcases a multi-party environment and has a clear need of security, the smart logistics case expands on these concepts. In Smart Logistics, there is a higher level of network heterogeneity, and a much higher level of dynamism due to the mobile trucks and containers. Again, this research is driven by the local Flanders context, as shown by collaboration with multiple industry and academic partners in the ICON STADiUM and ICON COMACOD research projects.

In logistics scenarios, cargo owners want some cargo to be transported. These cargo owners use the services of logistics providers, which provide end-to-end transport. To do the actual transportation, these logistics providers use the services of transport providers, which provide point to point transport, such as warehouse to harbour transport, or sea transport.

In recent year, an increased demand in supply chain visibility has driven an increase in container monitoring. Logistic providers are instrumenting their containers with embedded nodes to provide environment monitoring inside the container, such as temperature and humidity. Transport providers also install embedded nodes in their trucks which monitor other variables, such as truck location, driver speed, and driver efficiency.

All parties in this environment are clearly interested in integration and collaboration in order to access the sensor data of other parties. Cargo owners want both data about the internal environment and truck positioning. Logistics providers need to ensure that the transport provider transports goods with adequate quality of service. Transport providers on the other hand want the internal monitoring data, so they can immediately react in case of issues. The transport providers prefer to receive data immediately from the embedded nodes to ensure low latency data. In return for direct access to the embedded nodes, the transport provider could provide network access to the embedded nodes in the container. Lastly government and customs officials also require access to temperature, lock, and location data to ensure security and quality. For example both the US C-TPAT [22] treaty, and the European Authorised Economic Operator certificate [88] require container lifecycle visibility to ensure security and allow more smooth customs processing. This again shows a clear need for collaboration and sharing.

These environments are also highly heterogeneous. Different logistics providers or transport providers likely will not use the same hardware, so different containers will contain different types of embedded nodes. This heterogeneity also shows in node capabilities. The embedded nodes in the container are typically fairly constrained to save space, weight and energy consumption. The embedded nodes installed in trucks are typically standard consumer computers, which are far less constrained, but require more energy and space.

These environments are also dynamic. Trucks connect to different trailers daily. Containers travel all across the world, needing to communicate with whatever local entity is present. Each network thus likely consists of some static parts, nodes that are always present such as those at gates, or lights, or motion sensors, and nodes that are mobile, and can appear and disappear at arbitrary times. Depending on the parties present in the network, the applications running on top of these networks can differ too. For example, some networks at harbours might run certain localisation applications, requiring custom software to be installed on some nodes. Some containers might want to use services of nodes around them to for example use Internet services, or use sensing services to verify their own measurements, or handle failures.

Security is again a major concern. Authorised parties, such as container owners or customs, should be able to communicate with the container nodes, inspecting the content, change cooling requirements, or open a door lock. Of course non-authorised parties should be prevented from accessing the node data and services, since disclosure and tampering might lead to significant economic loss. Imagine a burglar who can just inspect the container's contents, and chose the most profitable containers, or a vandal who turns the cooling off in a medicine container, spoiling the contents.

2.2 Stakeholders

This section presents the different stakeholders present in the different environment. These stakeholders are identified based on the industrial use cases presented in the previous section. This thesis identifies three roles, as shown on Figure 2.1: 1) the platform owner who owns the shared devices, 2) the network owner who manages the network to which the devices connect, and 3) the application owner who wants to use the devices and network to perform some kind of function. Note that a party can combine two or even all three of these roles. This thesis though attempts to clearly separate rights and responsibilities to get greater clarity on the studied environment.

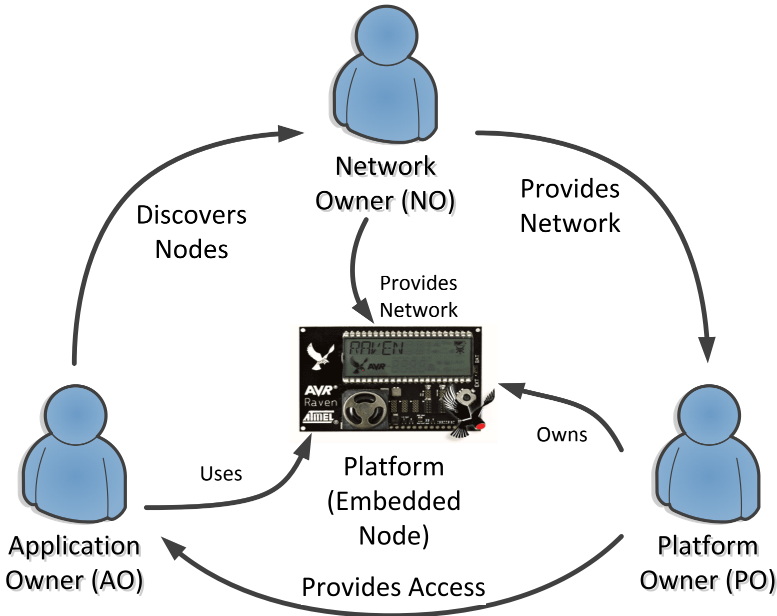


Figure 2.1: Overview of the three roles present in the NES ecosystem.

2.2.1 Platform Owner (PO)

The platform owner is the owner of the embedded nodes. He acquires these nodes because he wants to monitor or augment certain systems. However, once he has instrumented these systems, he wants to share the nodes with others, because this will increase the return on investment of the nodes, or allows the platform owner to offer additional services to the users. Take for example the logistics use case, where the logistics provider provides cargo monitoring to the cargo owner, and potentially can use the localisation data provided by the transport providers.

To share, the platform owner must be ensured of the continued safe and secure operation of his sensor nodes. The platform owner must be able to limit the access that users of the nodes have. For example in the logistics case, transport providers should be able to see the status of the lock of the container, but should not be able to open it. Additionally, the platform owner must be able to monitor the usage of his nodes, to prevent misuse, or to charge the usage. Take for example in the smart office case, where users of a coffee machine get their usage visualised, and potentially have to pay for using the coffee machine.

2.2.2 Network Owner (NO)

The network owner is the owner and manager of a network infrastructure. This infrastructure enables authenticated nodes to connect to the Internet, and communicate with other nodes present in the network, similar to current office WiFi environments. Take, for example, the logistics use case, where containers arrive in a harbour. The nodes in the containers prefer to transfer their data through cheap harbour WiFi, instead of through expensive GPRS connections. The network owner can also provide an overview of the nodes which are present in his network, and the services they offer.

The network owner must be assured that only trusted nodes can join the network. For constrained nodes, this can pose a problem, since quite often these nodes travel autonomously, and lack peripherals that are needed to enter a password, as is often done with WiFi.

2.2.3 Application Owner (AO)

The application owner is the user of the embedded nodes. They want to use embedded services provided by nodes in their environment of interest. Often the application owner does not own the nodes which he wants to use. Consider for example the smart office use case, where the office workers want to be able to set the preferred temperature and lighting conditions in their environment, while the infrastructure is owned by the employer.

By using shared services, the application owner does not have to invest in costly infrastructure himself, or can get information from environments that are often not available to him. For example in the logistics use case, the transport provider wants data from the inside of the container to ensure the quality of transport. However often he has no access to the inside of the container. The Transport Provider can gain this data though by using the shared nodes of the logistics provider.

2.3 Sensor Network Application Lifecycle

Based on the observations made in the smart office and smart logistics use case, this thesis proposes a slightly extended lifecycle based on the classic create, deploy, run, remove lifecycle [78], as shown on Figure 2.2. The create phase is divided in the policy declaration phase, the application creation phase, the network setup and application instantiation phase. The removal phase is divided

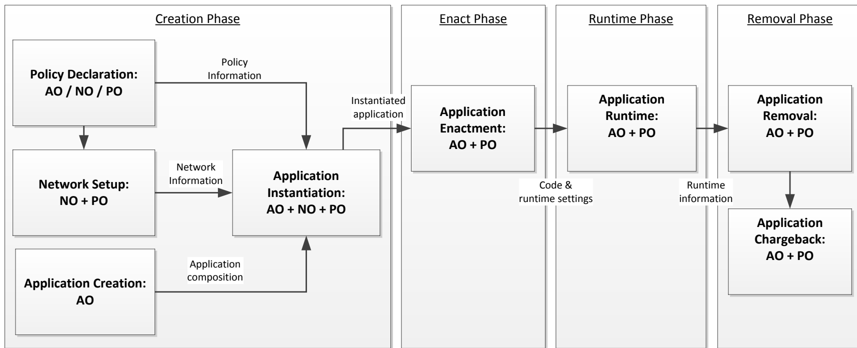


Figure 2.2: Overview of the lifecycle of a typical Networked Embedded System application.

in the application removal phase, and the application charge-back phase. This section explains each phase, and describes the responsibilities and requirements each role has for each phase.

The policy declaration phase entails the formal declaration of policies by all three roles: the platform owner, the network owner, and the application owner. In this phase, all roles need to be able to specify their policies in a way that: 1) is easy for the users, 2) allows for all requirements to be expressed, 3) can be aggregated and reasoned on in the back-end, and 4) can be enforced on shared embedded devices.

When applied to the use cases, the different visitors of buildings must specify that they trust the nodes provided by the building. In the logistics use case, the logistics providers must specify that they trust the different transport providers, and cargo owners to access the data from their containers.

The application creation phase produces an application which can be deployed onto a shared embedded network. In this simplified model, it is the duty of the application owner to acquire this application. The application owner can do this by either creating the application himself, or buying the application from a third party. In order to reduce effort and reduce cost, the application owner requires that applications can be reused and are not network or node specific.

In the domotics case, most consumers will likely download their embedded applications from platforms similar to the current app stores. These applications would then be instantiated on the user's local environment, and process data from the ubiquitous data sources. Building owners and logistics providers and transporters can potentially buy applications that will likely be customised for

the particular environment, as having generic systems will not be optimal, or sometimes even feasible to deploy.

In **the network setup phase**, the devices of the platform owner join the network of a network owner. The network owner then provides the nodes with Internet connectivity. In this phase, the platform owner requires that his nodes can only join networks that belong to trusted network owners. The network owner also requires that only nodes of trusted platform owners can join the network. The platform owner and network owner may also need to negotiate on the cost associated with using the network infrastructure, and the Quality of Service offered. This phase assumes that all devices that join the network can communicate with each other. While currently, significant research is being done on alternate MAC, physical and routing protocols, it is clear that in order to enable these shared and interconnected use cases, these devices need to be able to communicate with each other. It is thus necessary for clear standards to emerge, which will be adopted by either all systems, or certain isolated communities.

In the use cases, clearly the building operators, transport providers and harbour owners must set up their local network by installing the necessary gateways and setting up Internet connections. Without these systems, it would not be possible to create a shared embedded network. These parties can then offer their network to visitors of their site, just as they are likely currently offering a WiFi network to visitors.

In **the application instantiation phase**, the application owner instantiates the application that he wants to deploy on the shared embedded network. The application owner uses the application he previously acquired, and instantiates it by either adding certain requirements for the application, or assigning behaviour manually to certain nodes or locations.

Take for example a localisation application in a harbour. The harbour owner wants to monitor and locate all containers currently present in his harbour. To do this he buys a generic localisation application, and instantiates it to automatically deploy on all nodes that enter the harbour network. He also sets certain parameters, such as fixed localisation beacons, and sampling rate. Additionally he must be able to set certain non-functional requirements, such as the level of security required for all communication.

In the domotics use case, the end-user might want to have an application that automatically regulates the HVAC and lights whenever he is near to his preferred setting. To do so, he will have to instantiate the application, and modify it with the for example which temperature, air quality and lighting the user prefers.

So in this phase the application owner requires that he can easily instantiate

applications onto the embedded networks, and add some additional non functional concerns. Additionally the application owner needs to be able to retrieve network and node data, such as which nodes are capable of offering which services, and which services he is allowed to use.

During **the application enactment phase**, the application owner enacts the application he instantiated in the previous phase. This thesis operates under the assumption that new application functionality must be deployed onto generic nodes, allowing for example a temperature controller to aggregate data from multiple sources and send commands to a thermostat. In order to do so, the application owner needs to deploy certain code or configurations into the embedded network. Since many environments are quite heterogeneous in terms of hardware and available services, it is most often the case that only a limited subset of nodes will receive a certain configuration [89, 51].

Take for example the smart office example. If the building security officer is interested in whether or not the doors and windows are locked, he does not need information from all the light, temperature and electronics nodes in the building. This assumptions is a bit different from much of the related work on embedded network deployments, which often assumes full homogeneous networks that all need to operate using the same image.

So during this phase, behaviour needs to be deployed onto sensor nodes, either by deploying configuration policies or binary code. This deployment can be delegated to the platform owner. However, this thesis proposes to allow application owners to deploy their own behaviour. This enables application owners to configure nodes even when potentially no Internet is available, reduces the load on the platform owners, and is the most generic case. Naturally both the application owner and platform owner want to be ensured that the deployment happens securely. The application owner wants that his application is deployed unchanged, while the platform owner needs the continued safe operation and integrity of his nodes.

The use cases clearly show that at certain times users will want to deploy certain configurations or additional code onto the network to establish some desired functionality. In the domotics use case, the visitor wants to deploy some behaviour on the lighting system, that receives some information from lighting sensors, and then automatically adapts the current lighting level. In the logistics use case, the harbour customs might deploy a piece of code that calculates the container's current location based on several beacons.

Once all functionality is deployed and running, **the application runtime phase** start. The application communicates over the network, and potentially interacts with Internet based services, or other local services either of the

application owner, or of other parties. In the use cases, the runtime phase is the phase where the application behaviour is running. In the domotics case, the user would get his environment preferences automatically enacted wherever he goes. In the logistics case, the embedded applications gathers the location of all tracked containers in the harbour. During this phase the application owner requires that he can modify his application, that data is sent securely across the network based on platform owner and application owner data security policies and the platform owner requires that he can monitor the actions the applications and application owner do, and that he can limit those actions.

Firstly, the application owner must be able to modify his application during runtime. This can either be fine grained adjustments, such as changing the heating setting in a smart office environment, or coarse grained adjustments, such as when a new node enters the harbour network, and needs new code deployed.

Secondly, the data sent across the network must be secured depending on data security policies by the application owner and platform owner. Since embedded network applications potentially produce delicate data, it is vital that the different parties can enforce data security policies. This means that the data must either be sent encrypted, or authenticated over the network. However, adding security measures incurs a cost on the resources of the nodes. As such, the platform owners or application owners might decide to not secure certain kinds of non-sensitive data to reduce the burden on the node.

Finally, the platform owner must be able to monitor and control the access of both users and applications to his node system for at least two reasons. The first reason is that the platform owner must be able to ensure that applications or users cannot, accidentally or wilfully, cause harm to or excessive strain on the system. By monitoring their behaviour, excessive use can be identified or prevented. Second, in future we might start to look at these networked embedded systems as a shared sensing infrastructure, much like the current cloud infrastructure provides a shared computing infrastructure. Just as currently users of the cloud are billed based on their usage of computing resources, users might be billed in this embedded *mist* environment based on their usage of sensing or computing resources. Thus the operations of the user must be logged to allow a detailed calculation of incurred cost.

When the application owner wants to end the application, **the application removal phase** starts. During this phase, all functionality deployed by the application owner is removed from the used nodes, and the node usage is charged back to the application owner. In this phase, both the application owner and platform owner require that this clean-up happens securely, completely and does not interfere with others. The application owner should not be able to

remove functionality that is used by others. On the other side, all functionality that is no longer used should be removed completely in order to free all reserved resources. Once all application parts have been deleted, final monitoring data is collected, and the actual usage can be charged to the application owner.

In the domotics use case, when a user leaves a building, the user specific application which modifies the building's systems based on user preferences can be removed, to free up resources. It is unlikely that building operators will charge users for using the HVAC and lighting system, but this technology would at least make it possible. In the logistics scenario, the harbour customs can provide a localisation service for the logistics provider, letting him know where his cargo is at all times. When the node then leaves the harbour, all functionality and configuration should again be removed from the node, to free up valuable node resources.

2.4 Problem Statement

This section specifies which specific problems this thesis tackles, and the approach taken. The goal of this thesis is to raise the abstraction level of the configuration and management of applications running on shared systems and identify and implement the necessary software solutions to enable sharing on resource constrained embedded nodes, as shown on Figure 2.3. Hence, this thesis is divided into two parts: 1) A management infrastructure that provides abstractions to model applications and owner requirements, and 2) a security middleware that provides a software layer to secure the multi-party interactions and enforce the previously created policies. The remainder of this section define these two problems more clearly.

2.4.1 Security Management Abstractions

The security management abstractions aims to allow the different roles to easily express their security requirements in a policy oriented fashion, to distribute these policies, and set up networks and deploy applications so they are compatible with the expressed policies. This section identifies for each role what they need to express.

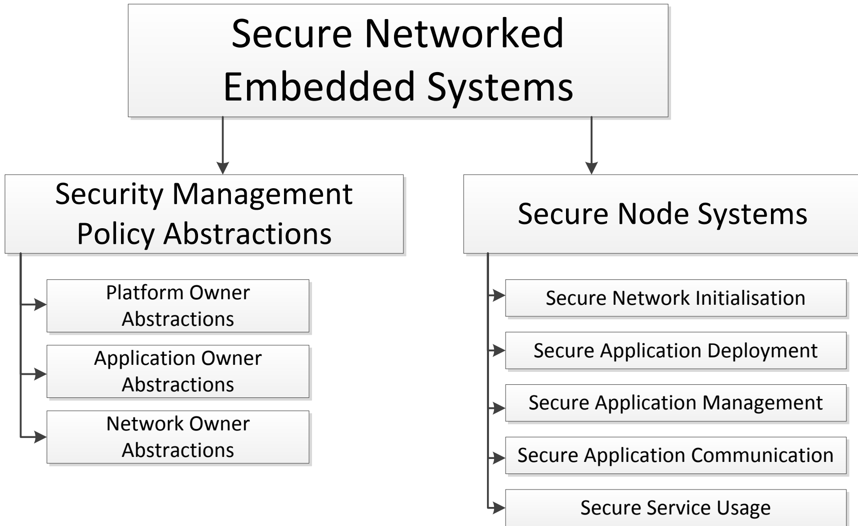


Figure 2.3: Overview of the problems tackled in this thesis. To create secure networked embedded systems, two components are needed: users must have abstractions to declare their policies, and these policies must be enforced by secure node systems during the lifecycle of the application. This thesis will look at the required abstractions for each role, and at the node security systems to enforce these policies for those data flows that are impacted by sharing.

Network Owner

The network owner must specify which platform owners are allowed to use his network, and what he requires in compensation for using his network. The network owner can either specify each party that he allows to use his network, however this solution would be very un-scalable. The trust relationships would likely be created through third party trust providers, who ensure that all approved parties are trustworthy. This thesis identifies two main costs: 1) charging money for using the network, and 2) requiring certain sensor data and/or processing services to be made available to the network owner.

Platform Owner

The platform owner must express the policies regarding the networks which his nodes are allowed to connect to, the services that users are allowed to use on

his nodes, and the security requirements for data produced by and commands sent to his nodes.

The platform owner first must express the policies with regards to the network owners. As a platform owner, it would be very un-scalable to manually list all trusted network owners, so likely here too a third party trust provider will be used. The platform owner must also specify the money he is willing to spend in order to use the network, and which node services he is willing to provide to the network owner. If the network owner is trusted, and the price of the network owner is below the willingness to pay, the platform owner can allow his nodes to use the network.

Second, the platform owner must also express his policies with regards to the application owners. The application owners want to use node services to create an application which uses local sensing or actuation services. Again first the platform owner must be able to trust the application owners using his node. Again the most scalable solution will be the usage of third party trust providers to set up the trust relationships. However, certain platform owners (such as private persons) likely will only want to open up their platforms to a very limited number of people such as friends and family. In such cases, the platform owner must have the ability to manually enter the trusted parties. The platform owner must also specify which nodes and which services of those nodes are available for the application owners, with which limits, and at which cost. These requirements must be taken into account at the time of the application owners deployment planning, and must be monitored and enforced during runtime.

Next, the platform owner can also potentially specify certain data level security concerns. For example certain platform owners will offer location services to application owners. However, the platform owner might want to ensure that all location data produced by his nodes is kept confidential and only sent over the network encrypted. The platform owner must be able to specify such requirements, they must be taken into account during the application instantiation phase, and enforced during the runtime.

Last, the platform owner must model the current node environment, and keep a log of the current node usage by the different application owners. In order for application owners to decide whether or not to use a node, they must be able to see which resources this node offers. Platform owners thus must model and provide this information. Second, once the application owner decides to use or deploy some functionality on the node, the application owners usage must be stored to ensure no excessive resources can be used. The actual node usage of the application owner should be logged in order to allow charge-back.

Application Owner

The application owner must be able to express which platform owners and network owners he trusts, which resources his application uses, which cost he is willing to pay, and what security requirements he has with regards to the data.

First the application owner must specify whose nodes he is willing to use. The application owner only wants to use nodes of parties he trusts sufficiently so he knows his application is not threatened by using those nodes. Again likely application owners will use third party trust providers, or parties he knows to be trustworthy. Many use cases involve the application owner to deploy an application on nodes locally, so likely in a single network. Thus there must also be trust between the application owner and network owner, so the application owner can request trusted nodes from the network owner. In such cases the network owner can act as trust provider between the application owner and the platform owner.

Second the application owner must specify which resources his application uses. This is necessary in order to allow automated identification of suitable platform owners and nodes, plan the deployment, and create a cost estimate of the resource usage. The application owner can then in advance get an estimate of the cost of running his applications, and be certain the necessary resources are available. The platform owners of the nodes on which the applications run can reserve the required resources in order to prevent resource contentions.

Lastly, the application owners must specify their application data security requirements. In a distributed embedded application, data will be transmitted continuously over the air. There are potentially many different types of data being communicated, such as for example temperature readings, or location beacons. Since security incurs a cost, the application owner could prefer to not secure certain types of data to reduce the load on the nodes and the incurred cost. The application owner policies must then be merged with the relevant platform owner policies in order to create policies adhering to both preferences.

2.4.2 Secure Node System

The second problem this thesis tackles is the creation of a secure node system. This system must be able to offer the necessary security as required by the different parties during the entire lifecycle of the shared NES application. In order to create such a system, this thesis looked at the different data flows which involve the node system which need to be secured, an approach recommended by many parties a.o. the SANS institute [9]. From these data flows this thesis

isolated five data flows that involved multiple roles, and as such would likely have to be updated when creating a secure node system, capable of supporting shared NES applications. The remainder of this section identifies the security requirements for the secure node system based on these different data flows, and based on the requirements of the different roles as listed previously.

Network Initialisation

The first data flow is the network initialisation. This data flow starts when a new node enters the network of an network owner. The node must set up communication with the gateway, in order to join the network. Both the node and the network owner must be ensured of each others trustworthiness. This trust must be established, after which the necessary key material can be deployed on the node.

Once trust is established and the network owner key material is deployed, the node is part of the network. Likely additional networking options need to be set, such as channel details, routing options and other network configuration parameters. However, this thesis considers routing, intrusion detection and network management out of scope for two reasons: 1) the data flows of these interactions are not significantly altered by introducing it in a multi-party environment, and 2) there is abundant related work available that provides solutions for these issues.

Application Deployment

During the lifetime of a node, it will need to receive code updates. These code updates can either be the platform owner that wants to update his core functionality, or an application owner that wants to deploy or change functionality on the node. The platform owner requires that only trusted and authorised parties can deploy code updates. It should be possible for the platform owner to require that code can only be deployed after being reviewed. Once approved, the code must be deployed unchanged. Additionally the platform owner must be able to specify resource restrictions on the deployed code.

The application owner also requires that his code is deployed without alterations, to ensure that a networked attacker cannot alter his code to gain control over the node. Some application owners might also require their code to remain confidential, and thus require the code deployment to be encrypted.

Application Management

Once the nodes are placed in the environment, entered in the network, and got the necessary new applications installed, the nodes needs to be managed. The application owner needs to be able to change application settings of his personal applications. The platform owner on the other hand needs to be able to control the node system and manage the resource restrictions on the installed applications. Management typically happens by sending management command messages to the node system, which processes these messages and returns a reply. Both the application owner and the platform owner require that these commands are authenticated and authorised, in order to ensure node and application integrity. Additionally for management functionality, the platform owner wants a secure-by-default approach to ensure all management communication always adheres to a high standard of security.

Application Communication

In distributed embedded applications, the different components need to be able to communication with each other in order to aggregate and process data. The application owner and platform owner will have security requirements for the transmission of this data depending on the sensitivity of the data. The secure node system must be able to enforce those data security policies. The system must support multiple different levels of security, since depending on the applications the users might want no security, only authentication, or full confidentiality enforced. Additionally the system must be able to support per data type policies and per interaction policies: depending on the sensitivity of the data, the level of security can be reduced to reduce resource overhead. Additionally, certain nodes might be involved in multiple applications, each of which might have different security requirements.

For example a harbour owner has a localisation application for his harbor. He requires that all localisation data is exchanged using at least an authenticated protocol. The container owner of a certain group of nodes however requires that location data is sent confidential on an end-to-end basis. The node system must support such different policies to support the full range of NES applications.

Application Service Usage

Applications running on embedded nodes or users interacting with these nodes will use node services, such as file system, processing and sensors. The platform owner requires that this service usage is restricted to approved users and

applications only, and that the usage is monitored. The platform owner must be able to limit which application owners and which applications have access to which services and with which limitations. These restrictions must be deployed onto the node, ensuring that an application component can only use a limited amount of node services. Lastly, the actual usage of the applications and users must be logged and aggregated at the platform owner to ensure that the platform owner can intervene in case of excessive node usage, and the platform owner is able to bill the application owners based on actual node usage.

2.5 Requirements

This section specifies the requirements of the security framework. First it looks at the different kinds of attackers that are potentially present in an embedded network. Second this section looks at the non function requirements of the security framework.

2.5.1 Attacker Model

The security framework must be able to handle multiple different types of attacks. This section identifies four main types of attackers: 1) outside attackers, 2) network attackers, 3) physical attackers, and 4) insider attackers. This section specifies for each of these attackers their goals, means and constraints. Additionally for each attacker this section details what the node middleware should be able to prevent or mitigate.

Outside Attacker

The outside attacker is a classic attacker according to the Dolev-Yao model [30]. He can intercept, manipulate, duplicate and create messages, but he cannot break any cryptographic primitives or protocols. The goal of the outside attacker can be either to gain information gathered by the network, or to enact some kind of control over the network. The information on the network can either be data produced by the embedded nodes, or data about the nodes themselves, such as identification information or capabilities. Gain control over the network can also take different forms. He can change application level parameters such a temperature setting. Alternatively he can influence the network by attempting to corrupt nodes or influence the network by using for example a Denial of Service attack. The node security middleware must prevent outside attackers from being able to manipulate the sensor node systems, to prevent him from

gaining any control over or information from the sensor nodes. This thesis however does not consider network level attacks, and does not aim to mitigate against them. Additionally a node cannot prevent an outside attacker to deny communication, however he should not be able to enact any other influence on the nodes.

Network Attacker

The network attacker is an attacker that controls embedded nodes which are a valid part of the wireless network. As such, he has all the capabilities of an outside attacker, and because the network attacker has permission and access to the network has some additional capabilities. He can use this access to perform additional attacks, such as interfere with routing. For example in a wormhole attack, the network attacker inserts himself in between the communication of different embedded nodes to be able to potentially read any communication that is not end-to-end encrypted, or to perform traffic analysis. His goals are similar to the outside attacker in that he wants to gain information on the network, or some kind of control over the network by manipulating messages.

Physical Attacker

The physical attacker is an attacker who can gain physical access to a system, and probe the memory and code. This allows a physical attacker to retrieve all key material stored in the attacked node, and perform arbitrary modifications, since current generation sensor nodes are not capable of resisting these kinds of physical attacks. Recently state-of-the-art has proposed solutions to prevent the probing of memory through tamper proof hardware [59], prevent the leaking of sensitive key data through trusted platform modules or physically unclonable functions (PUFs) [75], and detecting changes in the software through attestation [14]. The goal of the physical attacker is to penetrate the network using the credentials gained from one or several compromised nodes. As the network attacker, he wants to get information from the applications running in network, or exert some control over those applications. While it is impossible for node software to prevent these kinds of attacks, the node software should be able to mitigate the consequences of a node compromise on the other embedded nodes. The middleware must be able to recover any application functionality once a node disclosure has been detected, and must prevent physical attackers to gain a significant amount of control over other nodes using the disclosed key material.

Inside Attacker

The inside attacker is an attacker who has valid access and credentials to the embedded node, but has certain access limitations and restrictions. He wants to gain more influence on the embedded node and perform operations that should not be available to him (elevation of privilege), or use resources without the system being able to monitor the usage (monitor bypass). The node middleware must prevent the elevation of privilege attack, and must prevent inside attackers from being able to bypass the monitoring system. Additionally the monitoring system should be able to detect any unauthorised access of authenticated users, in order to potentially detect elevation of privilege attacks.

2.5.2 Non Functional Requirements

This section specifies the non functional requirements of a secure embedded network framework. These requirements flow from the observations made in the use cases: that the software and systems are continuously evolving and very heterogeneous, that users want to express their policies transparently of the underlying infrastructure, and that those policies are enforced transparently from the applications, and that security system is flexible with regards to communication pattern and security level.

- Evolvability of infrastructure: new sensor nodes continuously join the network or running embedded network application, existing nodes move away, or break down. The system must be able to handle the continuous churn of nodes.
- Evolvability of software: new applications can be installed on the embedded network, existing applications can be updated and changed due to changed requirements, and obsolete applications can be deleted.
- Heterogeneity of infrastructure: the framework must support nodes with a with a wide variety of capabilities and available hardware.
- Transparency of heterogeneity: security policies and application requirements should be expressed without having to be concerned about the exact platform on which the actual code will be running or which other software is potentially running on the platforms.
- Transparency of security: applications developers should not be concerned about the security requirements. Security requirements must be expressed and enforced transparently for the applications.
- Flexibility of communication: embedded applications potentially have many communication patterns, such as one-to-one, many-to-one, one-to-

many, or many-to-many. all communication patterns need to be supported by the communication framework.

- Flexibility of security: enforcing security incurs a cost. The application deployer should be able to chose the level of security, and the associated cost.

2.6 Summary

This chapter provided the context of this thesis. It first proposed two use cases, which were derived from collaborations with local industry and academia: the smart office use case, and the smart logistics use case. These use cases clearly show that networked embedded systems are multi-party environments, where embedded nodes from different parties work together to create a smarter environment. In such environments, security is naturally a key concern. Additionally these use cases show that there will be a high degree of software and hardware heterogeneity, and mobility, which is often overlooked in current related work.

Second this chapter identified three unique roles in this NES ecosystem based on the presented use cases. The platform owner owns devices which can be constrained and mobile. In order to gain maximum functionality from these nodes, he want to share these nodes with other parties. The application owner wants to use these shared devices to easily set up certain functionality. For example in case of the smart office, users want to interact with the HVAC or lighting system. Finally, since many of these devices are mobile, they require a network in order to communicate with each other and with the Internet. The network owner provides this network to the platform owners and application owners in order for the devices to be connected and usable.

Next this chapter proposed an extended lifecycle, in order to identify the responsibilities and requirements of these roles during the entire application cycle. The seven phases are: 1) the policy declaration phase, where the different parties declare their security requirements, 2) the application creation phase, where the application owner either creates or acquires the application to deploy, 3) the network setup phase, where the devices of the platform owners connect to the network of the network owner, 4) the application instantiate phase, where the application owner instantiates his application based on the environment in which he wants to operate the application, 5) the application enactment phase, during which the application owner deploys all necessary application components and configurations on the network, 6) the application runtime phase during

which the desired functionality actually runs, and 7) the application removal phase where the application owner removes all installed code and configurations.

Then this chapter identified the main problems in order to realize this lifecycle in a secure fashion: 1) the different roles must be able so specify their security requirements, and 2) the embedded nodes must be able to enforce this security policies and operate securely during the entire application lifecycle. It then listed the requirements for security abstractions for each role, and identified the five main data flows which are significantly impacted by the evolution towards shared networked embedded systems: network initialisation, application deployment, application management, application communication, and application service usage. For each of these flows the function of flow is explained, and the requirements of the applicable roles are detailed.

Finally this chapter identified the requirements of an environment that realizes the proposed lifecycle. It proposed an attacker model which identifies 4 different attackers: 1) an outside attacker who can only listen to and modify network messages, 2) a networked attacker who has some valid access to nodes in the network, 3) a physical attacker who can manually probe embedded devices and extract their secrets, and 4) an inside attacker who is authorised to use some limited services of the system. Next it listed a set of non-functional requirements which are briefly summarized as the evolvability and heterogeneity of infrastructure and software, transparency of heterogeneity and security, and flexibility with regards to application communication pattern and security requirements.

Chapter 3

Related Work

This chapter looks at the current state-of-the-art on management abstractions and security solutions for networked embedded systems specifically, and distributed systems in general. It evaluates these solutions to tackle the problems identified in Section 2.4: the need for security management abstractions for the different roles, and the need for a secure embedded node system suitable for a multi-party environment.

3.1 Management Abstractions

First this chapter looks at the currently available systems, languages and abstractions that exist, which potentially allow the different roles to express their security requirements in a standardised fashion. Since in order to model the security of the system, you also need to model the system itself, this section then looks at the different abstractions which are currently available to model networked embedded systems.

3.1.1 Security Management Abstractions

This section looks at security management abstractions. As stated in the lifecycle, during the policy declaration phase, the platform owner must express which application owners are allowed to use which resources, to which limits, and at which costs. To do this, a language together with a set of domain specific abstractions is necessary to express the different requirements. These

abstractions must be formally expressed in a policy document, in order to allow automated reasoning and deployment. These abstractions must then be able to be automatically parsed and enforced in all relevant systems and embedded nodes.

Many languages and frameworks offer the ability to encode different kinds of security policies in current distributed systems, such as XACML [39], SAML [11] or Ponder [26]. However, none of these languages currently offer domain specific abstractions for NES, but rather offer a language in which virtually any kind of policy and abstraction can be encoded. It is then up to a domain specific framework to provide the abstractions, parse these specifications and actually enact them. There is a limited amount of work done that aims to model the embedded environment, but it does not fit with the proposed application lifecycle. The remainder of this section will look at the different policy languages, and previously proposed abstraction models.

The eXtensible Access Control Markup Language (XACML [39]) and the Security Assertion Markup Language (SAML [11]) are two well known policy encoding languages based on XML. XACML offers a language to describe access control requirements. SAML is an XML-based, open-standard data format for exchanging authentication and authorisation data between parties, in particular, between an identity provider and a service provider. However there are some shortcomings of these standards for usage on embedded systems: 1) it lacks the infrastructure to easily express and enforce such rules in constrained networked embedded systems, 2) the parsing and execution of such rules is resource-intensive, and 3) writing policies in XACML is a cumbersome effort without adequate support. XACML and SAML have been proposed to be used in embedded networks [37], but they consider a beagle bone to be a resource constrained node. Since a beagle bone has a 32-bit 720Mhz processor, 256MB RAM and 4GB of ROM, this thesis would classify it as a resource rich node.

Ponder is a generic security policy specification language [26] for distributed systems. It allows system administrators to specify security policies that can be mapped on various access control mechanisms, such as firewalls, data bases, and potentially embedded nodes. Ponder supports policies expressing requirements with regards to user registration, user access control, and resource monitoring and enforcement. It allows expressing policies in terms of roles and groups with large organisations, or in federated organisation. Ponder is declarative, strongly-typed and object-oriented, making it flexible, extensible and adaptable to a wide range of management requirements, including networked embedded systems. While it offers a good potential language, it is an extensible general purpose language. As such, almost any security policy can be expressed in it. It does not offer the specific abstractions needed to model the security requirements, but rather offers a language which can be used to model them.

There is some related work that proposes abstraction models for expressing application level security requirements [19] for networked embedded systems. These models solve certain requirements such as access control or network security monitoring policies, but don't aim to create a full security solution, and use resource heavy certificate based approaches for enforcing the model. To the best of our knowledge there is no holistic security policy abstraction model which allows different roles in the embedded network lifecycle to express their security requirements and policies, which are then deployed and enforced in a resource constrained environment.

3.1.2 Application and System Abstractions

The application owner must be able to build and instantiate applications, during the application creation and application instantiation phase. To build an application, the owner must create an implementation which will execute the desired behaviour, and model this behaviour using some kind of modelling abstractions and language in order to easily abstract, manage and secure the desired systems. These modelling abstractions must allow the application owner to functionally model: 1) the generic application components, 2) how the different parts of the application interact with each other, 3) how the different component can be parametrised, 4) the instantiated deployment requirement, and 5) which resources these different components use based on the parameterisation. The generic model must show which components can be deployed on which kind of nodes, and which node resources are needed to deploy the component. This is necessary to enable reasoning over the deployment, and to ensure that the different nodes support the requirements.

Currently, there is no widely used standard method for modelling networked embedded systems and expressing the related security and management requirements. This lack of standard application models has potentially led to the fact that there are almost no NES security policies abstractions available. There are currently some modelling languages and frameworks, but these languages typically either only model the data produced by the embedded nodes, such as SensorML [20], or describe fairly static embedded systems, such as AADL [56]. There are many language which allow for the description of any kind of software application on an architectural level, of which UML is currently one of the most popular. There also exist generic architectural models for component based architecture, of which SCA [16] is one of the best known. However, while they provide a good starting point to describe and model the architecture, none of these architectures and modelling languages currently provide the abstractions necessary for application owners to describe all their

requirements. The remainder of this section will detail the previously mentioned technologies.

Most related work for modelling embedded networks looks at modelling the data produced by the sensors. SensorML [20] is a typical example of a data modelling language that specifies models and XML encodings to annotate data and sensors. This way it provides a framework which allows to express the geometric, dynamic and observational context of sensor systems. And while it succeeds at annotating the produced data, it does not model node resources such as ROM, RAM, or processing, nor does it model the node system. This thus prevents resource usage requirements to be expressed. Potentially the data modelling abstractions can be used to express security policies using the modelled data types, however no infrastructure is currently available which supports and enforces such policies.

The Architecture Analysis and Design Language (AADL [56]) defines a language for describing both the software architecture and the execution platform of embedded real time systems. It provides modelling abstractions to express the resource requirements of components, and the provided resources of embedded systems, and allows resource validation. However it targets single-owner, single-application systems, and thus it does not allow to express usage limitations, nor cost annotations. It also has no support to express communication security requirements for distributed applications, but rather expects the application running on top of the platform to handle this.

The Unified Modeling Language (UML [41]) is one of the most widely used architectural modelling languages, and allows to easily model and visualise the design of any system. It can be used to model virtually any system, and is often used to model embedded systems, as it is widely known. However, it only provides a visual model for expressing architectures and applications, with currently no default language encoding to allow it to be more easily exchanged between systems. Second, it is a generic model, and as such does not provide abstractions specific for embedded systems. While it can be helpful to visualise the embedded application model, it is not suited as a tool for formally encoding and annotating embedded applications.

The Service Component Architecture Framework (SCA [16]) offers a set of abstractions to describe component-based application compositions. It provides abstractions for provided and required service interfaces, offered by components, and how these services need to be connected. It also provides a set of rules on how to encode these abstractions into XML. This provides the basis for expressing applications, application level security policies, and system requirements. However, currently it does not provide abstractions to model the underlying node services required by the components.

The SCA framework has been applied to embedded networks, for example by Remora [107]. Remora combines XML-based SCA component descriptions with an implementation. The description declares the component services, references, produced and consumed events, and properties, but still lacks the underlying node resource requirements. QARI [49] is another example that adapts the SCA framework for NES. However, QARI focuses on allowing the application owners to specify higher level application requirements, so the application owners do not have to concern themselves with identifying which functionality has to be deployed on which node. It has limited support for modelling the sensors available, but has no support for modelling security requirements.

3.2 Secure Node Systems

This section looks at the related work that aims to secure the 5 data flows, which in Section 2.4 have been identified as the key data flows which are impacted by enabling sharing in the networked embedded system. These five data flows are:

1. Secure network initialisation: how can a new embedded node securely join an existing network.
2. Secure application deployment: how can a user securely deploy new embedded code on a shared platform.
3. Secure application communication: how can two distributed application components securely communicate with each other.
4. Secure application management: how can a user manage the application components on a node.
5. Secure service usage: how can the system monitor the behaviour of the users and application components.

3.2.1 Secure Network Initialisation

In the Network Setup phase, the embedded nodes must connect together and to the gateway to set up a secure network. The nodes need to be authenticated with the gateway, and network key material needs to be deployed on the embedded nodes, in order to allow authenticated or encrypted network layer messages to be sent. Many protocols have been proposed that aim to efficiently distribute and deploy keys onto embedded nodes, after which secure communication can be set up. This section first discusses the related work on key distribution and key agreement in embedded networks, then it looks at the field of Vehicle Area Networks (VANETs), because VANETS have to deal with an inherent

high mobility and connectivity for which several solutions have already been proposed.

WSN key deployment protocols can roughly be divided in two types: (1) symmetric key deployment protocols and (2) asymmetric key deployment protocols.

Symmetric Key Protocols

Symmetric key deployment protocols currently are the preferred protocols in embedded networks. They have the advantage of incurring limited overhead in both communication overhead and code and execution overhead. Hence, it is currently considered more suitable for embedded networks. Camptepe et al. [10] provide a survey of current key distribution protocols. At this time, there are two general categories of symmetric key protocols: (1) protocols with pre-shared keys with the gateway and (2) protocols with a pre-distributed key ring.

The first category of symmetric key protocol are the protocols where every node shares a symmetric key with the gateway. The gateway acts as the Key Distribution Center in this scheme and can securely deploy a group and network keys to each of the nodes. If two or more nodes want to securely communicate, they ask the gateway to generate a secret key for them and deploy it to these nodes. Examples of such a system are LEAP [125] and PAKA [113]. It is clear that this system is fairly scalable, secure and light weight. However, this system is not mobile. It is assumed that every node has a key pre-shared with the gateway. Since this thesis assumes that nodes travel between multiple networks, they will not always share a key with the gateway.

The second category is the network-only category of key deployment. In this case, there is no central trust entity that is able to deploy new keys or key material. Every node has a pre-deployed key ring or some pre-deployed key material [34] which is used to generate new keys. When two or more embedded nodes want to communicate with each other, they compare the key material on their key ring and use the shared key material to generate a shared secret. These systems can be divided into two sub categories, the probabilistic and the deterministic protocols. The probabilistic protocols have no guarantee that they have shared key material with a node nearby. If no key is shared, a trusted third party is searched which can mediate between them. The deterministic protocols guarantee that each node can securely create a key with each other node. However, in order to guarantee this, the network size must be fairly limited in order for each node to have the necessary key material. It is clear that these are light weight protocols. However, this system is not very scalable.

To reliably share a key with each node in each container, the key ring would have to be quite large. If the node only shares a key ring with other local nodes, this system would not be mobile, since new nodes would have no way of securely receiving the key ring.

Asymmetric Key Protocols

The second type of key deployment protocols are the asymmetric key deployment protocols. It was assumed that asymmetric keys require too much communication overhead and processing cost to use in embedded networks. Recently though, Elliptic Curve Cryptography has lowered the code and communication overhead required for asymmetric cryptography [2], making it a potentially viable alternative for embedded networks. Additionally, through the use of specialized encryption chips, the energy cost of encryption can be significantly lowered [106, 105]. Other related work is also looking at optimising existing algorithms such as RSA either by optimising hardware [50] or software [36]. However using asymmetric cryptography in resource constrained networks does still pose some challenges.

To achieve mobility and to securely agree keys, the embedded nodes have to verify the gateway's certificate and the gateway must be able to verify the nodes' certificates. Assuming many different Certification Authorities certify the gateways, the overhead on the node would still be quite significant. These certificates also have to be kept consistent with Certificate Revocation Lists. These additional requirements to secure the key agreement protocol cause a significant communication overhead, in addition to the significant amount of processing overhead, which should be avoided in embedded systems. So while this solution might offer an advantage in mobility and scalability, the code and communications overhead are still significant.

WiFi networks [55] are likely currently the most frequently deployed wireless networks in the world. WiFi networks can use multiple different ways to authenticate users. However, most protocols require the user to enter a pre-shared password in the device that wants to connect to the network. This password is then transformed into a secret element which is used to authenticate different parties with each other either using elliptic curve cryptography, or finite field cryptography. Once users are authenticated, messages are encrypted, typically using the AES algorithm. While allowing for easy network setup, it does require significant resources to generate all key material, and requires a pre-shared password secret to be deployed on the nodes, making it potentially too resource intensive for resource constrained embedded nodes. To be specific, one Elliptic Curve Diffie-Hellman key generation operation on a Tmote Sky

[80] takes ca 4 seconds according to Liu et al [66]. To compare, a single AES encryption operation of a 16B block takes ca 1.9 ms when performed in software on the same Tmote Sky platform and ca 449 us when performed in hardware by the CC2420 radio chip, according to Healy et al [47].

VANET Key Protocols

VANETs are ad-hoc networks of vehicles and roadside infrastructure. This unique field of wireless networks poses a great deal of challenges, among which is high mobility. Since many critical applications will require Vehicle To Vehicle (V2V) communication, communication between vehicles must be secure. Securing these vehicle networks has recently caught the attention of several researchers. New and innovative key management schemes have been proposed to meet these challenges.

Most research in VANET security proposes that each vehicle has two certificates: a permanent, global certificate and a temporary, local certificate, for example the TACK protocol [104]. The global Certification Authority is the vehicle registration authority of the country where the vehicle is registered. This CA certifies an asymmetric key pair to each vehicle it registers. However, this certificate cannot be used in a local setting due to privacy constraints. So, in order to be able to securely communicate with other local cars, the local authority provides a service where each vehicle can request a local certificate, using its global certificate. The vehicles can then use this local certificate to set up secure communications with other local vehicles.

It is clear that this solution meets the mobility requirement. Keys are deployed securely and in an ad-hoc fashion. However, it requires a lot of asymmetric key management and communication between the vehicles, the local and the global Certification Authority. The required communication and processing overhead makes this protocol unsuited for use in networked embedded systems.

Summary

To summarize, current research in secure embedded network initialisation lacks mobility, while current mobile security (VANET) research is potentially too resource-intensive to be used in resource constrained embedded networks, due to the extensive usage of asymmetric cryptography. Additionally, the pervasive presence of the Internet remains unused in almost all current network initialisation protocols.

3.2.2 Secure Application Deployment

During the application enactment phase, the application owner wants to deploy new code functionality onto the resource constrained devices. This process must happen securely, to ensure the integrity of the platform. Only components that have been approved by the platform owner should be able to be deployed. Additionally, the platform owner must be able to enforce policies with regards to which node services are available to the new code, and add some usage limitations. As evolution is often considered a vital feature in many systems, code deployment is a part of many embedded operating systems such as ContikiOS [31] or TinyOS [48], or standard operating systems such as Linux or Android. The remainder of this section first looks at secure deployment protocols for embedded systems, second it looks at some current main-stream operating systems.

Much related work in secure embedded code deployment secures the TinyOS code dissemination protocol Deluge [52], such as Sluice [65] and Seluge [54], which propose a series of hash chains or hash trees where the start of the chain/tree is signed by the basestation certificate. Extensions of this work have been proposed, namely Seluge-ImageMan [67], which mitigates the problem of rebooting to an invalid image, and by Tan et al. [108] who propose to encrypt the update blocks. The recently proposed SDRP [46] identifies that it is possible that multiple different users could potentially perform deployment of code images, but uses asymmetric encryption for authentication, and does not consider limiting code capabilities based on token or user permissions.

Secure code deployment is an integral part of many current main stream operating systems such as Android, or container systems such as OSGi [84]. In Android and other smart phone environments, users can download so-called apps from an app store. These apps are cryptographically signed by the app store, to ensure integrity of the application. These apps are sometimes verified by the owners of the app store, however this is not always done. These apps also contain a list of required services. Applications can only use the services listed in this required services list. The users who own the platform, typically cannot decide to limit usage of certain services. The decision to use an app is all or nothing, either you allow all listed service usage without limitation, or don't install it. While the typical use case of Android is a user installing something on his own system, the OSGi component framework [84] allows users to deploy new code components onto remote platforms, but the provided security is fairly similar. Components are cryptographically signed, and can only use those services that are listed in the component's manifest file.

Linux is another example of a main stream operating system that enables deployment of new code in the form of packages. A clear evolution has also

been seen in the way it manages permissions. Initially, processes could only be started as root (all permissions), or non root (limited permissions based on thread, user, and group permissions). This evolved to a system where processes could be started with a list of capabilities which would express what the process could do. However, in the current Debian Package Manager, capabilities are not enforced on a per packet basis, but rather on a per thread basis. It is thus up to the user or calling program to assign permissions to a thread. However, recently, the Docker component platform enabled a new way to package applications for Linux. Docker allows each application to run much more isolated in its own container, with possibility of per container capabilities. Linux thus has many tools at its disposal to provide secure code deployment. However, all these systems typically use asymmetric cryptography to ensure validity of code.

To summarize, most embedded network code deployment protocols don't take into account that many embedded networks are multi-user environments. They fail to allow multiple users with multiple different access rights to deploy and manage code. Recently some deployment protocols look at the multi-user aspect, but often use expensive asymmetric encryption, or neglect the need to enforce resource usage limitations.

3.2.3 Secure Application Communication

During the application runtime, messages must flow securely over the network. Depending on the type of data the application components are producing, application owners and platform owners will want to enforce different kinds of security policies, such as enforcing integrity, authenticity, and confidentiality. Additionally the platform owner will want end-to-end security, to ensure that only parties that are involved in the communication stream can produce or read confidential messages.

The state-of-the-art has provided many solutions to enable secure communications in networked embedded systems [18, 98]. The current related work can generally be subdivided into developing new and more efficient algorithms such as Noekeon [23, 53], creating new protocols specifically designed for NES, or altering existing well known protocols for NES usage. The remainder of this section first looks at new protocols designed for NES, and then at current common protocols, which are being adapted.

Currently popular protocols for securing embedded networks are Zigbee, LEAP, and TinySec. Zigbee [35] has different types of keys for different usage. Zigbee differentiates between 1) node master keys, which are unique per node and are used to set up other keys, 2) link keys which secure the communication links between multiple nodes, and 3) network keys which secure network wide

communication. However, Zigbee assumes that the master keys are pre-installed reducing mobility and evolution. Additionally Zigbee does not mention multi-user or multi-party abstractions. At network setup, the node is authorised by the Trust Center (TC) using the master key. Once authorised, network and link keys can be established.

LEAP [125] identifies 5 different types of keys depending on usage: 1) basestation keys, 2) pairwise node keys, 3) cluster keys, 4) group keys, and 5) master keys. At deployment each node receives the network master key. This key is then used to set up the other keys. Once the setup time is passed, all nodes delete the master key. This prevents new potentially malicious nodes from joining, but also makes network evolution very difficult.

TinySec [58] offers flexible security by offering two modes of security: an authentication only mode, and an authenticated encryption mode, allowing the network owner to decide which mode to use. It however does not offer a key management protocol, but can be used with any protocol available. It is also impossible to create different channels based on application level security requirements. Thus while offering secure communication, none of these protocols offer networking evolution and management, nor take into account multi-party ecosystems with heterogeneous security requirements.

So in many systems, you can only set a general communication security policy, without being able to distinguish between different data flows [125],[58]. However, from a functional point of view, application owners might need different security policies depending on the data type. For example the application owner could only require authenticity on environmental data, while requiring confidentiality for identifiable information. Current related work however does not offer such content dependant security policies. Some related work does offer some level of flexible security policies, such as the previously discussed TinySec, or Zigbee. In most cases however it does so on a network or link level, without any knowledge of the data that is flowing over it, or being able to adapt based on it.

One class of related work that offers E2E data security based on application type is Attribute Based Encryption protocols, such as for example FDAC [122]. Data items are encrypted using the relevant attributes associated with that data. Then only parties that have the necessary access structure, in the form of private keys, can access the data. This enables a limited amount of flexible security, at a significant cost due to the usages of elliptic curve functions.

Two common protocols for securing data communication on the Internet are TLS/SSL and IPSec. Transport Layer Security [28] (TLS) and its predecessor, Secure Sockets Layer (SSL), are standardized cryptographic protocols that aim

to secure client server communication over the Internet. It operates at the presentation layer (layer 6) of the OSI stack, and as such is typically reserved for End-To-End client-server communication. TLS/SSL requires significant amount of asymmetric cryptography in order to operate, making it very resource intensive for embedded networks. DTLS [97] is similar to TLS, but optimised for datagram communication, which is currently considered the most common form of communication in embedded networks. DTLS has been proposed for embedded networks, and some research is looking in how to optimise DTLS for resource constrained networks [62].

IPSec [60] is another commonly used network security protocol. It operates on the network layer (layer 3) of the OSI stack, and allows for much more flexibility in which endpoints it secures. It can create secure links between multiple networks, or between a host and a network, or between hosts. It offers flexible security: the network administrators can decide between encrypting and/or authenticating all traffic between endpoints on an IP level, by specifying so-called security associations. Additionally network administrators can specify which application traffic must travel over which security association. While this protocol can meet the functional requirements of having flexible security based on the semantics of data transmitted, it requires a significant amount of features to be implemented in order to be standard compliant, causing significant overhead for resource constrained embedded nodes. Research has proposed simplifications to the protocol to make it more suitable for resource constrained environments, but has not yet seen main-stream adoption [95].

3.2.4 Secure Application Management

During the application enactment phase, and the application runtime phase, application owners need to be able to manage their applications, and platform owners need to be able to manage their platforms. Since there are multiple different users and components on an embedded node, it is crucial to isolate the different users from each other and enforce limitations on their capabilities. The current research in access management can again be divided into the symmetric key approaches and the asymmetric key approaches. The remainder of this section first looks at some common symmetric key approaches, next at some asymmetric key approaches, and lastly at some access control frameworks currently used in large scale enterprise networks.

The first subset are the symmetric key approaches, such as sAQF [119] and SpartanRPC [15]. sAQF adds user authentication codes to ensure that only authenticated users can use node services. It assumes a single user of the network who has a large key pool, who installs a unique subset of this key pool

on each node. To authenticate a message, the user calculates a 1 bit MAC with each key, and concatenates all these MACs into a single large MAC. The user can then broadcast this message to the network, and each node can verify the message by verifying parts of the MAC using its own key pool. This ensures that a breach of any single node does not allow the attacker to send authenticated messages. However the system does not mention how it actually enforces access control.

SpartanRPC [15] secures RPC service usage. To use a service on a node the user must have the secure capability token, which is basically a symmetric key. When using the service, the user must add a MAC signed with that key. While it offers resource constrained access control, it is burdensome to apply fine grained authorisation decisions due to the requirements of many keys, and the disclosure of one user's keys requires a rekeying of all the capabilities to which he had access. Additionally, the security measure must be added at a language/interface level, breaking the transparency requirement.

Alternatively there have been many proposals using asymmetric cryptography approaches. One subset proposes to use boom filters to verify signatures [96]. Other related work proposes Authenticated Querying, where messages are signed with a certificates issued from a single network CA [7], or ring signature authentication [45]. While the asymmetric key schemes offer multi-user authentication, they are often too heavyweight for efficient use in constrained networked embedded systems. Most schemes also do not offer a clear management infrastructure allowing evolution of access rights, or neglect authorisation altogether.

An often used access control system for user authentication in back-end systems is Kerberos [81]. In Kerberos, a user who wants to use a service must first authenticate himself with the Authentication Server. The way this authentication happens is very extensible, but can take the form of a simple username-password combination, credentials or other authentication methods. If authenticated, the Authentication Server returns a Ticket Granting Ticket (TGT). Using this TGT, the user can then send a request to the Ticket Granting Service (TGS), requesting access to a specific service. The TGS then returns a client-service token, and the client-service key. The user can then authenticate himself to the service using the client service token, and encrypt his ensuing communication with the client-service key. While this approach offers a lot of potential, it does pose significant overhead when deployed in larger organisations. For example, the default token size in Windows 7 is 12 000 bytes, which for some organisations is even not enough to encode everything.

3.2.5 Secure Service Usage

During the application runtime phase, multiple applications of different owners are concurrently operating on embedded nodes. To ensure that no single application or user uses too many of the node's limited available resources, either accidentally or intentionally, it is necessary to monitor these applications and users, and in case of violation, enforce resource restrictions. Additionally, the owners of the nodes often want to log the resource usage of users and applications, in order to bill the different users for their actual service usage.

The state-of-the-art with regards to application service usage monitoring and enforcement can be divided in two main categories [17]: 1) active node monitoring, where the embedded system monitors itself, and 2) passive node monitoring, where a secondary system monitors the network. The currently available solutions however have significant drawbacks in the proposed multi-party ecosystem, and seldom offer resource limitation policy enforcement.

A number of active monitoring solutions for distributed networks is currently available. A prototypical solution is the Simple Network Management Protocol [13]. This is the current standard solution for many kinds of networks. It allows a user to request information such as monitoring data from networked devices. The cost of this approach however is significant, since it is not optimised for constrained devices. A solution that is targeted to embedded devices is Sympathy [94]. This system is mainly used for debugging and allows fine-grained collection of different metrics such as next hops, neighbours, uptime, etc. While it offers many tools to debug a system during testing, it does not offer any infrastructure to perform fine-grained monitoring and enforcement of application services.

Passive monitoring has often been proposed for networked embedded systems, since this reduces the burden on the nodes which perform the actual application functionality, and can be deployed as and when needed. Most approaches operate by having a secondary network of nodes in the environment, which monitor all traffic, interpret messages and relay them back to a central server which does the processing. Current solutions exist which either use an Ethernet infrastructure [64] or Bluetooth [6]. While these approaches can indeed monitor embedded networks without any node overhead, they do require the deployment of a costly and redundant secondary network, cannot and should not be able to decrypt end-to-end encrypted messages, cannot detect transmission issues, cannot monitor the service usage of the application components, and have issues with node mobility. For these reasons this thesis considers passive monitoring to be unsuited for the targeted functionality.

3.3 Summary

Related work has proposed some partial solution for modelling stakeholder security requirements, and application level security policies. Much related work exists on creating different secure node subsystems, such as secure network initialisation, secure application deployment, or secure application communication. Yet currently, no comprehensive embedded node security framework has been proposed offering 1) a set of abstractions to express and a framework to enforce application owner, platform owner, and network owner security requirements, and 2) a secure node system that offers secure network initialisation, code deployment, node management, application communication, and application service usage. Neither has it been shown that such a secure node system, supporting shared usage, can be created on top of the class of resource constrained embedded platforms.

Chapter 4

Architecture

This chapter provides an architectural overview of the distributed security infrastructure. The security infrastructure consists of 1) the end-user tool, 2) the application owner server, 3) the platform owner server, 4) the network owner gateway, and 5) the embedded node, as shown in Figure 4.1. The rationale behind this, is that each role has a server which stores the data model for that role, and executes the necessary continuous processes. These servers are managed by the end-user, and manage, monitor, and receive data from the embedded nodes. This modularisation allows a party to instantiate the servers for those roles that it wants to support. Next, this section lists the structure and responsibilities of each of these systems. Finally, this section details the required security modules on the embedded node to secure node communication, and provides an example scenario detailing how these modules interact based on the smart logistics use case.

4.1 End-User Tool

The end-users must be able to detail their requirements and functionality for the different roles that the end-user wants to take on. For this purpose, an end-user tool was developed. The tool is comprised of three sub modules: 1) the application owner module, 2) the platform owner module, and 3) the network owner module. Each module can potentially be isolated in its own application, however, to ensure the client only has to install one application onto his system, the end-user tool bundles the functionality of all three modules. In this architecture, the tool is a single stand-alone application, but it would be

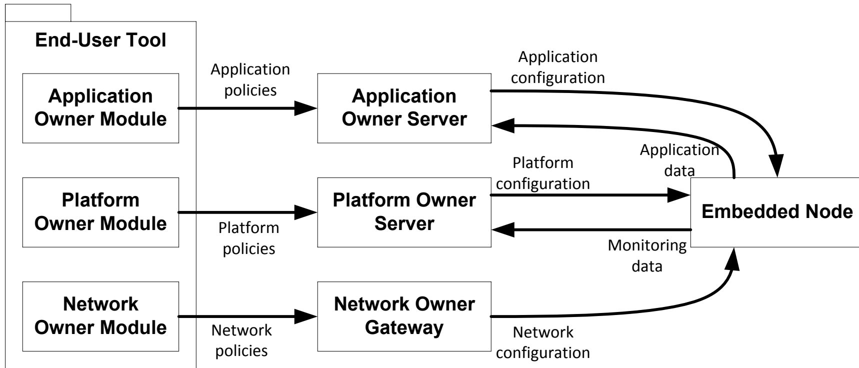


Figure 4.1: Architecture of the distributed security infrastructure.

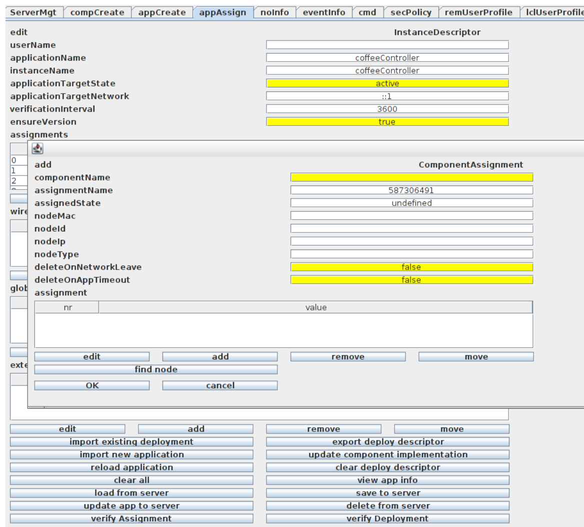


Figure 4.2: Screenshot of the current implementation of the end-user tool.

possible that the end-user can just browse to a server, and perform the required actions in a web based client. In that case, each module would then be the website that the user retrieves from the relevant server. This section will now detail the features that each module must support, based on the requirements listed in Chapter 2. Figure 4.2 shows a screenshot of the current implementation of the end-user tool.

The application owner module must allow users to easily create, deploy, and

remove applications. This consists of the following duties, listed in the order of the proposed application lifecycle:

1. Specify which platform owners and network owners are trusted to use the required resources, and specify the necessary security policies.
2. Create, import and export reusable components and applications.
3. Get node information from trusted network and platform owners.
4. Plan and verify the application deployment and select on which nodes the application should run. The module must assist the user during this process, automating node selection based on user requirements.
5. Store the deployment plan on the application owner server.
6. View the current application state.
7. Change the deployment plan based on updated requirements.
8. Remove the application.

The platform owner module must allow users to manage the platforms they own. This consists of the following duties:

1. Maintain a node repository which for each node describes which services the node provides, the installed keys and other security information.
2. Specify which application owners are allowed to use which services, at which cost, and with which limitations.
3. Specify which network owners are trusted.

The network owner module must allow users to manage the networks they operate. The user must be able to:

1. View current available nodes in the network together with owner.
2. Set cost, requirements, and security level of the network.
3. Specify which platform owners have access to the network.
4. Specify which application owners have access to the node repository.

The end -user tool must assist the user in expressing his security requirements based on domain specific abstractions. Next the tool must encode these specifications in a machine-readable format, such as JSON or XML, and store them on the relevant owner server for storage, processing and enforcement.

4.2 Application Owner Server

The application owner server consist of 1) the application manager which manages the applications which are operated by the application owner, 2) the policy repository, which stores the application owner's policies, preferences and

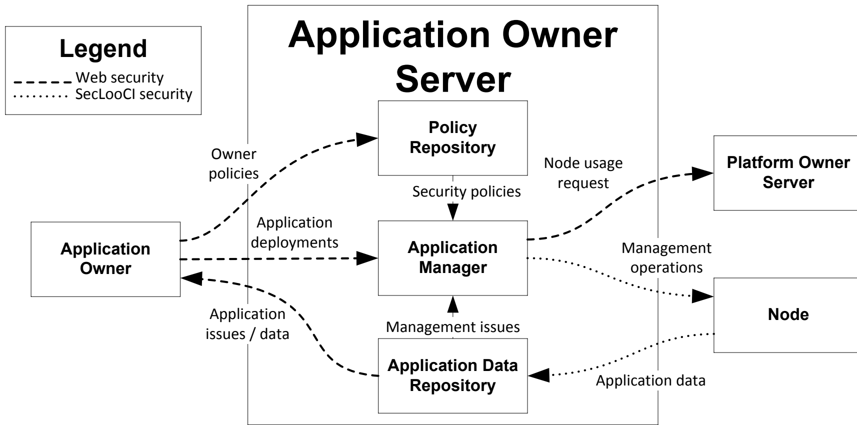


Figure 4.3: Overview of the Application Owner Server.

application templates, and 3) the data repository, which stores the application data provided by the embedded nodes, as shown on Figure 4.3.

The application manager is responsible for managing the applications the application owner wants to deploy. This consist of four duties: 1) verifying that deployments are feasible and committing the application usage of the deployment to the relevant platform owners, 2) deploying the necessary code, configuration and parameters on the required nodes, 3) continuously verifying that the application is still in the desired state, and 4) cleaning up applications when the application owner does not need them anymore. To deploy an application, the application manager will first request the necessary permissions and access rights from the platform owner server. When cleared, it proceeds with deploying the application. When a problem is detected, either because the embedded network changes (e.g. a node reset) or the application owner changed the requirements, the application owner server will take the necessary reconfiguration actions to reestablish the desired functionality. If the server is unable to reach the desired state, it must notify the application owner. When the application owner wants to delete an application, it notifies the application owner server. The application owner server then removes all relevant code and configurations, and notifies the relevant platform owners that the application is deleted. The platform owner can then gather the relevant monitoring data, either from database, or recent data from the nodes, and provide the application owner with a final bill for his usage. The application owner module then reports this figure back to the application owner, who can pay the cost.

The policy repository stores the general security and trust policies of the

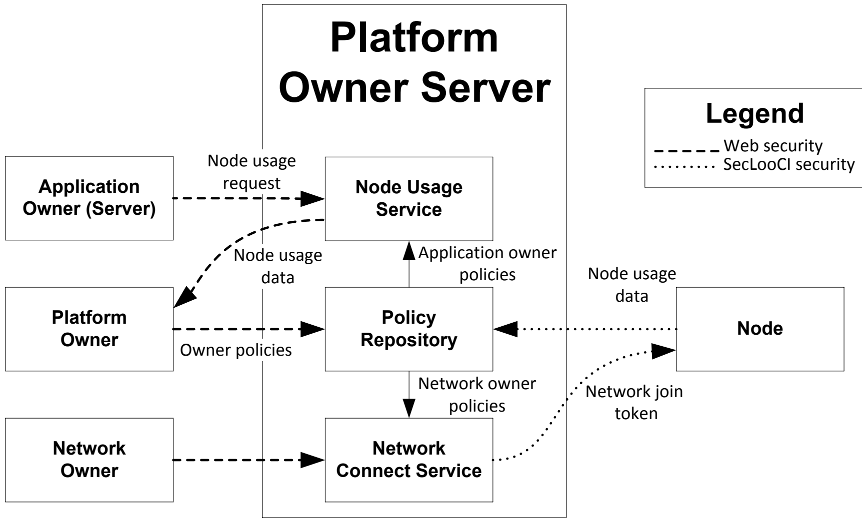


Figure 4.4: Overview of the Platform Owner Server.

application owner. These policies are taken into account when verifying applications, and contacting platform and network owners. Second the data repository must also store the components and generic applications the application owner can potentially use. The application owner can then instantiate these applications, and deploy these components when he wants to establish certain application functionality.

Lastly, the application owner server must contain an application data repository. One of the goals of embedded applications is typically to gather and process data. The data storage acts as the data end-point, where the application owner's embedded application components can send their data to. The data storage can then process this data, notifying the application manager in case of potential management issues, or notifying the application owner in case of application issues (such as for example a fire alarm). The data storage also provides a data retrieval service where the application owner can retrieve the data logs and perform additional processing.

4.3 Platform Owner Server

The platform owner server consists of 1) a policy repository, which stores the platform owner's policies with regards to node, network and application usage,

2) a node usage service, which provides a service for application owners to request node information, commit their node usage, and consult costs, and 3) a network connect service, where network owners can report that nodes have arrived in their network, as shown in Figure 4.4.

The policy repository contains the platform owner's policies with regards to nodes, application owners, and network owners. It contains the descriptions of the nodes, and the current state of the nodes, which consists of the current location of the node, the currently deployed applications and configurations present on the node, and the currently available and used resources. The platform owner's application owner policies dictate which nodes and resources application owners can use. Finally the platform owner's network owner policies dictate which network owners the platform owner trusts sufficiently in order to use the network owner's network.

The node usage service stores the current node usage of the application owners, and provides the application owner a service to request and commit node usage. When an application owner requests node information, this service must return the embedded nodes which are available to that application owner, the currently available resources, the resource limitations and the associated costs. The service must also allow application owners to commit and update their node usage, remove old deployments, and consult the current or total accumulated cost.

Finally the network connect service provides the network owners a service to update mobile node locations and instantiate trust relationship between nodes and networks. This service is used when a new node enters the network of the network owner. This service enables the network owners to report the new location and IP address of a given node and can be used to set up a trust relationship between the platform owner, the network owner, and the mobile node, by exchanging key material, policies and tokens in order for the nodes to securely join the network of the network owner.

4.4 Network Owner Gateway

The network owner gateway consists of 1) a policy repository which stores the network and security policies of the network owner, 2) a network join service where new nodes can join the network, and 3) a local node repository where application owners can consult which nodes are in the network, as shown in Figure 4.5.

First, the policy repository stores the network owner's policies on 1) network

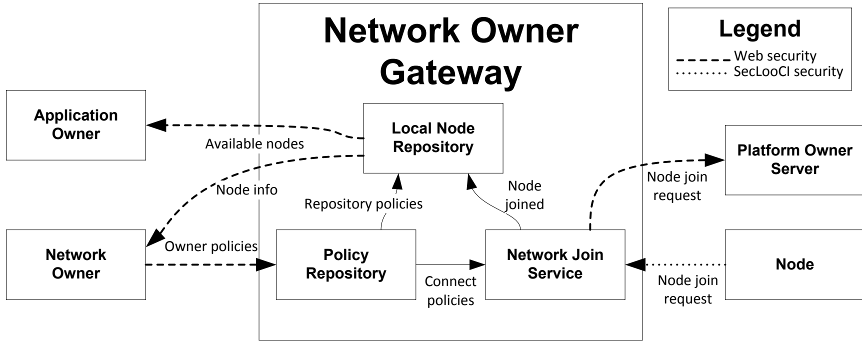


Figure 4.5: Overview of the Network Owner Server.

settings such as which network algorithm is used, on which channel, which time slots etc., 2) network security policies such as the MAC layer security level, algorithm, and re-keying frequency, 3) user policies such as which platform owners or third party trust providers the network owner trusts to use its network, and which application owners are allowed to use the node repository, and 4) some network meta-information, such as which locations and zones are present in the network.

Second the network join service is the point of contact when a new node enters the network. When the node joins, it first contacts the network join service, and transmits the owner's identity. The network gateway must then verify that the party is trusted. Next it must set up a secure connection with the node, and deploy the necessary network settings and security policies. This happens through the platform owner server, which is then also notified that the node has migrated to a new location. Once the security and network setup is done, the new node can communicate over the network, and potentially to the Internet if the gateway provides an Internet connection.

Last, the local node repository provides an inventory of the currently present nodes and their owners. This enables application owners who want to deploy an application locally on the network of the network owners, to query which nodes are available, and who they need to contact for more information. The network owner can potentially filter the available nodes based on the publicly available services, or known trust relationships.

When application owners want to deploy applications on the network of the network owner, the application owners need to be trusted by the different platform owners in the network. In this case, the network owner can act as a third party trust provider. The platform owners can allow all application

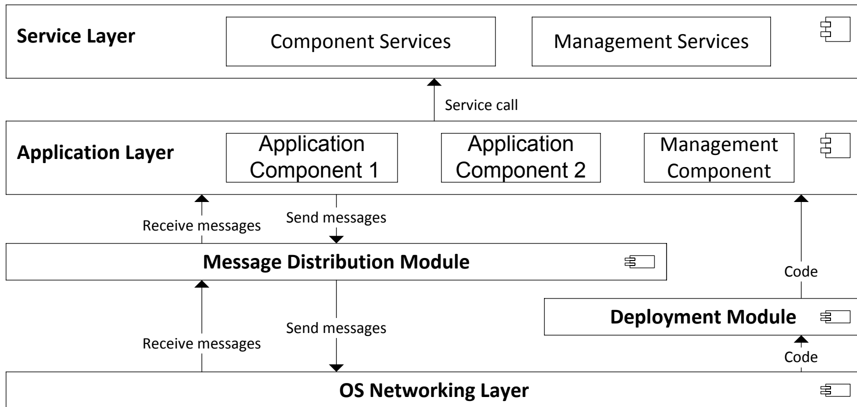


Figure 4.6: Overview of the Basic Node Architecture.

owners that the network owner trusts, to deploy applications on those nodes while they are in the network of the network owner.

4.5 Embedded Node

The embedded node is responsible for gathering sensor data, performing local actuation and computation, communicating the results with other embedded nodes, sending application information back to the application owner, and sending application monitoring information back to the platform owner. In order to be able to handle evolving requirements, such as changes in the environment, application owners and platform owners must be able to deploy new functionality onto the node, and manage and change existing functionality. Additionally they must do so adhering to the security policies provided by both the application owner and platform owner. The LooCI component architecture provides the functional support, but currently has no security provisions. This thesis will take the LooCI architecture, and add the necessary security systems to enable secure operations. The remainder of this section first details the LooCI [51] non-secured node software architecture, next this section proposes 7 additional modules to be added to the node software architecture to enable the secure operation of the embedded node platform.

4.5.1 LooCI Node Architecture

The basic node architecture must enable evolvable applications to perform sensing, actuation, and computation, and communicate the results with other embedded node systems, or back-end infrastructure. The architecture must enable evolability of applications and settings. As such, this thesis chose to build upon the LooCI architecture, which provides a parameterizable component model which communicates over a distributed event bus, as application model, because it allows the easy encapsulation of functionality, is easy to reason over, and allows clear modelling of both required resources and communication channels. These components perform the sensing and actuation by using Operating System services which allow reading or setting of digital or analog signals. These components can then process the data, and communicate the data over a distributed event bus. The distributed event bus has policies which specify how an event should be distributed i.e. to which nodes and components the event should be sent.

The LooCI component middleware supports both course grained and fine grained evolution. Course grained evolution is achieved by adding and removing code binaries on the node. This allows entirely new functionality to be deployed on the node. Fine grained evolution is achieved by either reparameterizing the components, which changes the functional behaviour of the component, or changing the distribution policies in the event bus, which changes the way data is communicated.

The components of the middleware are shown on Figure 4.6: 1) an OS networking layer which provides basic networking and routing functionality, 2) a deployment module that allows new code to be deployed, 3) a message distribution module that emulates a distributed event bus by distributing events to the correct component or node based on event semantics and stored policies, 4) an application layer which contains multiple application components, and 5) a service layer which provides OS and middleware services. Application components are responsible for interpreting messages they receive and then call the necessary services to gather information or enact change depending on the event. The middleware services are responsible for managing the application components and the message distributions policies. The OS services offer sensing, actuation, timing, memory access and other services to the components.

This thesis assumes that all nodes can communicate with each other using a standard physical, data link and networking layer, such as IEEE 802.15.4 and IPv6/6lowpan. While currently there is still significant research being done in optimising these layers, there is a clear need for common standards to emerge in order to enable the advanced sharing and collaborative scenarios this thesis

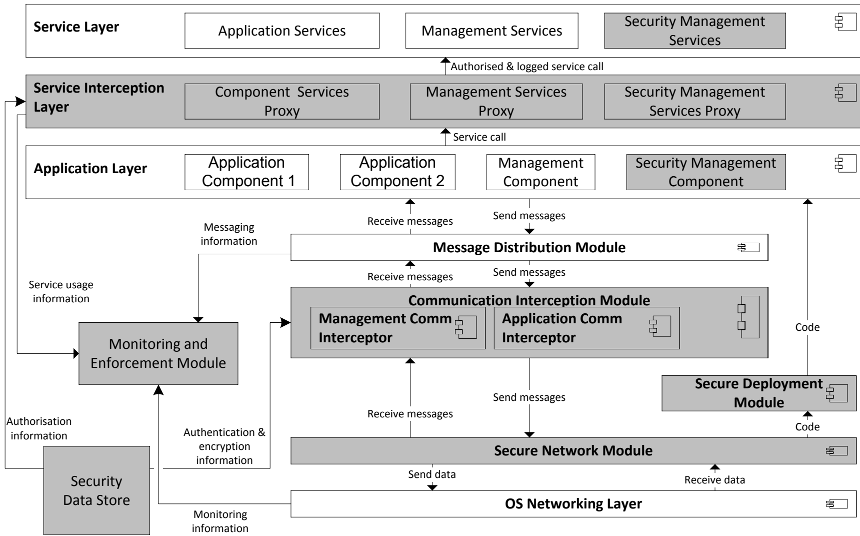


Figure 4.7: Overview of the Secure Node Architecture.

proposes. The OS networking layer is also responsible for joining networks and exchanging MAC layer messages.

While this thesis looks specifically at the LooCI middleware, the basic architecture can be extended towards other component based node systems. A good other example is a component based CoAP or HTTP server which allows GET, PUT, POST and DELETE commands. A typical example for resource-constrained embedded networks is the Californium CoAP Framework [63], which can be considered the low resource alternative of an Apache Tomcat server. When the server receives a message, from the OS networking service, it sends it to the message distribution module. This module then looks at the PATH of the message, and potentially at the command, and delegates the event to the correct component for event handling, which is called the local endpoint in Californium terminology. The application component then looks at the PATH, command, and message content, determines the required operation, performs the requested actions, and returns a reply. Californium supports the deployment of new code binaries, which can then register themselves with the message distribution component for a certain PATH and command.

4.5.2 Securing the Node Architecture

The secure node architecture must enable the basic node architecture to perform its duties, and augment this system with security. This thesis specifically focusses on securing the basic node system in a shared environment. As such, this thesis has identified five data flows which must be revisited with regards to security at 2.4.2: 1) network initialisation, 2) application deployment, 3) application management, 4) application communication, and 5) service usage. To secure these five data flows, this thesis proposes to augment the basic node middleware architecture with the following seven middleware modules, which each contribute to securing one or more of the data flows, as shown on Figure 4.7:

Secure network module: secures the network initialisation data flow, enables secure joining, and management of wireless network.

Communication interception module: authenticates and encrypts the application and management messages based on installed policies.

Service interception layer: provides authorisation and monitoring of the application and management services.

Security data store module: stores credentials, access policies and usage policies for the different users, applications and services.

Secure deployment module: secures the application deployment data flow to allow multiple different parties to deploy application components.

Monitoring and enforcement module: registers the usage of the different application components and node users, and potentially limits them.

Security management module: allows administrators to add and remove users, parties, roles, and security policies.

The rationale behind these modules are 1) to add secure communication layers on top of the current communication layers, 2) change the deployment protocol with a secure version, and 3) add an authorisation and audit framework based on the guidelines in RFC 2904: AAA Authorisation framework [115]. The remainder of this section details for each of these modules their roles and responsibilities, how they contribute to securing which data flow, and how they interact with each other.

Secure Network Module

The secure network module secures the network initialisation data flow, and further behaves as a secure networking layer on top of the normal networking layer. This is similar to the way WiFi [55] initialises a secure network and

provides a secure network layer for the layers and applications above. The module secures network level messages and ensure that outsiders can not read any messages in network. While technically this module can provide secure routing and other secure networking services, this thesis considers them out of scope. Ample related work exists that proposes solutions for these issues, and the data flows required for these services are not significantly impacted by sharing.

The main duty of this module is to join the available networks in a secure fashion. The module will detect when there are new networks in range, and register the node with the network owner gateway. The network owner can then negotiate with the platform owner or with the node to set up a secure relationship and deploy the necessary network settings onto the node. Once the network is set up and the security key material is deployed, the node can be considered part of the network, and the secure network module can provide a secure network layer to the other modules.

Communication Interception Module

The communication interception module provides end-to-end secure communication for both the application communication data flow, and the management communication data flow. This can be considered similar to the way TLS [28] provides end-to-end security for application level protocols such as HTTP on top of a TCP network. However, in this case, the module provides policy driven end-to-end security. To gain a clear understanding of the duties of this module, the communication interception module is divided into two sub-modules, the application communication module, and the management communication module. They are however encapsulated in a common wrapper that identifies which type of event arrives at the communication interception module based on event meta-data, and then delegates it to the correct sub-module. Both also use the same cryptographic services to encrypt and decrypt events.

The application communication module secure the application events. It does so based on the policies set by the application owner or platform owner. The module distinguishes between four different policy levels: 1) not secured, 2) integrity protected, 3) authenticity protected, and 4) confidential. The module must be able to apply the necessary policies to outgoing traffic, doing the necessary cryptographic operations, and be able to require certain incoming communication to have a certain level of security, depending on the type of the data. The module must enable new policies to be added for both incoming and outgoing traffic, change keys, and remove outdated policies.

The management communication module secures the management communication. This communication must always be authenticated and authorised, to prevent anyone from accidentally or intentionally turning off management security. The module must be able to verify that a message is actually sent by a user. All incoming and outgoing messages must be encrypted and authenticity protected, to ensure that inspection information cannot be disclosed, nor that messages can be altered in transit. The management communication module ensures that all management messages sent up to the message distribution module are authentic, which is necessary for the Service Interception Layer to enforce authentication policies.

Service Interception Layer:

The service interception layer is the Policy Enforcement Point: it enforces that only an authorised entity can access services, and as such provides security for the application runtime data flow, and the application management data flow. However, depending on the flow which is secured, and the service which is protected, the service interception layer will have to provide different operations, and a different type of proxy is inserted, based on the type of service. For application services, a service proxy is inserted, for management services, a manager proxy is inserted.

The service proxy monitors and limits the service usage of application components. These are components that provide application functionality to the application owner. The service proxy ensures that only components that are allowed to access the service, are able to do so, that the service usage is within policy limitations, and that the usage is reported. When a component requests using a service, the service proxy contacts the monitoring and enforcement module. The monitoring module consults the platform owner policies and previous usage. If the component is allowed to use the service, and ample limit is available, the node usage is allowed. When the service returns, the proxy logs the resource usage with the monitoring module.

The manager proxy authorises, monitors and limits the service usage of users accessing node or middleware management services. The manager proxy ensures that only authorised users can access management services, and that the usage is logged and within limits. It does so similarly to the service proxy: when a request is executed by a management component, the manager proxy checks on behalf of which user the management event is executed. The user's authorisation is validated, and the monitoring module is consulted to ensure that ample resources are available. If allowed, the request proceeds. When the request returns, the resource usage is sent back to the monitoring module. This module depends on

the Communication Interception Module to ensure only authenticated message arrive at the management components.

Security Data Store

The security data store is the Policy Information Point of the authorisation framework: it is responsible for storing the information and policies with regards to authentication, authorisation and audit. This is comprised of: 1) the necessary data for authenticating and authorising users, such as their IDs, their access rights, and their key material, 2) the monitoring data of users, 3) the access rights of the different application components, and 4) the monitoring information of the different application components. This information is provided by the service interception layer, which provides user and application monitoring data, the secure deployment module, which provides application limitation policies, and the security management module, which can manage the installed users and application policies. The data in the security data store is used to decrypt and verify messages in the communication interception module, and is used to perform access checks in the service interception layer.

Note that the security data store is not a large database of text based XML policies, but rather a small data store of binary policies and compressed information stored in the RAM of the embedded node.

Secure Deployment Module

The secure deployment module ensures that only approved code can be deployed onto the sensor node. This module basically replaces the non secure deployment module by adding security to the deployment protocol. The secure deployment module also provides the initial limitation policies of newly installed applications components to the security data store. This includes which services are allowed to be used by the application component, and any potential service limitations.

The secure deployment module operates based on a token system. Any party that wants to deploy an application component, must first prove that it is allowed to use the deployment service, by sending a deployment token. This deployment token must contain the identity of the owner of the codebase, the allowed services, any service limitations, a time limitation of the token, a time limitation of the deployment and proof that the deployment of the code is approved by the platform owner. The token deployment system allows for secure offline deployment. The application owner must be able request the permission to deploy beforehand, and then later use the token offline.

Monitoring and Enforcement Module

The monitoring and enforcement module is the Policy Decision Point of the authorisation architecture [115]: it provides the central decision module where monitoring data is gathered, and service limitation decisions are made. It ensures that no component is able to use excessive amounts of memory or CPU, nor able to read sensor resources too many times, causing too much energy usage. The module offers two services: a service monitoring service, and a policy enforcement service.

The service monitor receives information from the service interception layer about which applications and which users are using which services. This service then stores this to ensure it is available for the policy decision service. This information is also sent to the platform owner in order to enable charge-back and provide node usage visibility. The data acquisition part must be customizable based on the platform owner requirements. Some platform owners might require that all resources are monitored, at a slight cost to additional storage and computation resources. Other platform owners might require that only a small subset of resources are monitored, or none at all, to maximally preserve the resources available. It should also be possible to specify aggregation policies, so only aggregated data is maintained.

The policy enforcement services decides whether a new request to use resources is allowed. When the service receives a request from the interception layer whether or not to allow a certain call, it must consult its policies and runtime information: it retrieves the relevant policies from the security data store, checks that a component or user is permitted the resource. If permitted, the engine checks that the user or component has not yet used too many of that particular resource by querying the stored monitoring data. If there is still sufficient buffer for the request to happen, the request is permitted.

Security Management Module

Finally the security management module is the Policy Administration Point: it is responsible for managing and inspecting the different security modules. The management module allows users to 1) add, edit and remove users and permissions on the platform, 2) inspect and edit the service limitations of both users and components, 3) inspect the current usage of applications and users, and 4) add, edit or remove communication policies with regards to different application level data flows. The security management module is thus basically a service that enables authorised users to query and modify the data and policies contained in the security data store.

4.6 Example Scenario

To clarify how the different modules interact with each other, this section presents a simple yet typical embedded network scenario, inspired by the previously discussed smart logistics use case. In this scenario, the harbour customs authorities want to monitor the location of all containers. To do this the harbor customs will deploy a location calculation service on all smart containers present and use radio triangulation to calculate the current location based on static beacons.

This scenario showcases multiple parties sharing a common infrastructure. The harbour customs want to deploy an application on a shared network, which makes them the application owner. The harbour owner provides a network for all the containers, which makes him the network owner. Finally the logistics providers who own the containers will provide the actual platform, which makes them the platform owners.

When looking at the larger picture however, the scenario becomes more complex. The logistics provider has applications of his own running on his sensor nodes, and potentially also wants to use services of other containers around his containers, making him also an application owner. The harbour owner will also have some static embedded nodes installed to monitor the harbour, making him also a network owner and platform owner. The customs officers can then use these static nodes of the harbour to provide the necessary beacons, as shown on Figure 4.8. The harbour owner might then too want to monitor the state of the containers, so he too installs an application on the shared platform. However, to provide a clear example, the remainder of this section only looks at the simplified and isolated container localisation scenario. The remainder of this section goes through the different stages of the lifecycle as identified in Section 2.3 and identifies how the different security modules identified in the previous section interact with each other.

4.6.1 Policy Declaration

The first stage is the policy declaration phase. In this phase the different parties (harbour customs, harbour owner, and logistics providers) must specify which parties they trust. The harbour customs (application owner) specifies which logistics providers (platform owners) he trusts, and that he trusts the harbour authority (network owner). The harbour authority must specify a trust relationship with the harbour customs, and the different logistics providers, and must specify to which extend the logistics providers can use his network. Finally

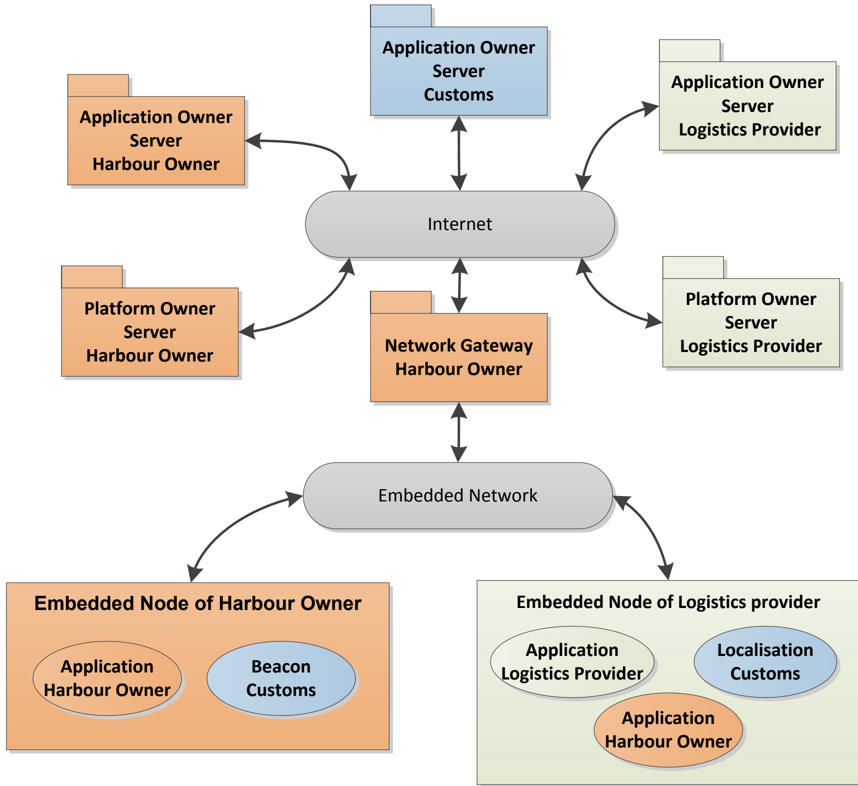


Figure 4.8: Overview of the harbour scenario.

the logistics providers must model his nodes, and their available resources, and specify that the harbour customs is allowed to use the services of his nodes.

After a user enters these specifications in the end-user tool, he saves them on the respective server (network owner gateway, platform owner server or application owner server). Naturally the connection will be secured: for these interactions, standard Internet security protocols such as TLS or IPSec for secure communication and Kerberos or Shibboleth for authentication and authorisation can be used. Once the policies are stored on the server, the other parties can retrieve the policies, and the policies can be checked when other parties ask permission to use certain node or network resources.

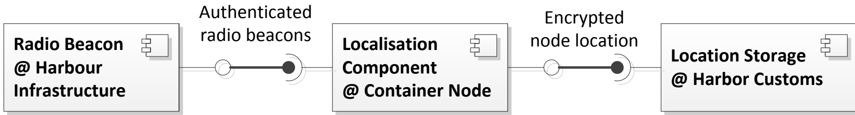


Figure 4.9: Example Localisation Application.

4.6.2 Application Creation / Acquisition

Second, the harbour customs must either create or purchase the localisation application. This localisation application will be comprised of different components which will communicate with each other. This case proposes a simplified localisation application which is comprised of the following components, as shown on Figure 4.9: 1) a beacon components, located at multiple fixed locations in the harbour, which send out timed and localised beacons signals, 2) a localisation component which is deployed on the asset to localize that listens for beacons, calculates the locations, and publishes it, and 3) a location storage component that is subscribed to all localisation events, and stores them for further processing.

Each of these components must be able to be deployed on multiple different types platforms, and must specify which exact resources and services are required for execution. Due to the potential heterogeneity in the different parts of the network it is necessary that either generic components are available that can be deployed on multiple platforms (e.g. a java environment), or in case of embedded devices, a specific component must be available for most common platforms. Especially the localisation component will likely be deployed on many different platforms, and as such could likely be required to interact with multiple underlying platforms. Second, each component must also specify which resources and services are required from the underlying platform. This is necessary in order to ensure that the harbour customs can reserve the necessary resources from the different platform owners, and that it can plan in advance on which nodes it can or cannot deploy the application.

4.6.3 Network Setup

Next the network must be set up. At the initialisation of the network, the harbour authority will make a central gateway available, where new nodes will be able to connect to in order to get Internet access. When a new container enters the harbour, the embedded nodes can start the network setup, as shown on Figure 4.10. Each node will contact the gateway, notifying it that it is present, which node it is, and who its owner is. The gateway will then verify

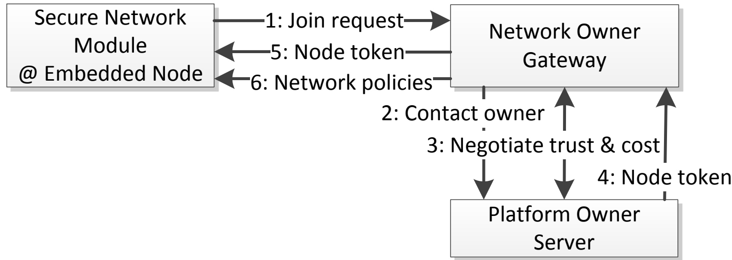


Figure 4.10: Network setup data flow.

that the owner of the node is trusted, based on previously entered policies. If trusted, the gateway will contact the logistics provider notifying that the node has arrived. The logistics provider verifies that the gateway is of a trusted party, and provides the party with a token or credentials that the gateway is trusted.

Next the gateway can use these credentials or this token to contact the secure network initialisation module and deploy the network key and other network settings onto the embedded node. Once done, the node is considered part of the network, and is added to the network owner's node repository. The network owner then also notifies the harbour customs that a new container node has arrived.

4.6.4 Application Instantiation

The harbour customs server is notified that a new node has entered, and reconfigures his localisation application with the new node. The server contacts the logistics provider's platform owner server to request which services are available for the localisation application. The logistics provider server and harbour customs server mutually authenticate each other and negotiate policy and security requirements based on previously entered policies. For example the logistics provider requires that all location data of his nodes must be encrypted, while customs would satisfy for authentication only. This negotiation ends when the harbour customs and logistics provider agree on the component deployment and security context (location is encrypted, beacons authenticated). Once agreement is made, the harbour customs receives a token from the logistics provider which he can use to contact the secure deployment module and deploys the localisation component on the embedded node.

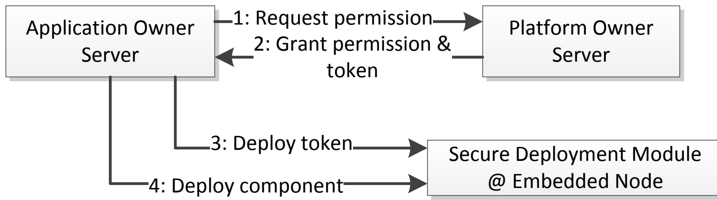


Figure 4.11: Component deployment data flow.

4.6.5 Application Enactment

The harbour customs sends the security token to the container node's secure deployment module, and deploys his code binary as shown on Figure 4.11. The node verifies that the deployment request is authorised by checking the deployment token. After the node received the component, the node verifies that the component is authentic and authorised. Once the component is verified, the node install the component in its runtime environment. Next the customs officer is installed as a user of the platform, so he can manage his application component. The harbour customs then sets up the necessary configurations, such as the current beacons, the subscription for the node location messages and security channel information: key material to authenticate beacon location messages and to encrypt the node location messages as shown on Figure 4.12. Once this is done, the harbour customs starts the application. To enact all these management operations, the harbour customs contacts the node using his credentials and manages the node using reconfiguration requests. All these requests are encrypted, authenticated and authorised.

4.6.6 Application Runtime

The final phase is the Application Runtime phase. During this phase the distributed application components communicate securely with each other. The system monitors the components' resource usage, and the users' platform usage, and if necessary the harbour customs can reconfigure the components to react to changing requirements.

Application communication Once the component is running, it will receive messages from beacons, and send messages to the harbour customs. Each beacon message is authenticated by the application communication interceptor when coming from the secure network layer as shown on Figure 4.13. The interceptor drops all non-valid beacon messages. After interception the beacon message is dispatched to the application component. This component then

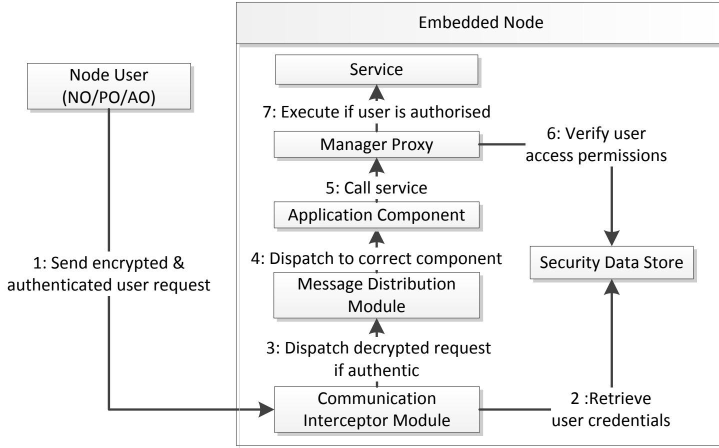


Figure 4.12: Service management data flow.

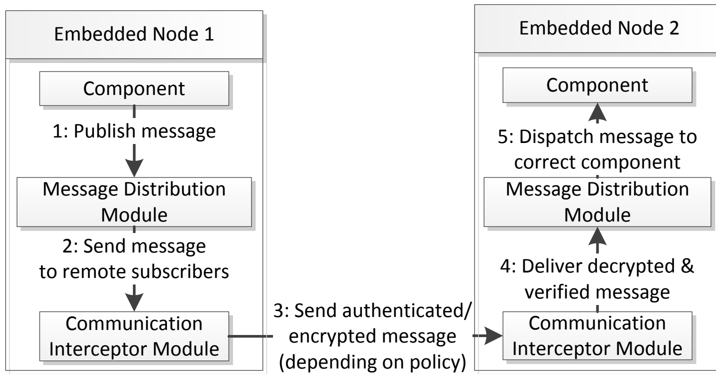


Figure 4.13: Application communication data flow.

periodically calculates its location, and sends out the node's location using a node location message. This message is then sent to the harbour customs. The application communication interceptor again intercepts this message and encrypts it, enforcing the outgoing application security policy.

Application monitoring The monitoring frameworks monitors and limits the memory and processor usage of the component, based on deployed resource limit policies, as agreed upon in the contract. The monitor component tracks node service usage by using the service interception proxies as shown on Figure 4.14 and ensures that the allowed limits are not surpassed.

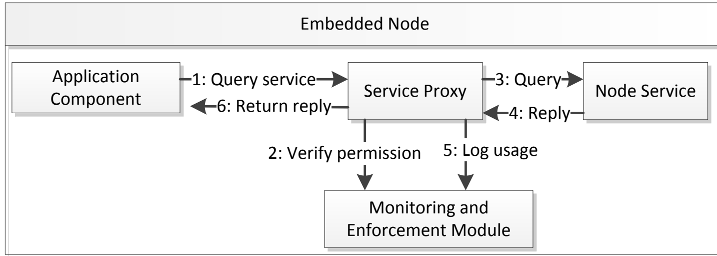


Figure 4.14: Service usage data flow.

4.6.7 Application Removal

Once the container leaves the harbour, the logistics provider or customs removes the component and all associated policies. This frees up the used resources for other future applications. All removal and clean-up commands are again authenticated and authorised.

Once all components have been cleaned up, and final node usage data has been acquired, the logistics provider can potentially charge the harbour customs for the node usage. However, since the harbour customs can offer the localisation service to the logistics provider, the logistics provider might not require any payment.

4.7 Summary

To summarize, this section presented the architecture of the SecLooCI secure embedded network framework. It first listed the responsibilities of the five major entities in the architecture:

1. **the End-user Tool** provides users with an environment where they can express their application and security requirements.
2. **the Application Owner Server** stores the application and deployment specifications, monitors the applications, and collects produced application data.
3. **the Platform Owner Server** stores and enforces platform owner policies and current node usage, and provides a node information service for application owners and network setup service for network owners.
4. **the Network Gateway** stores and enforces network owner policies, provides a network setup service for new nodes, and a node repository for the application owners.

5. **the Embedded Node** executes the requested application behaviour, in a secure fashion.

Next this chapter proposed a set of middleware modules that ensure the security of the lifecycle of the embedded applications. These modules are:

1. **the secure network module** securely adds the nodes to new networks and provides a secure network layer.
2. **the communication interception module** ensures that all messages that need to be secured, be it encrypted or authenticated, are done so based on the installed policies.
3. **the service interception layer** intercepts all component requests to either management or system services, ensures that the application or user is authorised to do so, and queries the monitoring and enforcement module to verify usage limitations.
4. **the security data store** stores all necessary security policies in a compact binary format, in addition to the current user and application monitoring data.
5. **the secure deployment module** enables users to securely deploy new code components onto the node.
6. **the monitoring and enforcement module** gathers and aggregates all monitoring information, and based on stored policies and logged usage, decides whether requested usage is permitted.
7. **the security management module** enables administrators to view and manage current security policies, user policies, and current usage.

Finally this section presented an example scenario showing how the different parts of the global architecture, and the node software architecture interact with each other during the previously presented embedded application lifecycle.

Chapter 5

Security Management Abstractions

This chapter details the security management abstractions. These abstractions are used by the different roles to express their policies and requirement. This chapter first looks at which abstractions are necessary for each role, in order to create a secure environment. Then it provides an evaluation of these abstractions. This chapter concludes with a discussion of the abstractions, to show that these abstractions meet the requirements of all roles.

These security management abstractions are data abstractions which specify the minimal set of necessary information the different roles must specify in order for the security framework to operate, and thus provide a simplified interface to the users. As such, they can be considered a data model of the information required to be expressed by the different roles. These abstractions represent just the set of data that must be expressed, typically strings or integer values, and can be encoded in any machine readable policy or data modelling language such as XML or JSON. This contribution has been published at the 1st ACM Workshop on Middleware for Context-Aware Applications in the IoT [72].

5.1 Platform Owner Model

The platform owner is the party that owns the nodes. As identified previously, the platform owner must specify which application owners and network owners he trusts, which resources he shares with which application owners, and at what

cost. This section lists the data model in which the platform owner can express the necessary policies and information in order to support the requirements. These abstraction are created and edited using the platform owner module of the end-user tool. Once complete, the platform owner sends them to his platform owner server, where they are stored, processed, and enforced whenever necessary.

The model consists of three parts: 1) the node model which describes the capabilities, limitations, and current state of the different nodes owned by the platform owner, 2) the application owner model which describes the permissions and policies regarding different application owners, their current deployed applications and their node usage, and 3) the network owner model which describes the trusted network owners.

The first part of the platform owner model is the node model. This model describes both the static and dynamic aspects of the nodes, as shown on Figure 5.1. The static information of the model contains an identification for each node, which can be a unique ID and/or MAC. Second it specifies the node type and which resources are available on the node. The type of node indicates which runtime environment the node supports. This could be for example an AVR node running ContikiOS, or a Tmote Sky node running TinyOS, or an OSGi node. This is necessary, since code often needs to be compiled for a specific platform. Resources represent the infrastructure of the embedded node, which is available to the embedded applications. Typical examples are file system, processing, networking, and sensors. Each of these resources can have associated parameters. These are for example how much of the file system a component is allowed to use, and how frequent a component can read from or write to the file system. Each parameter can have unique limits, and can have a certain cost, depending on the billing model of the platform owner. The cost can be zero, to model a parameter which application owners can use for free within limits. Resources can also be annotated with additional meta-data, which allows extensions to be added.

The dynamic information of the model contains the current IP and location of the node, and the current resource usage of each node. The current IP and location of the node must be updated each time the node migrates. This can be done during the network initialisation data flow, when the network owner contacts the platform owner to set up the node trust relationship. The last part of the model is the current resource usage, which details for each node how much of the limited resources are currently being used. This must be updated when the application owner requests to deploy a new application during the application deployment, and when the application owner removes old applications.

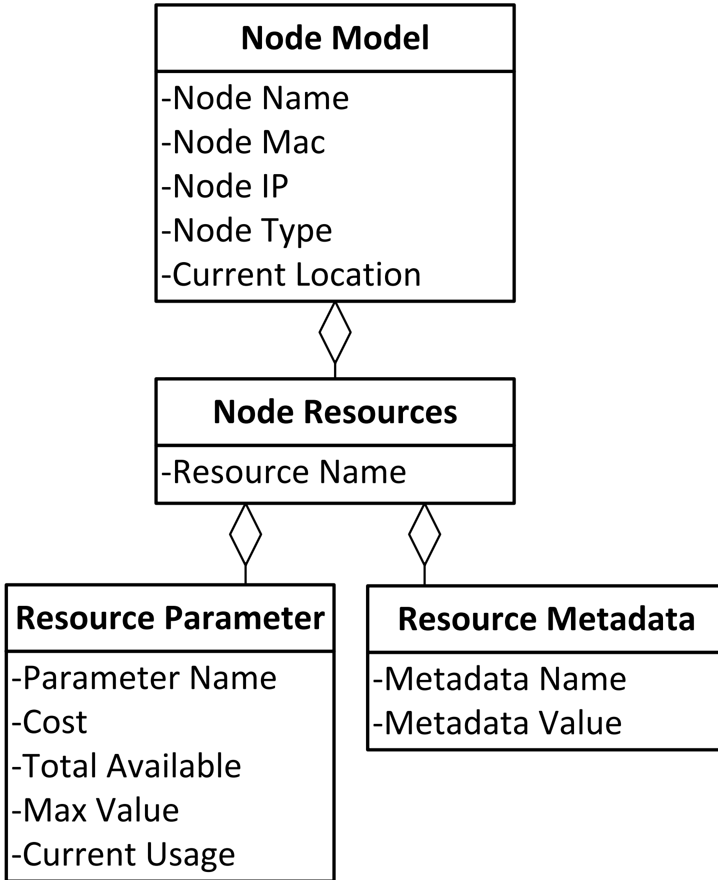


Figure 5.1: Platform owner abstraction: node model.

The second part of the model is the application owner model, as shown on Figure 5.2. This model describes which nodes and services are available for the different application owners, and stores the current deployments, together with the history of the resource usage. Platform owners can assign nodes to users on a per node, per type of node, per location or per node group basis. Additionally this can be assigned to groups of application owners to reduce the overhead on assigning permissions. Each application owner's current deployments are also stored. This includes which components are deployed or requested on which nodes, and which resources are reserved for these components.

To simplify the application owner model, we propose user profiles. A user profile is a default collection of limits, permissions and costs. These can be

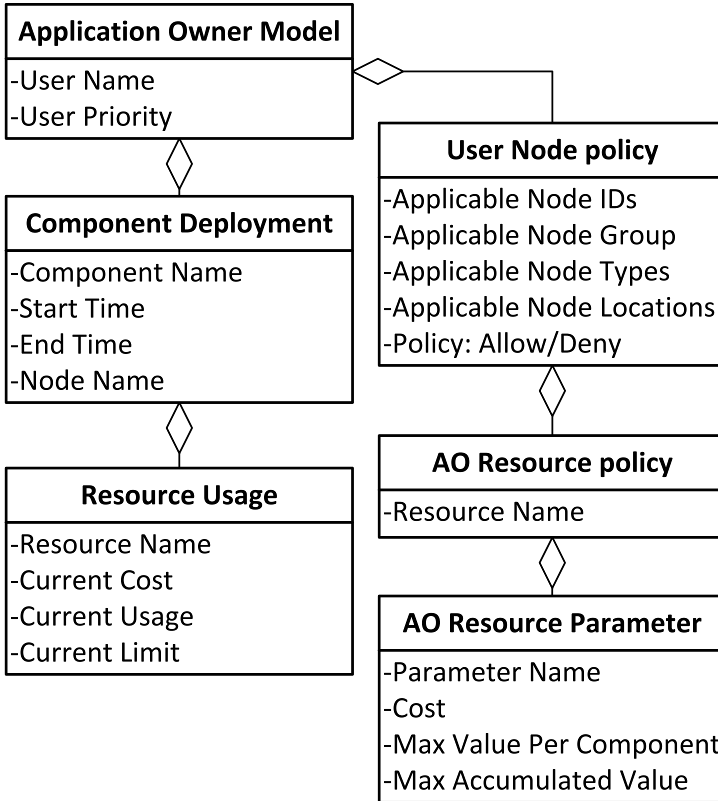


Figure 5.2: Platform owner abstraction: application owner model.

specified once, and then assigned to multiple different application owners. We also propose the following default profiles: 1) an admin profile which has access to all sensor and actuation services without limits, 2) an actuator usage profile, which has access to sensor and actuators, with predefined limits, and 3) a sensor usage profile, which has limited access to all sensors, but not actuators.

The last part of the platform owner model are the network owner abstractions. These abstractions allow the platform owner to specify the default policies regarding interacting with network owners, and list all network owners which are trusted enough so the platform owner is willing use their network, as shown on Figure 5.3. Policies can be added which specify the permissions of network owners to use certain additional node resources. The platform owner can also specify that he by default trusts all potential network owners, except those specifically blacklisted. Finally the network owner specification must also

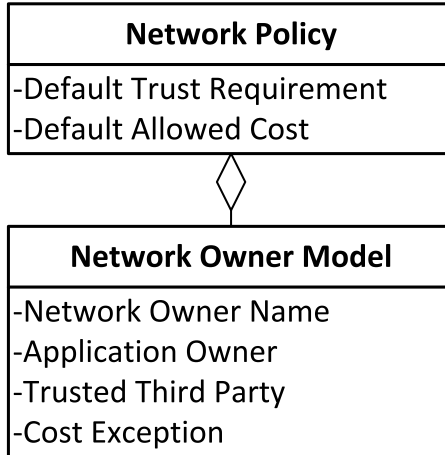


Figure 5.3: Platform owner abstraction: network owner model.

express what the maximum cost is to use the network.

5.2 Application Owner Model

The application owner model specifies all necessary abstractions required for the application owner. The parts that the application owner must specify are 1) which platform owners are trusted, 2) the generic application compositions and 3) the instance specific application deployments.

The platform owner model expresses which platform owners are trusted enough to use in application deployments. This can again be specific platform owners, or the application owner can specify that it trusts certain groups of platform owners, or trusts all platform owners which are trusted by a third party trust provider. Second, the application owner must specify generic application requirements. These requirements are for example different security requirements for different event types.

Applications are the functionality that gets deployed on the embedded node and performs the actual desired behaviour. This thesis wants to establish reusable applications which can run on multiple heterogeneous platforms and be easily annotated with security requirements. As such, this thesis proposes to extend the Service Component Architecture [16]. The extended model is comprised of three parts, as shown on Figure 5.4: 1) the component specification which

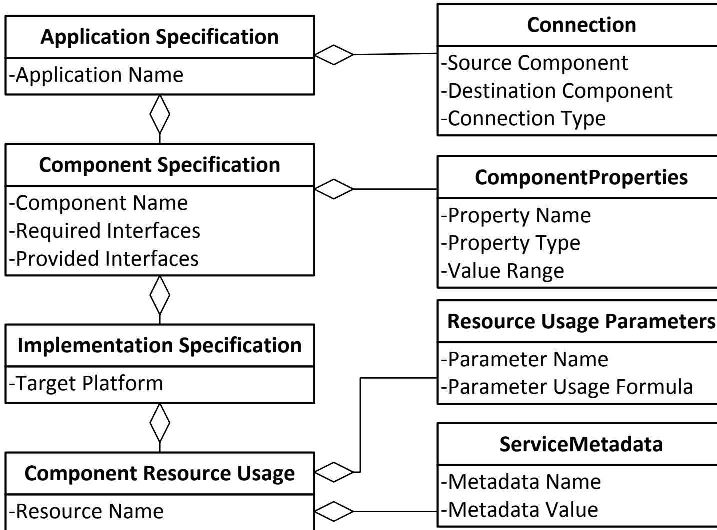


Figure 5.4: Application owner abstraction: application specification model.

specifies generic components, and as extension to SCA provides the necessary implementations for specific platforms, 2) the application specification, which groups the different components, and specifies how the different components are connected, and 3) the deployment specification, which specifies which components are deployed on which platforms and specifies security requirements, which is also an addition to SCA.

The component specification specifies a generic component interface. The generic component interface expresses: 1) the required and provided interfaces of the components, specifying which events it will send, and which events it can receive, and 2) the properties of the component, which are the values which allow modification of the behaviour of the component.

In order to deploy a component on an actual platform, it needs to have a code implementation. This is specified by the implementation specification. The implementation specification links to code binary that can be deployed on the embedded node. Further the implementation specification details on which platform the code can be deployed, and which resources are required. The implementation specification expresses which resources are used, since the required resources can be very dependent on the platform on which the implementation can be deployed. The required resources can be parametrised based on the parameters stated in the component specification. A single generic component can potentially have multiple implementations for the same platform.

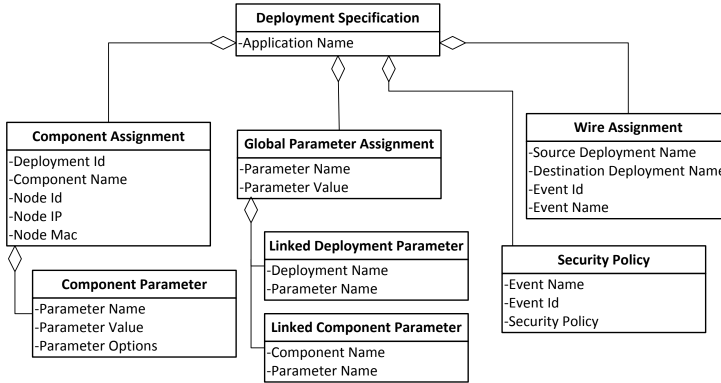


Figure 5.5: Application owner abstraction: deployment specification model.

During deployment, the deployment framework can optimise which component implementation to deploy based on the available resources.

The application specification declares the bindings between the different component interfaces. Non functional concerns can be added. The application specification can also declare global properties, which links multiple component properties. When such an application property is set, this application property is set automatically on all linked components.

Lastly, the deployment specification details the actual deployment. It assigns different components to different nodes, and can annotate each deployment with the assignment specific parameters, as shown on Figure 5.5. The application owner can also specify application properties if available, or specify his own global properties. Second it can optionally detail which components are connected, based on the bindings specified in the application specification. Lastly the deployment specification contains the deployment security specification. This deployment security specification details which events should be secured with which level of security. More specifically, for each event type or binding, the specification must list whether to send it using no security, authenticity verified or encrypted. The security policies must match the event security policies of the different platform owners which are part of the deployment, and the generic event security policies of the application owner.

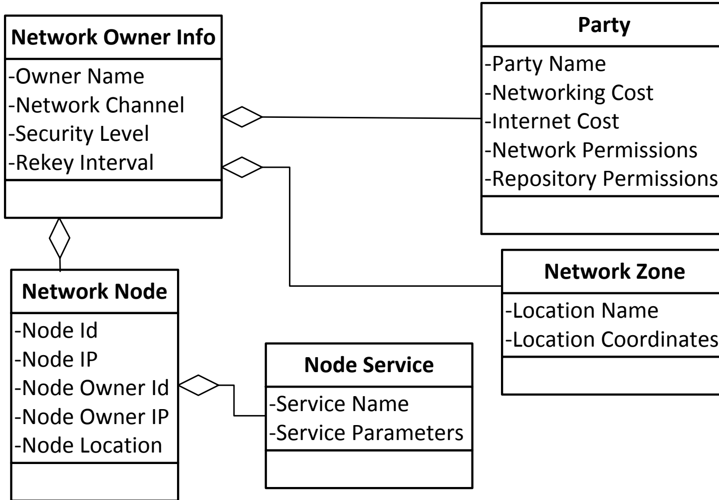


Figure 5.6: Network owner abstraction: network model.

5.3 Network Owner Model

The network owner model 1) specifies network policies and data, 2) models the current nodes in the network, 3) details which parties are allowed to use the network, and 4) which parties are allowed to use the repository. Figure 5.6 shows an overview of the network owner abstractions.

The network policies specify generic policies with regards to the network. This can be general security requirements such as the required security for the networking layer, and the re-keying interval. The network owner can also specify the different zones and locations present at the network owner's location.

The network repository contains a list of all nodes which are currently in the network. Additionally for each node, the current location can be stored, together with the available resources which can be used by application owners, and the owner of the node. This enables application owners to easily query the network owner for nodes suitable for their deployment.

Lastly the network owner must specify which platform owners are trusted to use the network, and which application owners are trusted to use the repository. The network owner can add additional policies regarding to platform owners specifying the cost to use the network. The network owner must be able to specify platform owner specific rates if necessary. The network owners can also specify access rules for the repository to certain application owners, for example

just allow it to query nodes of certain platform owners, or in certain zones.

The network owner can delegate the trust management to third party trust providers. The network owner would then trust a trust provider, who in his turn trusts a number of parties. The network owner would then trust these parties to use his services. Note that the network owner can potentially also act as a trust provider between application owners who want to use the services of nodes in the network owner's network, and the platform owners who own the nodes in the network.

5.4 Evaluation and Discussion

This section evaluates the proposed models. To evaluate the models, we propose a typical smart logistics scenario where a logistics provider wants to ensure the cold chain by monitoring average temperature of refrigerated containers. First this section details the smart logistics scenario. Next this section looks at the user effort required to model a simple temperature monitoring application. Third it looks at the verbosity of the models. Then, the section looks at the time it takes to validate deployments. Finally this section discusses how the architecture meets the requirements.

5.4.1 Logistics Scenario

To evaluate the model, we use a logistics scenario, which was developed in the context of the COMACOD industrial research project [114] and which builds on top of our previously discussed smart logistics use case (see Section 2.1.2). In this scenario, a logistic provider wants to monitor the temperature of his refrigerated container, while it is being transported by the transport provider, for example by truck. To do this, the logistics provider instruments his refrigerated containers with resource constrained embedded nodes, such as an AVR Zigduino node, running the Contiki embedded operating system [31] and the LooCI component middleware [51]. For more details on the choice of this platform, we refer to Section 6.1.1. The container is linked to a truck, with an on-board-unit which runs LooCI on OSGi [84] on top of the Linux OS, such as for example the one shown on Figure 5.7. This on-board-unit receives the temperature measurements, aggregates them, and periodically relays these readings over an Internet connection to the data sink of the logistics provider.

We use LooCI because it offers runtime deployment and easy adaptability, which is necessary to enable these complex dynamic use cases. The underlying LooCI

Figure 5.7: Example of an on-board-unit installed in trucks.

platform should run the node security middleware as presented in the next chapter, and the detailed overhead of the security middleware is discussed in Section 6.7. Most of the model however does not necessarily depend on the underlying platform, and as such can operate independently.

The application thus consists of three components: 1) a temperature sensing component that runs on the embedded Zigduino node, 2) a temperature aggregation component that runs on the on-board-unit running OSGi on top of Linux, and 3) a sensor sink, which we assume also runs an OSGi platform for simplicity. However, during periods that the container is not connected to a truck, the aggregator component must run on an embedded node, which potentially is equipped with its own Internet connection.

5.4.2 User Effort

As first evaluation, we quantify how much these abstractions aid users in managing their constrained embedded network. To do so, we calculate the number of actions a user has to take to deploy, change, secure and remove the previously described temperature application. We first implement the scenario using the LooCI management console which is currently the default way to

Table 5.1: User effort to enact the system. Case one is a limited deployment of 3 assignments, case two is a larger deployment with 201 assignments.

	LooCI	Tool	Reduction
NrAssignments 1	6	6	0%
NrActions enact 1	28	10	64%
Time to enact 1 (s)	160	45	71.9%
NrAssignments 2	501	501	0%
NrActions enact 2	2403	505	79.6%
Time to enact 2 (s) (est)	14000	4000	71.4%

interact with nodes running the LooCI component middleware. We compare these readings to the work needed when using the end-user tool. Results are shown in table 5.1. We evaluate two cases: case one is a deployment of 4 temperature sensor components, an aggregator component, and a data sink component. Case two is a scaled up version of case one with 400 sensors, 100 aggregators, and 1 data sinks, which represents the scenario presented, but then for an entire fleet of refrigerated containers. We deployed this onto the AVR Zigduino sensor nodes, OSGi on-board-unit and back-end server.

Using the LooCI console, enacting case one requires 6 deploy, 6 instantiate, 6 activate, and 10 connect actions, totalling 28 actions, which takes a user ca. 160 s. These actions are command line, so the user must ensure the correctness of each command. To enact this case using the tool requires: import application info, retrieve network info, 6 assign actions, one verification, and one deploy action. This results in 10 actions (64%) and takes ca 45 s. To enact case two using the console requires 501 deploy, 501 instantiate, 501 activate, and 900 wire actions, totalling 2403 actions, taking an estimated 14 000 seconds. To enact case two using the manager requires again two import actions, two actions to verify and deploy, and 501 assignment actions, totalling at 505 actions, which is only 20% of the default LooCI case. This takes an estimate of 4000 seconds.

We identify three change scenarios: 1) change a property of an existing deployment, for example because new regulations require higher sampling rate, 2) relocate the sink, for example because of a server upgrade, and 3) remove the system, for example to upgrade all applications. Results are shown in table 5.2. For classic LooCI, each property that has to be changed, requires one command. To set a property using the manager requires 2 actions (change property and commit). To change the sampling property of all 400 sensors requires 400 actions in the console case, but only 2 actions in the tool. The second evolution is to move a component from one node to another. To relocate the sink in the simple case requires 6 commands in the console, and only 2 actions in the tool. To relocate the sink in the second case requires 303 commands using the

Table 5.2: User effort to change the system.

	LooCI	Tool	Reduction
NrActions change property 1	6	2	66.7%
NrActions change assignment 1	6	2	66.7%
NrActions secure 1	20	4	80%
NrActions remove 1	6	1	83.3%
NrActions change property 2	400	2	99.5%
NrActions change assignment 2	303	2	99.3%
NrActions secure 2	2500	4	99.8%
NrActions remove 2	201	1	99.5%

console, while still only 2 commands in the tool.

To secure a deployment using the console, the user must manually add the necessary encryption and decryption policies and key material on each relevant node. In the console, the user has to manually enter the key, which is hard to repeat for the user. The exact commands the user has to perform is to add an encryption channel and an outgoing encryption policy on the outgoing node, and add a channel and an incoming encryption policy on the receiving node. This has to be repeated for each binding between components. This results in 20 actions in the first small case, and 2500 actions in the second larger case. Using the management abstractions, the user just has to select the preferred level of security for each data type. Since there are 2 types of bindings present in the example, the user just selects for each event the correct level of security, which results in 4 actions.

To remove a deployment using the console, the user must manually remove each component, requiring 6 actions in the first case, and 501 actions in the second. The owner also does not necessarily know who has deployed which component, and which components are currently in use, causing potential issues. The tool maintains a list of assignments which enables traceability, and removing the entire deployment in a single action.

5.4.3 Modeling Language Evaluation

This section evaluates the size of the abstractions of the temperature application, as presented in Section 5.1. Evaluated encodings are: component description, application description (Fig. 5.4), and owner description (Fig. 5.2). The encodings are evaluated based on the size. While this evaluation is not an ideal evaluation criteria, it does allow for some simple comparisons, and indicates

Table 5.3: Overview of the sizes of different descriptors.

	Size
Implementation description Contiki	1 332 B
Implementation description Java	2 327 B
Component description	7.0 kB
Application description	10.4 kB
Deployment description (201 assign)	116.7 kB
Security description (2 event types)	301 B
Network description 3500 nodes	22.5 MB

how much is needed to store and transmit the different descriptors. Table 5.3 gives an overview of the sizes of the different descriptors.

A component description for the aggregator component is comprised of: 2 implementations (OSGi + Contiki), 1 interface, 1 property, and 4 resources (ROM, CPU, RAM, sensor) requires 6 952 B. Each implementation increases this with 2 000 to 3 000 B, depending on binary size. An application description for a temperature monitoring applications consists of 3 components: a temperature sensor, aggregator, and storage component. Both the sensor and aggregator have two implementations. The total application requires 10 412 B.

For an evaluation of the deployment description, we modelled the previous application of 201 component assignments, each with 1 property assignment, and 2 global application assignments. This produced a deployment description of 116,7 kB. The associated security description which specified the security for the temperature and the aggregated temperature data flow, was 301 B.

Discussions with H.Essers, a logistics service provider in Flanders, provided a realistic application context. All these nodes have limited processing, memory and energy resources. The total node list, containing 3 500 nodes (1 200 trucks and 2 300 trailers) with on average 7 resources (file service, data reception, production, processing, and 3 actuators / sensors), has a size of 22.5 MB.

This evaluation shows that scaling is manageable even for a large number of nodes and different implementations.

5.4.4 Verification Evaluation

This section briefly evaluates the time required to verify deployments on a Dell Latitude E6500 (Intel Core2 P9500 2.53 GHz, 4 GB RAM). The verification of 3 assignments takes on average 0.6 ms, and of 1000 assignments, 252.15

ms. Since the time required to verify is about equal to timePerAssignment × numberAssignment × avgNrResourcesPerNode × avgNrParamsPerResource, the time to verify scales linearly with the number of assignments.

5.4.5 Discussion

This section discusses how the requirements introduced in Section 2.4.1 are met based on the proposed abstractions, user tool and the presented evaluation.

Platform owner requirements: The platform owner can express his embedded node limitations and usage restrictions using the platform owner model (5.1). This is needed for example in the scenario, where the logistics provider needs to be able to limit the capabilities and available resources of both the transport provider, and cargo owner. These limitations and restrictions are then deployed as binary policies onto the embedded node. The SecLooCI middleware then enforces these policies on a node local level. The next chapter will show that the necessary security middleware can be deployed even in very resource constrained embedded networks.

Reusable application The application modelling language enables developers to create reusable applications, through the assembly of component functionality, as shown in Section 5.2. Very often, application owners will need to deploy the same application on multiple different systems. Take for example the logistics provider who wants to deploy the temperature tracking application on all his nodes. The application owner models enable the application owners to instantiate these applications repeatedly onto an embedded network, and parametrize the system as needed. The size of these applications is small enough, so they are fast to exchange between users or store in application repositories as seen in Section 5.4.3.

Scalable By distributing data across all parties, the solution becomes quite scalable, and prevents a single point of failure. For example in the smart logistics scenario, there will potentially be millions of containers producing data. By inherently distributing the architecture over the different parties, the solutions becomes inherently more scalable. Additionally, if for some reason one party or system fails, the other systems can continue to operate. If all information was centralised, a fault in that server would effectively make the embedded network unusable. Currently, if one platform owner fails, the nodes of other platform owners remain available. If one application owner fails, the applications can still keep on running, and the applications of other application owners can continue to operate and be monitored.

User assistance The tool can assist the during the entire application lifecycle.

During the policy declaration phase, the tool allows users to easily encode their policies. During the application instantiation phase, the tool allows the user to select the preferred node from a list of only possible nodes. The tool further offers a list of possible values for the user to select from wherever possible, so the user can only select valid items. If impossible to offer a list, the user just has to fill in the string or number value in a textbox. A user will never have to write an entire XML specification himself, but the tool generates this based on user input. The tool evaluation (5.4.2) shows that for sizable embedded networks, the effort for deploying, securing, changing and removing applications is reduced significantly. The tool also validates the assignments, ensuring that the user does not use too many resources of any node or platform owner, and calculates estimated total cost.

Previously it was often up to the user of the network to keep track of all installed hardware, while now the system can keep track. Ideally this approach can be extended to a system where the user only needs to declare his top level monitoring requirements, which are then automatically mapped to a suitable deployment. The current node models and application models provide parts of the necessary information to perform such mapping, but we currently consider this out of scope of this thesis.

Node Type Transparency By separating implementation from component description, users assign functionality to nodes, without having to manage implementations. Component developers can provide multiple implementations for the same platform, so the tool can decide which implementation to deploy based on available resources. Research on when to make such trade-offs is future work. The scenario for example shows that the aggregator might need to run on the on-board-unit of the transport provider, or on a node in a container, depending on the current state of the network. These platforms are not necessarily the same. However the application owner wants this to be handled transparently.

Reliability of Application By specifying the target deployment, the system can autonomously verify that the application is in the desired state, increasing reliability of the application. The system can verify periodically, or can monitor the output of the sensing application, ensuring that all components send data with their assigned sampling frequency. The prototype shows that when a node resets, the system is restored in the correct state after the verification period expires. This can include redeploying code or resetting properties. However, the more frequently you verify your application, the more communication overhead you create. In embedded systems, it can occasionally happen that nodes reset, either by environmental conditions, or bugs in the software. Being able to quickly return to an operational state with minimal user effort, greatly reduces the burden on the application owner.

Evolvability of Application Real world deployments often see periodic changes in the requirements or the context of an application deployment. For example in a logistics scenario, the truck carrying the container will change fairly often. Additionally, depending on the cargo, the logistics provider might need to change the sampling rate, or temperature alarm boundaries, which also happens fairly frequently. By having a declarative model associated with the additional security and functional requirements, the system can be quickly and securely adapted to meet these changing requirements.

5.5 Summary

This chapter detailed the security and application management abstractions. These abstractions are used by the different roles to express their policies and requirement:

- The platform owner abstractions allow the platform owners to express: 1) the resources and state of his embedded nodes, 2) the policies and current usage of the application owners, and 3) the trusted network owners, and network policies.
- The application owner abstraction allow the application owners to express: 1) their generic components, 2) the implementations which realise the components on the nodes, 3) the applications which consists of multiple components connected together, 4) the deployments which specify which component is deployed on which node, with which parameters and security requirements, and 5) the general security and trust policies of the application owner.
- The network owner abstractions allow the network owners to express: 1) the policies and cost with regards to using the network, 2) the nodes currently in their network, 3) the trusted platform owners, and 4) the trusted application owners.

Finally, this chapter provided an evaluation of the provided abstractions, and concluded with a discussion of the abstractions, to show that these abstractions meet the requirements of all roles.

Chapter 6

Security Middleware

This chapter describes the implementation of the different systems of the SecLooCI node security framework that implements the different architectural modules as discussed in the previous chapters. First this chapter looks at the underlying embedded node platform on which the different systems are implemented, and lists the assumptions which were made during the design and implementation of the different prototype systems. Next this chapter presents an implementation for a secure subsystem running on these embedded nodes for each of the five data flows. For each subsystem, the implementation is evaluated in terms of memory overhead, processing overhead and communication overhead. The node framework architecture and validation have been published in the Ad Hoc Networks journal [70].

6.1 Underlying Platform and Assumptions

In order to create the framework, we made certain assumptions. Firstly, the framework assumes that embedded nodes always start in the care of the platform owner. When the platform owner has the embedded nodes, he can securely deploy initial key material. These keys are used later to establish confidentiality and authenticity for different kinds of communication, so it is crucial that this deployment happens securely. This initial key material consists of one or more long term secret symmetric keys.

We made the choice of only using symmetric key encryption in the development of the security middleware. This significantly reduces the amount of memory and

processing needed for protocols and key material compared to asymmetric key cryptography. This will reduce the overhead of the different systems, to ensure that the produced systems are as light weight as possible. More capable nodes such as embedded PC's or smart phones can support asymmetric key encryption. This enables adaptations of the proposed protocols and implementations, with different security guaranties. However, the goal of this thesis is to create an infrastructure that enables sharing of resource constrained embedded nodes, hence we chose the protocols with the lowest overhead.

The nodes use TCP [91] and UDP [90] as networking protocols on top of IPv6 6LoWPAN based wireless sensor networks, with a 802.15.4 MAC [79] layer. We chose this setup because first it allows clients, servers and nodes to easily communicate with each other across the globe, without having to be aware of the specific embedded network settings. Second, most programmers, developers and network administrators have decent knowledge of these standards. By using standard systems, these parties can leverage on existing knowledge to tackle any potential problems. Further we assume that nodes receive a global IPv6 address from network owners, potentially using the IPv6 Stateless Address Autoconfiguration [111]. This then allows any party to communicate with the embedded nodes. This network setup is not necessary for the infrastructure to operate: other protocols can be used, yet these protocols would require seamless integration with the Internet and must provide easy global connectivity.

Lastly the system provides secure node operation, communication and management. Current node systems offer little to no memory protection. Thus, when code is deployed onto these embedded nodes, it likely will have access to all memory of the embedded node. This thesis assumes that either countermeasures are present on the embedded node to prevent code from performing malicious actions, or the code is validated pre-deployment. These countermeasures can be present either in hardware or software. The hardware countermeasures would prevent over the air components from accessing prohibited memory, while software countermeasure would be for example code verification or code modification techniques. The code verification can be done during the planning phase: when the application owner requests permission to deploy certain code components, the platform owner can request that the application owner presents the binary and associated code. This code can then be manually or automatically verified to ensure platform integrity.

6.1.1 Underlying Platform

The SecLooCI framework secures the LooCI middleware [51], which operates on top of the Contiki OS [31]. LooCI is a component-based middleware comprised

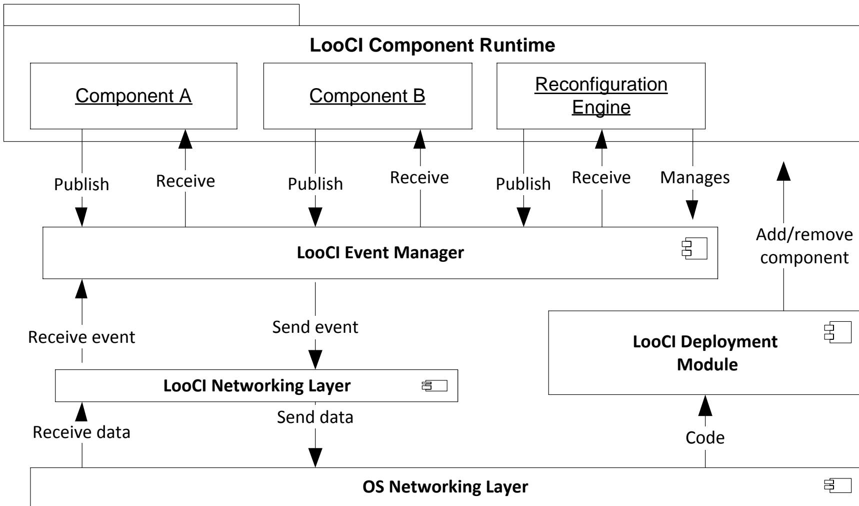


Figure 6.1: Overview of the LooCI component middleware.

of an execution environment, a component model and an event-based binding model. The LooCI middleware currently supports the following platforms: Contiki on AVR Ravens and Zigduino's, SunSPOT sensor nodes, Android smart phones, and OSGi java environments. The LooCI middleware is comprised of the following parts as shown on Figure 6.1: 1) The LooCI networking layer, 2) the LooCI event manager, 3) the LooCI component runtime, 4) the LooCI management component, and 5) the LooCI code deployment module.

LooCI uses events as the sole mechanism to communicate between components. A LooCI event is comprised of: (1) sender address and source component as part of the UDP header, (2) extension headers, (3) event type, and (4) event payload. LooCI components are code binaries, which are deployed over the air at runtime using the LooCI code deployment module. The binaries are then instantiated into running components. Components are only allowed to interact with other components through the usage of LooCI events flowing over the distributed LooCI event bus. The components can communicate either locally with other components on the same node, or remotely with other components across the network. Each component can subscribe to typed events or publish typed events himself. Components themselves are oblivious however to the exact senders or receivers of those events. This loose coupling promotes component reuse in different distributed contexts, since there are no hard wired connections between components.

The event manager implements the distributed event bus. It keeps wiring tables,

Table 6.1: LooCI and Contiki memory usage.

	ROM (B)	%	RAM (B)	%
Contiki operating system	42 688	32.6%	9 712	59.3%
LooCI middleware	24 942	19.0%	2 644	16.1%
Total unsecured platform	69 630	51.6%	12 356	75.4%
Available on AVR Zigduino	131 072	100.0%	16 384	100.0%

dictating which events of which types from which components have to be sent where, either locally to another component or remotely to another node. If it is sent across the network, the event manager transfers the event to the LooCI networking layer. The networking layer translates outgoing events to UDP messages, and incoming UDP messages to LooCI events. Reconfiguration and inspection of the event manager is done by the LooCI reconfiguration manager. All reconfiguration and introspection of the LooCI middleware is done using LooCI management events with the exception of deployment of new applications, which uses an optimised deployment protocol.

For prototyping we use the Zigduino microcontroller platform. This platform is arduino-compatible and is equipped with an Atmega128RFA1 [5] 8-bit microcontroller running at 16 Mhz and 802.15.4 integrated radio. It has 128 KB of flash memory, 16 KB of RAM memory, and 4 KB of EEPROM. This device can be considered a prototypical current microcontroller. Additionally it is supported by both the LooCI component middleware, and the Contiki Operating System, making it a suitable choice of platform.

The overhead of Contiki OS and LooCI is small, yet significant for resource constrained environments. Contiki OS requires 42 688 B of ROM and 9 712 B of RAM memory to operate. This includes the uIP stack for networking and other libraries for memory and sensor management. The LooCI middleware requires another 24 942 B of ROM and 2 644 B of RAM on top of the Contiki OS. An overview of the overhead is shown in table 6.1.

For encryption support, the prototype uses the AVR crypto-lib [1], which offers software implementations of most popular encryption, authentication and integrity algorithms such as AES, SHA, and CMAC. The cryptographic algorithm overhead is listed with each protocol, but these algorithms are reused across multiple protocols. To get a full detailed overview of the overhead, please refer to table 6.7 at the end of this chapter.

The test setup used for each subsystem is a single hop network where the overhead is calculated for a single node. All overhead is measured on a prototype implementation running on a Zigduino sensor node. We chose this setup for

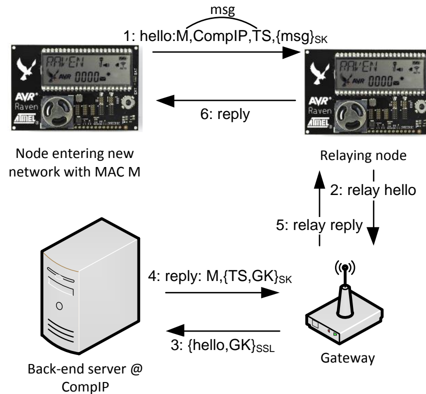


Figure 6.2: Secure network setup key exchange protocol.

the following reasons:

- Many current industrial, real world and research setups only use single hop networks with very little mesh network, because the ease of setup and maintenance.
- The evaluation setup allows for a clear and easily reproducible test setup.
- The goal of the security middleware is to enable sharing on a per node basis, with users interacting only with a single node at a time. The middleware does not contain any network wide operations. We thus believe the best evaluation is to evaluate the middleware overhead on a per node basis, instead of on a network-wide basis.
- There are currently no standard mesh network test setups. Any results from any arbitrary test setup thus would have very limited to no comparison value to other experiments. Additionally most large scale networking results are simulations, which would be hard to compare with a real world implementation.
- Some recent research has shown that single hop networking can be more energy efficient compared to multi-hop networking, and is definitely easier to set up and maintain.

6.2 Secure Network Initialisation

The secure network initialisation system MASY secures the network initialisation data flow on the node. On the node it is implemented by the secure network

module. Figure 6.2 shows the secure initialisation data flow in detail. This work has been published at the 2010 IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob 2010 [73]).

The basic protocol operates as follows: 1) when a node detects that it has entered a new sensor network, it sends out a hello message containing the identity of its platform owner in the form of an IPv4 address, the node's identifier (his MAC address) and the timestamp. The hello message is authenticated using the long term node secret key. This is sent as one message to reduce network overhead. 2) When the network owner receives this message, it looks up whether the platform owner is known and trusted. If so, it relays the hello message to the platform owner over a secure connection, with the notification that the node is now in the network of the network owner. 3) The platform owner validates the message authentication code based on the identity and timestamp fields. If the token is valid, the platform owner requests the network key from the network owner, encodes it in a token containing the identification and new timestamp, and sends it back to the network owner. 4) The relaying node then sends the token to the new node, which decrypts and validates the token.

This protocol ensures that only nodes that are owned by platform owners which the network owner trust, can join the network. This is enforced since if the network owner does not know or trust the platform owner, he can decide to not continue with the protocol, effectively locking the node out of the network. The protocol also ensures that nodes can only join networks of those network owners that the platform owner trusts. When the network owner contacts the platform owner, he must authenticate himself. If the platform owner then does not known or trust the network owner, the platform owner can decide not to create a join token, which effectively prevents the node from joining the network. While it is often desired that this trust exists between network owner and platform owner before joining, it is possible that both parties have a trust by default policy, so that even if they do not know each other before the node enters the network, they still allow the node to join the network.

The described process is optimised for resource constrained embedded devices, and as such a number of trade-offs have been made: 1) encoding the platform owner identity as an IPv4 address, 2) the identity of the node and platform owner is broadcast over the network, and 3) the current protocol does not allow the network owner to change the node settings. These issues can be mitigated by: 1) allow other owner identifiers, 2) add anonymization to the platform owner and node identifiers, and 3) instead of deploying the network key, allow the network owner to manage the embedded node. The protocol does not implement the proposed changes due to lack of time, and because most changes require significant additional infrastructure and resources, with little actual gain. The next paragraphs will detail the logic on why each trade-off was made, and more

details on potential fixes.

The first trade-off is the IPv4 owner identifier. For the network initialisation protocol to operate, the platform owner must have a unique identifier which enables the network owner to contact him. Currently the protocol uses the IPv4 address of the platform owner's back-end server, since it offers an easy, globally unique identifier which can be immediately used by the protocol, and is efficient to encode. The downside is that it is hard for companies to change IP addresses. Additionally the address of the server is fixed, limiting potential cloud migrations etc. Possible alternatives are using an IPv6 address, using a DNS system, or using a DNS alike central lookup repository, however each of these has advantages and drawbacks. The IPv6 address allows for larger address space, but suffers also from limited migration possibilities. The DNS approach allows for more dynamism, however probably requires larger identifiers, or requires the setup of a new alternate identifier ecosystem.

The second trade-off is the lack of privacy of the node and owner identification. The current protocol uses static node identification and owner identification, which are broadcast unencrypted. This allows passive listeners to potentially track nodes, and the actions of certain owners. To mitigate this, we propose two countermeasures. The first countermeasure is changing the node identifier (its MAC address) at any new network initialisation process. By using cryptographic key chains, the platform owner can still validate the node ID, without broadcasting a known ID. However, the network owner still needs to be able to identify the owner of the node. To mitigate this threat, we propose to use anonymization techniques for the platform owner by using a secure proxy. The platform owner ID would be an anonymised ID that a trusted third party can resolve to the actual owner. The trusted third party can negotiate between network owner and platform owner, potentially without disclosing the platform owner identity.

The third and last trade-off is the inability of the network owner to directly deploy and change network settings. Currently, the protocol deploys a network key onto the node, to enable access to the network with minimal setup. However, the network owner might need to deploy additional network settings, such as routing settings, or network channels. Under the current protocol, the network owner would have to go through the platform owner repeatedly, or has to be added as a user to the node. To enable immediate access to the node, the protocol could additionally deploy a network owner user to the network, including a user key and user limitations. This would add the network owner as a user of the node. The network owner could then access the network management services of the node, changing the network key, and other network settings. In the current implementation, we decided not to add this, since it would require additional overhead to set up the network.

Node MAC	Home IP addr	Timestamp	Signature
8B	4B	4B	16B

(a) MASY request message

Node MAC	Timestamp	Group key (Enc)	Signature
8B	4B	16B	16B

(b) MASY reply message

Figure 6.3: Packet format of the MASY Hello message. Signature is signed, and group key is encrypted with the node unique key, only known to the sensor node and Platform Owner.

This remainder of this section evaluates the communication, memory, and processing overhead of the secure network component. This section evaluates the setup of the secure key material, assuming other policy deployment actions can be done after the key material has been deployed. The results are compared to TinySec [58] as reference for an efficient WSN secure network layer. TinySec is a link layer security architecture designed specifically for resource constrained environments. It offers a secure network layer, but has no functionality for network initialisation. However, since almost all embedded systems security protocols operate under the assumption of pre-shared keys, we chose to compare to TinySec.

Communication overhead: Two types of messages are transmitted across the network: Hello messages, which are 32 bytes and Reply messages, which are 44 bytes. The exact format of these messages is shown in Figure 6.3. These figures ignore MAC/IP layer overhead. During communication, all messages can be encrypted, authenticated or integrity checked, depending on the network policy. The communication overhead for these cryptographic primitives is limited to the additional MAC. The current implementation uses an 8 B MAC to ensure message integrity. TinySec does not offer any clear registration functionality. TinySec also uses an 8 B MAC during normal communication, equalling the communication overhead during operations.

Memory overhead: The total ROM overhead of the component is 5 690 B and consists of: (a) the component implementation (946 B), and (b) overhead for AES encryption algorithm (4 746 B). The total RAM overhead of the secure network component is 218 B and consists of the encryption context and message buffers. TinySec has a ROM overhead of 7 148 B and RAM overhead of 728 B, which is comparable to the secure network component.

Processing overhead: The secure network component operates at hello packet creation, reply packet reception, and when a message is sent to or received from

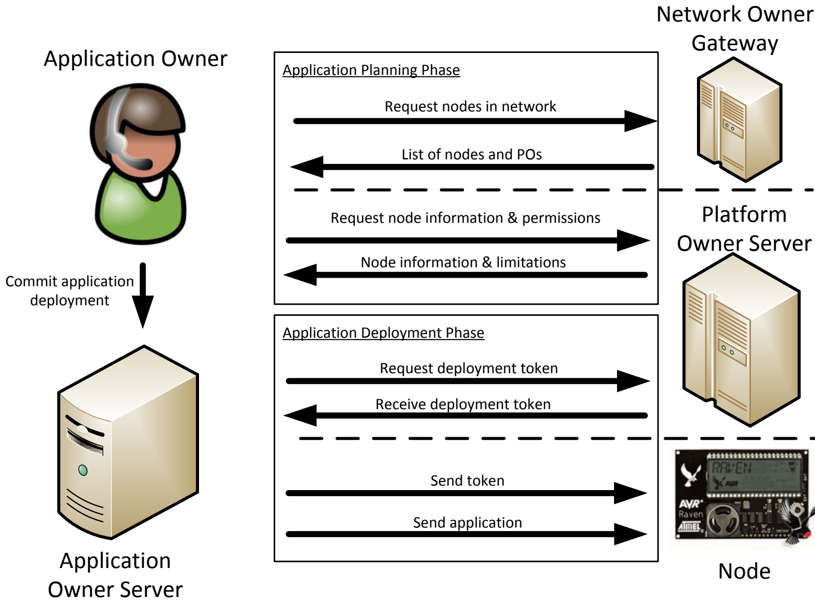


Figure 6.4: Overview of the Sasha Secure application deployment protocol.

the network. The creation of a hello packet requires the encryption of 16 B of data, requiring one AES operation. This takes a total time of approximately 1.6 ms using an AES software implementation. Receiving the network reply also only requires the decryption and authentication of 16 B of data, again taking approximately 1.6 ms. Encrypting and decrypting network packages also requires about 1.6 ms per 16 B block. To give a comparison, sending a plain UDP packet also requires ca 2 ms on the AVR Zigduino. TinySec uses two different block ciphers to process messages. Encrypting a 16 B message using RC5 requires 1.04 ms, while SkipJack requires 1.52 ms.

6.3 Secure Application Deployment

The secure deployment system secures the application deployment data flow as explained in Section 4.5.2. It listens to the network for user connections. When a user or server connects to it, he first must send a small token containing the application hash, the resource usage limits and the owner information. If the token is valid, the secure deployment component listens for the component code, installs it, and deploys the usage limitation policies listed in the token. If the token is invalid, the connection is closed. This section first details the

secure application deployment framework operation, and next it evaluates the communication, memory, and processing overhead of the secure deployment component and compares it to the Sluice secure deployment protocol [65]. This work has been published at the 2011 IEEE 8th International Conference on Mobile Adhoc and Sensor Systems (MASS 2011 [74]).

The secure application deployment framework operates in two phases: 1) the application planning phase, and 2) the application deployment phase. During the application planning phase, the application owner must retrieve information on the nodes he wants to use, plan his deployment, and commit it to his server. Next the application owner server commits the necessary resource usage to the platform owner server, retrieves the required tokens, and performs the actual code deployment, using the permission token obtained previously, as shown on Figure 6.4. The next paragraphs explain the two phases in more detail.

To start, the application owner must first retrieve information on the nodes he wants to deploy on. To do so, he can first query the network owner's node repository and learn which nodes are near his object of interest, together with which platform owners own said nodes. Once the application owner knows which nodes are available, he can contact all platform owners who have potentially interesting nodes, and request which nodes he is allowed to use, the node details, and the node resources and limitations. Once the application owner has all the node details, he can decide on which nodes he wants to deploy his distributed application. He assigns his application behaviour to the different nodes, validates it, and commits the application deployment plan to his application owner server. This then starts the application deployment phase.

At the start of the application deployment phase, the application owner server will commit the node usage of the components to be deployed to the relevant platform owners. It also requests the necessary tokens to deploy the application components, thus requesting permission to use the embedded node of the platform owner. This request contains the component specification, together with the max parameters with which he wants to use the component and the maximum amount of required resources to operate the component. The platform owner can then ensure the required resources are available, and provision them to ensure no resource conflicts occur. Optionally the platform owner can request the actual code binary of the component, and perform code validation. This would ensure that the deployed code likely would not exploit the embedded node. The platform owner then creates a token which encodes the permission to deploy the code.

The token is comprised of the hash of the code of the component, the length of the component, and limits on ROM, RAM, network and CPU usage. The hash ensures that the code deployed matches the code approved by the platform owner.

This is necessary to prevent code alterations by either the application owner or a potential networked attacker. The resource limits ensure that the component can only use a limited amount of network resources. The token timeout and tokenId ensure that tokens are not valid indefinably. Lastly, the component timeout allows the platform owner to specify that certain components are only allowed to be deployed for a fixed amount of time. The token can contain an optional MAC to ensure the integrity of the token, and a secret key to allow the application owner to encrypt the binary.

The token is sent back to the application owner server. Then the application owner server can either immediately use the token, or delay using it until the token times out. Allowing a long timeout allows the application owner server to deploy components in networks that are not immediately needed, for example as back-up to certain components, or to buffer the token for later usage, when the node is perhaps not connected to the Internet. For example in a large freighter, it might be necessary to deploy components while at sea. By having tokens ready, the shipper can deploy components at sea, without needing to communicate with a back-end.

When the application owner server then wants to deploy the component, the server first sends the token to the embedded node. The node receives the token, decrypts it and validates the tokenId and timestamp to ensure the token is valid. Currently no MAC is appended to the token, to reduce the net overhead. An additional 8-16 byte MAC can be added to ensure no token tampering and further ensure token validity. If the token is invalid, the node aborts the connection.

If the token is valid, the node continues listening to the channel. The application owner server then sends the component code, which can optionally be encrypted. If encrypted, the node decrypts the component by using the key which must be provided in the token. During the deployment, the node ensures that the server only sends the committed amount of bytes. When all the expected bytes have arrived, the node calculates a cryptographic hash. If the hash matches, the component is installed in the runtime, and the application owner server is notified of a successful deployment. If the hash does not match, or any other errors occur, the application owner server is notified of the issue at hand. The server can then either contact the application owner for further instructions, retry later, or try other back-up solutions.

When the code is installed, the correct resource restriction policies are deployed in the resource monitoring framework. The token is stored to ensure that tokens cannot be used multiple times. Finally, the secure deployment system stores whether the component has to be deleted at a certain time in the future, or when an other event occurs. These tokens are then checked every few hours to

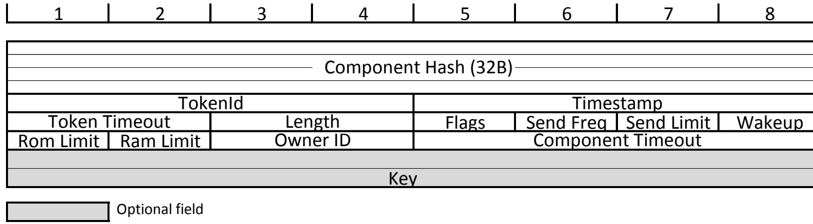


Figure 6.5: Overview of the secure deployment token. The key is optional and used if the application owner required confidential deployment. Token overhead is thus either minimally 56 or 72 bytes. The token is transmitted encrypted using AES128 and the long term node key. The token can be extended with additional binary resource restriction policies.

verify that all components are still valid.

The next paragraphs evaluate the secure deployment system on communication overhead, memory overhead, and processing overhead. These figures are then compared to the Sluice secure deployment protocol [65]. Sluice secures the TinyOS code deployment protocol deluge by adding a hash chain to the protocol, and signing the first packet with an ECC signature. This evaluation shows that the overhead of the SaSHA secure deployment system is comparable in overhead with regards to communication overhead, and performs better than Sluice with regards to memory and processing overhead due to using symmetric encryption as opposed to asymmetric encryption.

Communication overhead: The only communication overhead of the secure deployment system is the additional token, which is sent before the actual deployment. The total size of this token is 56 B to 78 B. Figure 6.5 shows the exact format of the token. The deployment secret key is optional. The token is encrypted and optionally authentication using AES128-CCM with an 8 B MAC, using the nodes long term secret key. This is the only mandatory network overhead. Since a typical component has a size of 1000 B to 2000 B, it entails a communication overhead of 2 to 5 percent. Sluice uses a similarly sized token of 44 B.

Memory overhead: The secure deployment component has a ROM overhead of 8.1 kB and a RAM overhead of 504 B. Table 6.2 contains a more detailed memory overhead analysis. To compare, Sluice has a ROM overhead of 9 kB and RAM overhead of 2000 B. The larger RAM overhead of Sluice is due to the larger buffer size.

Processing overhead: The delay to deployment by the added security consists

Table 6.2: Secure application deployment memory overhead.

	ROM (kB)		RAM(B)
AES crypto lib	4.2	Crypto buffer	240
SHA-2 crypto lib	1.8	Token and key buffer	184
Deployment component	2.3	Deployment buffer	80
Total ROM	8.1	Total RAM	504
Sluice ROM	9	Sluice RAM	2000

of the transmission and validation of the application token and validation of the component. Since most application components are more than thousand bytes large, the additional delay for sending a token remains limited to a few percent.

The time overhead of the computation on the node is estimated in the hundreds of milliseconds. Decrypting one 64 B block takes 8 ms, hashing 6.5 ms. A small component update of 1 kB would for example take 1024/64*(8+6.5) ms = 232 ms. Sluice uses the Elliptic Curve Digital Cryptography Signature algorithm, an asymmetric key algorithm. Due to this, the verification of a token takes 30 to 35 seconds, due to ECC operations, which is significantly longer than the proposed system.

The platform owner server, which manages and grants the tokens and the application owner server which requests the tokens and deploys the applications are both implemented in Java.

6.4 Secure Application Management

The secure application management system allows for the secure management of the embedded node by authenticating and authorising management requests. The protocol requires a separation of any management service into two subcomponents: the marshaller component and the service component. The marshaller interprets incoming service requests, accesses the service, and serializes the reply. The service contains the actual data and logic to perform the requests. This division is common in RPC architectures such as CORBA and RMI. The system consists of: a communication interceptor module, a service interception layer, a user data store, and a user management service [71]. This section first describes how to secure application management system authenticates users. Second it describes how the system authorises users. Figure 6.6 shows an overview of how the system authenticates and authorises a request. Third it describes the user management subsystem. And lastly, this section describes the implementation details and a short evaluation of the secure

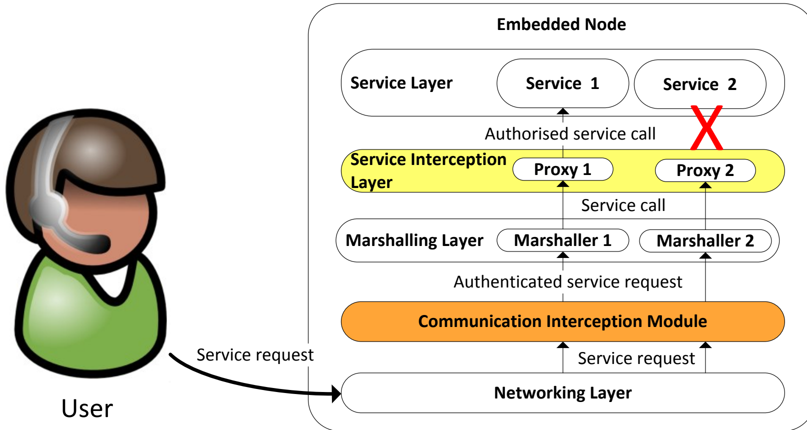


Figure 6.6: Overview of the authentication and authorisation of a service access request.

application management system. This work has been published at the 10th European Conference on Wireless Sensor Networks (EWSN 2013 [71]).

6.4.1 Authentication

The system uses a simple authentication mechanism. Each user is identified by the embedded node using a node specific numeric identifier. The user authenticates and secures his commands using a symmetric encryption algorithm in Counter with Cipher block chaining-MAC (CCM) mode. This provides the nodes with proof of authenticity and integrity of the payload using a Message Authentication Code (MAC). The system is agnostic to the actual cryptographic protocol used. The actual implementation of the protocol uses AES128 due to security, resource requirements and standardisation considerations. The userId and MAC are added to the message as headers. An authenticated timestamp is optionally added to ensure message freshness.

When a node receives a service request, it is intercepted by the communication interceptor module before being delivered to the marshaller. The communication interceptor module retrieves the userId and MAC from the message headers. It retrieves user information from the user database (Policy Information Point: PIP using XACML terminology [39]). If the userId is known, the communication interceptor module decrypts and verifies the payload. If validation fails, either because of incorrect MAC or unknown userId, the message is dropped. On success, the request is delivered to the marshaller. When the marshaller

sends a reply, it must reattach the userId. The communication interceptor module intercepts this reply, encrypts the payload, and attaches a message authentication code.

6.4.2 Authorisation

To authorise a user, an authorisation proxy is inserted between the marshaller, and the service execution component. This proxy offers the same interface as the service execution component, thus can be inserted transparently with minimal effort. The proxy acts as the Policy Enforcement Point (PEP) and Policy Decision Point (PDP). It uses role based access control to authorise a user. The system uses hard coded role requirements for each service. This choice was made to reduce the overhead of evaluating policy decisions, at the cost of flexibility. The proxy retrieves the user's current roles from the user database on the node, and verifies whether the user's role is sufficient.

The system currently differentiates between two types of roles: node roles and party roles. A node role defines the access permission that the user has across the entire node. This allows the platform owner to compactly declare that a user can view or reconfigure any configuration or service on the node or to perform certain node-wide reconfigurations. A party role specifies the role that the user has with that party and is only relevant for applications and configurations owned by that party. A user of a node can thus have one node-wide role, and one role with each party currently present on the node.

The system currently distinguishes 5 different access roles, listed in hierarchical order: 1) **no access**: no access to any service, 2) **viewer**: viewing information and configuration, 3) **user**: modifying existing configurations, 4) **manager**: creating and removing configuration, and 5) **administrator**: user management. Currently a higher roles also assumes all access rights of the lower roles. The number of roles and allocation of access rights is a generic framework. Roles and rights can easily be modified to adhere to domain specific requirements.

When a user deploys a piece of code, or instantiates a component, the action must be attributed to a party. Either the user's default party is used, or the user specifies on behalf of which party he performs the creation operation. Of course the user needs the necessary service permission in order to create new configurations. Further service requests regarding the added application will require that the requesting user has the necessary role to either that party, or node-wide.

The authorisation proxy also allows for monitoring the behaviour of users of the system. Monitoring node users allows for 1) detecting potential intrusion

	Logistics Provider (LP)	Transport Provider (TP)	Customs Officer
Lock app : LP	node admin	party viewer	party user
Location app : TP	node admin	party admin	party viewer

Table 6.3: Overview of the permissions of the different users with the different parties and their applications.

attempts, and 2) logging of embedded node usage caused by the different users, allowing charge-back of node usage to the users. If monitoring is required, the proxy reports the user's node usage to the monitoring and enforcement system. This system can then store the node usage data. This also allows for policies limiting the amount of requests a user can send.

An example from logistics is show on table 6.3: the logistics provider provides the node in the container with a lock application. The logistics provider user has administrator rights to the node and thus each service present on the node, allowing him to open, close and manage the lock. A customs officer has party user permissions to services of the logistics provider. The officer is allowed to use the lock service, which includes opening and closing the lock, and viewing the lock status. The customs officer however is not allowed to manage which other parties have access. The transport provider user finally has only party viewer permission. He can only view current lock status.

Suppose the customs officer installs a localisation component on the node, which queries the truck and broadcasts it to all users registered on the node. The customs user is party administrator, and can administer which parties are allowed to view the location feed, as agreed upon in the contract that the customs and logistics provider need to have signed beforehand. The logistics provider user is a node administrator, so he can also administer who can view the feed on the node, such as cargo owners or transport providers. The transport provider party is allowed to use services offered by the logistics provider, but only has view rights to applications offered by customs.

While the current security middleware only offers access control based on party roles, a small addition to the system would allow additional attribute based security, enabling parties only rights to certain allowed ranges of applications, even while having node viewer permissions. However, we decided not to implement such a system, to reduce the overhead.

6.4.3 User Management Service

User management is done by the user management service, making it the Policy Administration Point (PAP). The provided methods are: adding and removing users, adding and removing permissions, and updating key material.

The user management service provides two ways to add users, by command or by token. A service method allows users to add new users by command if he has the necessary permission. To add a user by token, the user requests the platform owner server to generate a token on his behalf. The platform owner then validates that the user is affiliated with the party for which he requests permission. If the user is affiliated with a party which is permitted the usage of the node, for example because that party has a component installed on that node, the platform owner server generates a user ID for that user with that specific node, and a specific user key. The platform owner then stores this key and userID, provides them to the user, and encrypts them in a user token. This user token is then provided to the user for when he needs to access the node.

This user token contains the following parameters: userId, partyId, node role, party role, key material, timestamp and timeout. The timestamp and timeout of the token ensure that a token will only be valid for a limited amount of time, allowing node recovery. This time is configurable, allowing for tokens with a longer validity. These tokens can be requested ahead of time, and deployed in disconnected networks. The generated token is encrypted with the node's secret key, shared only between the platform owner and the node. The node does not need to contact back-end infrastructure to ensure token validity. Once a user is added by command or token, the user only needs his userId and key to query node services.

Other provided services of the user management service include a user revocation service, a role management service, and a key management service. The user revocation service allows users with a node administrator role to remove any user on the node. Party administrators are allowed to delete users belonging to their party. The role management service allows users with a party administrator role to add and remove roles of that party from other users. This allows parties to manage access rights of users to their own services. The key management service allows users to refresh keys. However this does not ensure forward key secrecy. If a user's key is breached, the attacker can intercept the re-keying message and retrieve the new key. If this is detected, the user has to be removed. The user can be reinstalled by a node administrator or by using a new token.

If a user needs to manage multiple nodes, the same symmetric key can be used to allow group reconfiguration. The user then must request the platform owner to use the same key as his user key. This reduces the amount of messages the

LooCI Header	SecLooCI User Header	SecLooCI MAC Header	SecLooCI Timestamp*	SecLooCI Payload (enc)
6B	4B	10B	6B	*

Figure 6.7: Packet format of a SecLooCI management communication message.

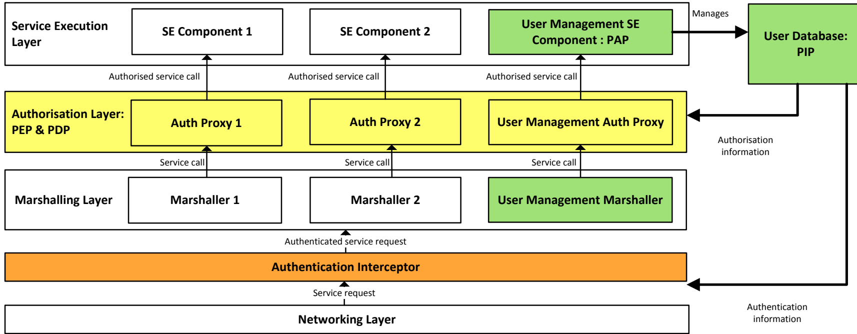


Figure 6.8: Overview of the Application Management protocol. Colored parts are components added for security: authentication interceptor (orange), authorisation framework (yellow), and user management (green).

user has to send to perform reconfiguration, but at the cost of security. In case of a node breach, the user's key would be revealed and could then be used to access all other nodes on which the user used the same key.

6.4.4 Implementation Details

This subsection details the specific implementation details of the prototype and evaluates the communication, memory, and processing overhead of the secure application management framework. This approach is compared to a symmetric key approach (TinySec [58]) and an asymmetric key approach (Authenticated Querying (AQ) by Beneson et Al [7]). TinySec is a prototypical example of a security network layer for embedded systems. It offers both encrypted or authenticated communication based on symmetric key algorithms. Authenticated Querying on the other hand is a typical example of a certificate based approach to sending authenticated messages, and operates using Elliptic Curve Cryptography.

To identify users, the system uses a 2 B numeric userID. A 2 B userId provides a sufficient number of users while ensuring limited overhead. This compact user identification can be node or network unique. Each authenticated message

is encrypted using AES128-CCM and has an 8 B MAC in the secure payload header to verify message authenticity and integrity. At service call, the service interception layer intercepts the call and verifies the user is allowed to access the service based on the roles of the calling user.

The current security middleware implements a hierarchical role based access control scheme. 4 roles are defined: viewer, user, manager, and admin, each with increasing powers of viewing application state, and modifying and deleting applications. A viewer can only see data, users can subscribe to existing data producing components, managers can adapt the settings of these components, and admins can influence the lifecycle (pausing, removing and starting components). These roles allow for a simple, clear and compact description of the required role for each functionality. However, the framework allows service creators to adapt the system and implement their own access control scheme.

In order to authorise an event, an addition was made to the LooCI middleware: when an event is dispatched to a marshaller, the dispatching user is recorded by the authorisation middleware. When a marshaller calls any proxy protected service execution component, the authorisation proxy intercepts the call and verifies that the requesting user has sufficient permissions to access the requested service. If the user is authorised, the proxy calls the management service execution component. If not, an error code is returned to the reconfiguration manager. When the marshaller sends out a reply, the middleware attaches the userId.

The user management service is implemented as a LooCI component, and is comprised of a marshaller, which interprets received messages, and a SEC, which does the actual user management and contains the user data and access policies. The service execution component thus performs both the functions of PIP and PAP in order to reduce implementation size.

Communication overhead: The secure application management framework requires additional security information to be added to each management message. This overhead is 14 B and is detailed in Figure 6.7. The payload of the messages is currently not padded to ensure the minimal size of reconfiguration messages. TinySec has a message overhead of 8 B, AQ has a message overhead of 20 B, both are comparable to the proposed system.

To register a user, it is possible to deploy a user token to the user management system. This requires the transmission of a token over the network containing the following fields : userId, partyId, nodeRole, partyRole, user-Key, timestamp, timeOut, user-timeout, and MAC. This token has a total message size of 40 B and is encrypted with the node's long term secret key. Installing the user by

Table 6.4: Secure application management memory overhead.

	ROM (B)		RAM(B)
AES-CCM crypto lib	4746	User information store	238
Communication interceptor	407	Encryption buffer	200
Event encryption support	1150		
Permission module	1158		
Service Interception	3296		
Security management component	1393		
Total ROM	12150	Total RAM	438
TinySec ROM	7 148	TinySec RAM	728

token is one option to allow new users to be added. Alternatively a service request can be called which has similar communication overhead, since the same data has to be transmitted. To compare, AQ has a user registration token size of 114 B due to ECC signature. TinySec does not mention any user registration token.

Memory overhead: The ROM overhead of the application management component is 12.15 kB, and the RAM overhead is 438 B. Table 6.4 contains a detailed overview of the overhead. To compare, TinySec has a ROM overhead of 7 148 B and RAM overhead of 728 B, AQ has a significantly larger ROM and RAM overhead of respectively 45 500 B and 2 000 B.

Processing overhead: The secure application management framework operates at three point in the call chain: at message reception, at service call, and at message transmission. The overhead of message transmission and reception are almost identical and composed of retrieving the user's key information from the user data store, and then either encrypting or decrypting the message. The authentication of a message comprises of (a) retrieving the userId from the request message, (b) retrieving the user information from the user data store, and (c) decrypting and verifying the authenticity of the message. This takes ca 3.2 ms for a 32 B management request (mostly decryption). Encryption of reply messages takes a similar time, since the main overhead is the encryption of the request. The authorisation of a message is much faster. It comprises of (a) a proxy interception of a service message, (b) retrieving user access rights, and (c) matching access rights with the current request. This requires only tens of nanoseconds. TinySec has a message processing overhead of only 1.04 ms or 1.52 ms, due to the usage of different encryption algorithms (RC5/SkipJack), and different hardware.

A user can register himself with the system using a token. This message is not intercepted by the Communication Interceptor to allow anyone to send such a token. However, the service itself verifies the integrity and validity of the

token by decrypting and authenticating the token. Since the token size is about 40 B, it takes about 4.5 ms to decrypt and verify the token. To compare, AQ requires 440 000 ms to validate the user registration token due to the usage of asymmetric cryptography, and requiring a certificate chain. TinySec does not allow for user registration.

6.5 Secure Application Communication

The secure application communication framework ensures that the application communication of the different application components is transmitted securely. In the LooCI Component middleware, communication happens using semantically typed events. The framework allows users to specify encryption and decryption policies based on these event semantics.

The framework distinguishes between encryption contexts, outgoing policies and incoming policies. Encryption contexts identify a security context which entails algorithm type, algorithm identification, key material and length of MAC. Outgoing policies dictate which events should be transmitted on which channel based on event type, source component and destination. Incoming policies dictate which level of security is required for certain events, and which events are allowed to pass, based on source node, source component, event type and context. The secure application communication framework operates at the transmission and reception of an event. An overview of the decision logic can be found in Figure 6.10.

When a message is dispatched from the event bus down, it is checked against the outgoing policies. If the event matches an outgoing policy, the associated channel is retrieved and the event is processed according to the channel parameters. If an event matches multiple policies, only the first matching policy is considered, which is the policy which has been on the node the longest. The system does not use multiple different encryption contexts. This decision is made, because in the LooCI framework, an event is sent once for each subscription. The framework assumes that each subscription can only be transmitted on a single channel. This prevents the need for duplication logic in the framework. Overlapping policies can be detected based on the source, destination and event type of the channel.

When an event comes up from the network stack, the encryption context is checked against present encryption context. If the identified encryption context is present, the matching security primitive is performed. Once the verification is completed, the event is matched against the incoming policies. The incoming policies dictate the minimal level of security that is required for an event to pass.

LooCI Header	SecLooCI Channel Header	SecLooCI Info Header	SecLooCI Timestamp*	Event Payload
6B	4B	0/6/10/18 /34 B	6B	*

Figure 6.9: Packet format of a SecLooCI application communication message.

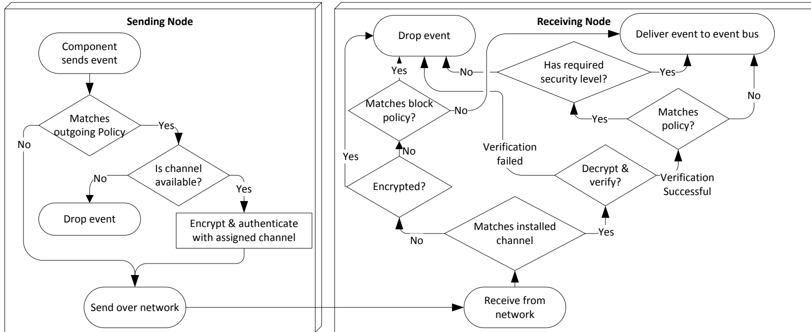


Figure 6.10: Flowchart of the decision tree of policy interpretation at sending and receiving application messages.

If the level of security provided for the event, matches the required security level, the event is submitted to the event bus. Additionally, it is possible to dictate a blocking policy that blocks all events, unless another policy explicitly allows it. This allows for example a platform owner to block all event types by default, unless a policy exists that explicitly allows it.

This remainder of this section evaluates the communication, memory, and processing overhead of the secure application communication component, and compares it with TinySec [58]. TinySec provides a link layer security architecture for networked embedded systems. It offers both confidential or authenticated communication, similar to the application communication framework. However, the choice can only be made at a network level, as opposed to the policy based approach of the SecLooCI framework.

Communication overhead: The security middleware implements 3 types of security policies: integrity, authenticity and confidentiality. Each message is identified by (a) a security information header, containing the channelId (2 B), and (b) a security payload, containing the hash or MAC payload, as shown in Figure 6.9. The size of the security payload depends on the exact policy installed. The middleware allows for a flexible choice of security and supports multiple lengths of security messages. For integrity and authenticity a security payload of 4, 8, 16 and 32 B is supported, if supported by the underlying algorithm, for confidentiality a MAC size of 0, 4, 8 and 16 B is supported.

Table 6.5: Secure application communication memory overhead.

	ROM (B)		RAM(B)
Communication interceptor	1 627	Storage buffer overhead	21
Security management component	1529	Overhead per channel	31
AES with CCM	4 746	Overhead per inc pol	7
AES-CMAC	1 018	Overhead per out pol	7
SHA256	1 980		
Total ROM	10 900	Total RAM (2 ch, 1 in, 1 out)	97
TinySec ROM	7 148	TinySec RAM	728

The total message overhead is 4, 10, 14, 22 or 38 B depending on the security payload size, and thus the level of security. The application owner can choose his level of security depending on the security requirements of the application, the available resources of the embedded node and the preferences of the platform owners involved in the deployment. TinySec has a fixed MAC overhead of 8 B.

Memory overhead: The prototype has a total ROM overhead of 10 900 B and a RAM overhead of 97 B assuming 2 channels, 1 incoming and 1 outgoing policy. Transmission buffers and encryption buffers are not included in this, and depend on specific transmissions and encryption protocols. Table 6.5 shows the overhead figures in more detail. Note that 7 744 B of the implementation overhead is taken up by cryptographic algorithms, which are reused in other parts of the security middleware. TinySec has a ROM overhead of 7 148 B and RAM overhead of 728 B.

Processing overhead: Processing a 32 B message takes: to encrypt / decrypt the message using AES: 3.1 ms, to authenticate or verify a message using AES-CMAC : 2.6 ms, to verify message integrity using SHA1 : 13.6 ms. While this processing overhead is significant, it is still only a few milliseconds of processing overhead. In most use cases this is a tolerable overhead. TinySec has a message processing overhead of only 1.04 ms or 1.52 ms, due to the usage of different encryption algorithms (RC5/SkipJack).

6.6 Monitoring and Enforcement

The monitoring and enforcement component FAMoS [69] monitors node usage by different components and users and allows policy enforcement.

The monitoring and enforcement framework operates by instrumenting the functions required to use the network, memory and sensors. It defines multiple different hooks to be able to monitor performance on different levels of the network, including the MAC and IP layer as shown on figure 6.11. Each

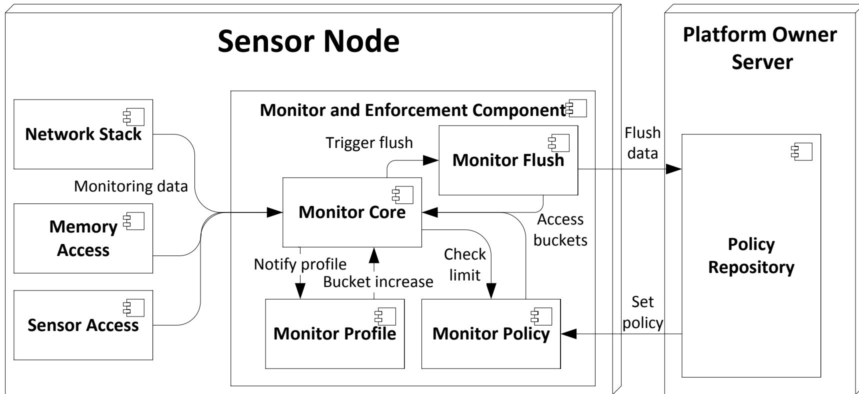


Figure 6.11: Overview of the monitoring and enforcement component.

hook has a 16 bit numeric id that uniquely identifies this hook. Additionally the monitoring module receives data from the Communication Interception Module (timings of interception overhead), the LooCI event bus (events received and transmitted), the service interception layer (user accesses), and the application layer (CPU overhead of application components). These hooks call the monitoring component when requests or data passes through them. In addition, users can define their own hooks and extend the hook ids to add their own logging calls. This work has been published at the 12th International IFIP Conference on Distributed Applications and Interoperable Systems (DAIS 2012 [69]).

The monitoring and enforcement module on the node is comprised of 4 blocks: the core, the profile, the policy and the flush component. The core block maintains the buckets, which are basically resource counters that keep track of the resource usages. It contains two types of buckets: basic buckets and extended buckets. Basic buckets, which store the throughput and count of a single hook, and extended buckets, which allow additional meta data to be appended to the bucket, such as for example the TCP port number, the user identifier, or component identifier. On a fixed schedule the monitor flush block flushes all buckets in the core back to the platform owner, which causes the core to clear all the buckets inside it.

The monitor core must support two operations: request usage permissions, and report usage. Certain hooks only implement the reporting functionality, others also implement the request permission functionality.

When a hook reports a node usage update, it sends it to the core. The core then contacts the profile component. The profile component receives the hookId,

any potential call context, and can retrieve the current environment context. Based on this data, the profile component then decides which buckets need to be incremented, by which amount. The identifiers of the relevant buckets are then returned to the core block, which updates those buckets.

Next the core block notifies the policy block with the identifiers that need to be updated. The policy block contains resource limitation policies which dictate which identifiers need to be rate limited, and operates according to a leaky bucket algorithm. For each policy, the policy block keeps a private bucket that does not get flushed. If an updated identifier matches a bucket, the current available limit in that bucket is decreased by the relevant amount. Every minute however, the policy block adds a small amount again to the buckets, to a certain limit, based on the relevant policy.

Next when a service interceptor or an other component needs to verify if a resource usage policy prevents the usage, it contacts the core block. The core block again contacts the profile block, which translates the request to a set of identifiers which need to have a limit available. The core block then contacts the policy block to check if there are policies installed for the relevant identifiers, and if so, whether or not capacity is available.

This division is done for the following reasons: by isolating the profile block from the storage and flushing block, it is possible to transparently change the profile and the policy block from the storage and flushing. This allows for over the air deployable profiles to be installed with minimal overhead, since storage and flushing are shared capabilities. By having the policies depend on the profile block, the policies can be installed in a very compact form. Each policy just implements a single leaky bucket, so a new policy can be expressed by expressing which bucket id needs to be limited, what the max limit of the bucket is, and the amount by which that bucket gets refilled every minute.

During node installation time, a platform owner can decide which profile to install on his node, and thus creates an image with the necessary hooks and profile block. Unnecessary hooks can be removed during compilation to reduce unneeded overhead. However any over the air deployed profile can then only access data from those hooks that are installed. The monitoring and enforcement component can potentially operate independent from the other proposed pillars. However, by using the additional security pillars, the monitoring traffic benefits from the additional security features presented, such as secure deployment and secure end-to-end communication.

If no over the air profile changes are likely to be needed, and no policies need to be enforce, the overhead of calculating the bucket identifier, storing those identifiers and flushing the buckets together with identifiers can be a significant

overhead. To reduce that overhead, a simple monitoring module can then replace the extended module. This simple monitoring module would then have to implement its own storage and transmitting functionality, with custom functionality on the receiver side to parse the binary message. While this significantly reduces both storage and network overhead, it does also reduce evolvability of both monitoring profiles and enforcement policies.

This section evaluates the communication, memory, and processing overhead of the full monitoring and enforcement module implementation on an AVR Zigduino and compares it to the Sympathy network debugger [94]. Sympathy is an active monitoring solution embedded systems. It is mainly used for debugging systems, and allows getting fine grained information about the network stack. It however does not provide service, user or application monitoring.

Communication overhead: The communication usage can be divided into two parts: (a) the overhead to send a monitoring flush packet, and (b) the overhead to install a usage limitation policy. The overhead to send a monitoring flush packet depends on the flush interval. A flush packet has a minimal size of 8 B. Each bucket that contains data extends this packet size with 6 B. On top of this is the network overhead, which can be 30 B to 40 B of IP and MAC headers. Deploying a usage limitation policy has a communication overhead of 8 B per policy. Sympathy does not explicitly mention monitoring packet size or overhead. Assuming 4 B per timestamp and 2 B per counter, Sympathy has a monitoring message size of 20 B, which is comparable to the proposed component.

Memory overhead: The basic framework without policy enforcement requires 3 290 B of ROM and 216 B of RAM. The policy enforcement framework requires another 2 140 B of ROM and 221 B of RAM. This is for a minimal monitoring policy. More complex policies will require more ROM and RAM due to the larger amount of monitoring logic, and amount of buckets to monitor. Sympathy has an overhead of 47 B of RAM and 1 558 B of ROM. This lower overhead is due to the fact that Sympathy has fixed monitoring which cannot be updated or changed, and can not enforce policies, so the monitoring and enforcement component has a slightly higher overhead for some additional features.

Processing overhead: The processing overhead of the prototype is minimally 57 ns. This can run up to 200 ns or more depending on the complexity of the monitoring and enforcement policies. The default case of sending a packet over the network takes on average 1260 ns network processing, and 2228 ns to actually transmit the message, totalling at 3488 ns. 200 ns is only a 5.8% additional overhead. Sympathy does not mention any processing overhead, but due to the static and simple monitoring policy of Sympathy, this paper assumes it can be compared to a very simple monitoring policy, requiring 50 ns.

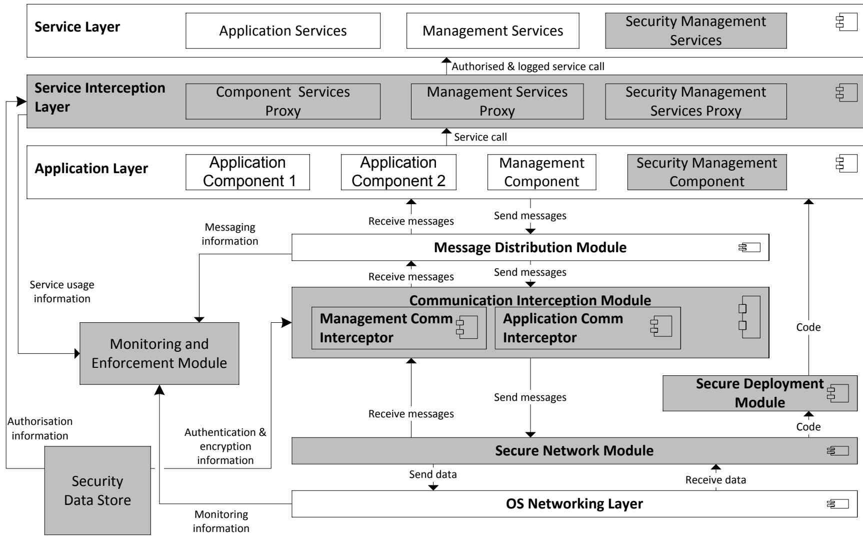


Figure 6.12: Overview of the Secure Node Architecture.

6.7 Integrated Security Framework

This section provides an overview the integrated security framework and proofs the feasibility of rolling out the security framework on resource constrained nodes by providing a static evaluation of the integrated prototype of the architecture. The SecLooCI node security framework adds 7 additional modules over the standard LooCI middleware, in accordance with the general security architecture and as show in Figure 6.12 :

- The **secure network module** secures the **network initialisation**, which is transparent to the upper and lower layers, and allows the creation of an embedded node registry at the gateway.
- The **communication interceptor module** performs all the **authentication and encryption** for both application and management communication.
- The **secure deployment module** secures the **application deployment**.
- The **security management component** offers **security management** services such as user installation, channel and policy management for the application security, and policy management for the monitoring and enforcement component.

- The **service interception layer** provides the interception points necessary to enable **authorisation of management communication and monitoring of application service usage.**
- The **monitoring and enforcement module** monitors the network and the **application service usage**, and potentially also the different users of the network.
- The **security data store** contains the necessary user and policy information needed for the application and management message security protocols to operate.

These seven modules are implemented and integrated on the Zigduino running the Contiki OS and the LooCI Component Middleware. Table 6.6 shows an overview of the overhead of the SecLooCI middleware compared to the unsecured LooCI and Contiki. Table 6.7 shows the ROM and RAM overhead of the different components of the middleware. The total additional overhead for the security middleware including cryptographic primitives is 27 372 B of ROM and 1 525 B of RAM memory. This is a significant amount of memory for these memory constrained devices. However, the security features which are offered by this middleware are necessary for the next generation of sensor network applications, and the implementation of the middleware shows that it is feasible to implement the required features on the targeted memory constrained devices, with a significant yet acceptable overhead.

Note that 28.3% of ROM and 9.3% of RAM is due to cryptographic primitives. A current trend is to implement these algorithms in hardware, to reduce processing cost and memory overhead. This would significantly reduce the overhead of the framework.

This integrated prototype shows that the SecLooCI node security framework is small enough to be deployed on currently available constrained embedded nodes. While it does consume a significant amount of memory, it is clear that currently available constrained devices offer sufficient memory to support these systems and to build applications on top of the security middleware. Additionally, thanks to the LooCI and SecLooCI middleware, the application components need to only focus on the application functionality, while message distribution and security are handled on the middleware level. This reduces the size and complexity of these components, partially reducing the burden on developers, and on devices to support evolvability.

Table 6.6: SecLooCI framework memory overhead. The overhead of the security suite is comparable to the overhead of the non-secure middleware, but clearly still within the possibilities of a low power sensor node.

	ROM (B)	%	RAM (B)	%
Contiki operating system	42 688	32.6%	9 712	59.3%
LooCI middleware	24 942	19.0%	2 644	16.1%
SecLooCI security framework	27 372	20.9%	1 525	9.3%
Total binary	95 002	72.5%	13 881	84.7%
Total available on AVR Zigduino	131 072	100.0%	16 384	100.0%

Table 6.7: Detailed SecLooCI framework memory overhead. A significant amount of ROM and RAM is used for the encryption algorithms. Hardware implementations could reduce this overhead.

	ROM (B)	%	RAM (B)	%
Secure network module	946	3.5%	218	14.3%
Communication interceptor module	2 034	7.4%	129	8.5%
Secure deployment module	512	1.9%	20	1.3%
Security management component	2 780	10.2%	295	19.3%
Service interception layer	3 296	12.0%	8	0.5%
Security data store	1 558	5.7%	238	15.6%
Monitoring module	3 290	12.0%	216	14.2%
Policy enforcement module	2 140	7.8%	221	14.5%
Middleware tools	3 072	11.2%	38	2.5%
Encryption algorithms	7 744	28.3%	142	9.3%
Total SecLooCI usage	27 372	100.0%	1 525	100.0%

6.8 Summary

This chapter presented the SecLooCI shared embedded node security framework. It started by presenting the underlying platform, explaining the test setup, and listing the assumptions required to create the node framework. Next it presented the five different security subsystems which each secure one of the five previously identified target data flows:

- The secure network initialisation subsystem secures the network setup by allowing nodes owned by multiple different platform owners, to securely join the network of the trusted network owners.
- The secure application deployment subsystem allows application owners to securely deploy new application components on the nodes of trusted platform owners.

- The secure application management subsystem allows any node user to securely manage either the application, middleware or operating system of the node.
- The secure application communication subsystem provides policy driven security for all embedded application communication based on event semantics.
- The monitoring and enforcement subsystem continuously monitors users and applications on the node and enforces usage policies.

This chapter concluded with an overview of the integrated prototype, which shows that current generation networked embedded devices can support the entire stack of Contiki Operating System, LooCI middleware and SecLooCI security framework. This shows that current state-of-the-art devices have the capabilities to support shared usage of the networked embedded systems.

Chapter 7

Case Studies

This chapter presents two case studies where we apply the SecLooCI infrastructure and SecLooCI node security middleware to secure two scenarios based on the uses cases presented in Section 2.1. First this chapter presents a case study inspired by the smart logistics use case, based on the life cycle presented previously. The scenario goes through the lifecycle, details the SecLooCI infrastructure operations, and identifies the various overheads. Second this chapter presents a real world deployment cases study based on the smart office use case.

7.1 Smart Logistics Scenario

This section presents an virtual integrated scenario, provides an overview of the overhead of the integrated prototype, and performs an applied evaluation of this comprehensive framework using the smart logistics scenario proposed in Section 4.6. It shows the steps in a typical lifecycle, which systems interact with each other, and the associated overhead.

7.1.1 Test Scenario and Measurements

This section implements the end-to-end scenario developed in the context of the ICON COMACOD and ICON STADIUM project, as presented in Section 4.6, where a harbour customs wants to deploy a simple localisation application on the containers that arrive in the harbour. The sample monitoring application is

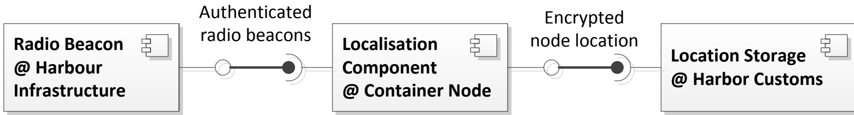


Figure 7.1: Example Localisation Application.

shown on Figure 7.1. This scenario is prototypical for a single hop network, which we use as our default test environment as discussed in Section 6.1. The time overhead and message size of each node interaction is listed in this scenario. Only node interactions are listed, excluding back-end to back-end communication, since only node interactions influence node lifetime. The memory overhead of the different components, which is mostly static, can be found in the previous section. For each interaction the total cost is listed. For an in depth view of all the different costs, please refer to table 7.1. The test setup consists of two embedded nodes and a gateway in a one hop network. The scenario does not consider multi-hop networking. Network and MAC layer overhead is not considered.

Network setup The first step in the network setup is the beacon that is sent across the network. This beacon has a size of 8 B containing the MAC address of the gateway. Once received, the embedded node generates the node token of 32 B, and transmits it. This takes ca 4 ms, mostly encryption. The NO receives and processes the token, contacts the PO, receives a network token of ca 44 B, and transmits it back to the embedded node which takes on average ca 100 ms, of which the embedded node requires another 6 ms to decrypt, validate and process the reply. Most of this time the embedded node is waiting for a reply to his key request. The total cost is 32 B sent, 52 B received, taking ca 100 ms, of which 10 ms is spent computing by the embedded node.

Application deployment The next embedded node interaction is the customs officer who wants to deploy a new application component. In this case, the customs officer deploys a localisation component, which listens to beacon events, does some very simple distance estimation in 2 dimensions based on the delay of the event and beacon location, and publishes its location every 60 s if it has received a beacon. A prototype component has a size of 2 121 B. To deploy the component, the customs officer first sends the application token, with a size of 56 B, and then sends the component. The embedded node only sends a success byte and the component identifier, totalling 2 B. This process takes about 12.3 s. Only about 8 ms were spent decrypting and validating the token, and 215 ms performing the hashing. This totals at 2 177 B received, 2 B sent, taking ca 12.3 s, of which 223 ms spent on crypto operations, and ca 10 s spent writing to flash memory. The remaining time is communication overhead.

Application management The next interaction is the management of the users and embedded applications. The necessary operations are: deployment of the user token by the Customs Officer, entering the five beacon locations in the application, entering the necessary subscriptions, adding the security policies (incoming channel and policy, outgoing channel and policy, and monitoring policy), and activating the component. The policies use an 8 B MAC for encryption and authentication. Each of these interactions is a service call by the customs officer to the management component. A total of 14 configuration requests are needed. The prototype performed this configuration in an automated way, taking about 3.45 seconds for end-to-end management operations. Each request requires an overhead of 14 B for security headers, and 6 of the 14 messages are security management messages. The total cost is 587 B sent, 573 B received, and only takes ca 48.3 ms of on-node processing.

Application communication At this point, the application is running and communicating over secured channels. To secure the messages, we require authenticity of the beacon messages, and confidentiality of the location messages. So when a beacon message enters the node, it is intercepted by the communication interceptor, the incoming channel is retrieved, and the MAC is validated. This takes about 1.3 ms. The incoming beacon message has a 16B MAC, 7 B security metadata, 4 B of LooCI overhead, and 9 B of functional data, totalling at 36 B. Then the message is sent up to the LooCI Event Manager, which delivers it to the localisation component. The localisation component calculates its current position based on available information, and broadcasts it. The broadcast is logged by the monitoring component, which takes a few nanoseconds. The event manager receives the event and routes it to the network. The communication interceptor intercepts it, checks outgoing policies, finds the confidentiality policy, and encrypts it, which takes again about 1.6 ms. The largest part of this overhead is encryption. The total process from receiving a radio beacon from the network, to sending the node location only takes about 3.9 ms. The send overhead is the cost for encrypting and authenticating the event, requiring 19 B. The message itself is 9 B. Hence, the total cost is 32 B sent, 36 B received, taking ca 3.9 ms. Assuming one message per minute, this becomes a daily total cost of 46 080 B sent, 51 840 B received, and 5 616 ms spent processing.

Application removal Once the node travels on, the customs will need to remove all components and policies related to the localisation application. To do this, he has to: 1) remove the codebase, which also removes the previously installed application component and any relevant wiring policies, 2) remove the inbound security channel, which also removes the relevant inbound security policy, 3) remove the outbound security channel, which also removes the outbound security policy, and 4) remove the customs officer user on the node.

Table 7.1: Overview of the SecLooCI framework overhead. Daily overhead calculated assuming one incoming and outgoing message per minute. Table shows that processing overhead is comparable to one day of operation. Transmission overhead is equal to about 19 days of operation.

	LooCI	%	SecLooCI	%	Crypto	%	Total
Network setup sent(B)	0	0.0%	16	50.0%	16	50%	32
Network setup received(B)	0	0.0%	36	50.0%	16	50%	52
Network setup processing(ms)	0	0.0%	0	0%	12	100 %	12
Deploy sent(B)	2	100%	0	0.0%	0	0.0%	2
Deploy received(B)	2121	97.4%	16	0.7%	40	1.8%	2177
Deploy processing(ms)	10 000	97.8%	0	0%	223	2.2 %	10 223
Mgt sent(B)	172	29.3%	303	51.6%	112	19.1%	587
Mgt received(B)	164	28.6%	297	51.8%	112	19.5%	573
Mgt processing(ms)	.42	0.86%	0.28	0.58%	47.6	98.5%	48.3
Total sent(B)	174	28.0%	319	51.4%	128	20.6%	621
Total received(B)	2285	81.5%	349	12.5%	168	6.0%	2 802
Total processing(ms)	10 000	97.2%	0	0%	291	2.8%	10 291
App comm sent per message(B)	9	28.1%	4	12.5%	19	59.4%	32
App comm rec per message(B)	9	25%	4	11.1%	23	63.9%	36
Time to process a message(ms)	1	25.7%	0	0.0%	2.9	74.3%	3.9
App comm sent per day(B)	12 960	28.1%	5 760	12.5%	11 520	34.8%	46 080
App comm rec per day(B)	12 960	25%	5 760	11.1%	11 520	34.8%	51 840
Daily message processing(ms)	1 440	25.7%	0	0.0%	4 176	74.3%	5 616

This is thus a total of 4 configuration requests, which can again be done in an automated way. Each request again is secured requiring 14 B, and 3 of the 4 messages are security management messages. The total cost is 75 B sent, 80 B received, taking ca 0,6 s.

7.1.2 Summary

The integrated prototype shows that the overhead of the SecLooCI security node framework is small enough to still fit on micro-controller level devices. The prototype overview showed that the overhead of the static system is significant, but comparable to other security solutions proposed by related work. The previous section has shown that the prototype communication overhead from security at the setup time is fairly limited, and exists mostly of necessary security meta-data. The deployment of a single application component, together with all the relevant security policies requires the transmission of the same number of bytes as would be sent during ca a single day of operation. This thus leads us to the conclusion that the current generation of embedded nodes can support secure node mobility, and secure software evolution.

7.2 Smart Office Deployment

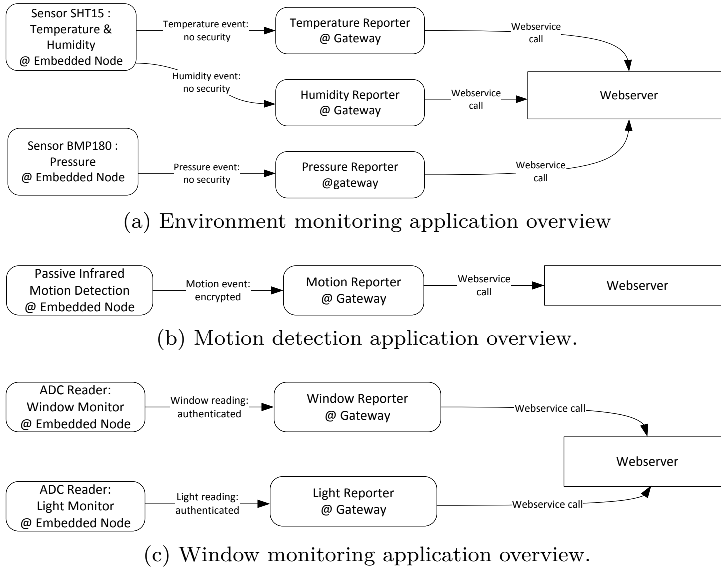
This section discusses the DistriNet Smart Office deployment, and its results, which is an implementation of the smart office use case as presented in Section 2.1.1. First it details the different applications running in the smart environment. Next it lists the different security requirements of the applications. Third this section reports on the communication overhead as monitored by the monitoring system. Last, this section discusses a few observations made during the construction and evaluation of the smart office deployment.

The security and management of the lab is done using the SecLooCI infrastructure. All applications and security requirements are modeled using the end-user tool. The application owner server stores these descriptions, and is continuously monitoring that the deployment remains in a valid state. Every hour, each application is verified to ensure that all components are deployed, parameters set correctly, and that the security policies are correctly installed on the nodes.

7.2.1 Applications

This section introduces the setup and the applications in the DistriNet smart office deployment. The smart office consists of the smart office server, which provide data storage and processing (a standard pc), the gateway node which offers access to the embedded network (a Raspberry Pi), and 8 sensor nodes providing sensing and actuation (AVR Zigduino's). This smart office evaluation focuses on 3 applications: 1) an environment monitoring application, 2) a motion detection application, and 3) a window monitoring application. These applications were inspired by the ITEA DiY Smart Experiences project, and were made in consultation with the office administrators. The remainder of this section details these three applications.

The first application is a simple environment monitoring application. One sensor node is deployed in the environment with two sensors, one which monitors the current humidity and temperature and another which monitors the pressure in the room. Two applications poll the sensors, and send the readings to the gateway, which stores it on the web server, as shown in Figure 7.2a. The first applications measures the temperature and humidity of the room using the SHT15 sensor. The second application measures the atmospheric pressure in the room using the BMP180 sensor. The applications sample the environment every 5 minutes. This data is then sent to the gateway, and stored on the server. The server then allows for long term monitoring, and can potentially



(a) Environment monitoring application overview

(b) Motion detection application overview.

(c) Window monitoring application overview.

Figure 7.2: Overview of the applications of the DistriNet smart office deployment.

warn building administrators when the temperature becomes too high or too low.

The second application is a motion detection application, as shown in Figure 7.2b. 5 sensor nodes, distributed across the smart office, are equipped with passive infrared motion sensors. The nodes send a change in detected motion immediately to the server. Additionally every 5 minutes, the current motion state is reported. On the server, this data is aggregated to a single data feed, which reports whether or not the lab is currently in use. This allows to provide long term reports on the daily usage of the smart office.

The final application is a window and light monitoring application. Each window is equipped with an ADC switch sensor, which detects when the window is open or closed. All the lights have a light sensor which detects when the lights are on or off. The ADC sensor is measured every 10 seconds. When a change is detected, this is sent towards the gateway, which then reports it to the server as shown in Figure 7.2c. Additionally the current state is reported every 5 minutes. On the server, the window and light state is logged. Additionally, when a window is open or the lights are on past a certain time each day, an e-mail is sent to the building administrator, who can then go and close the window or turn off the lights.

7.2.2 Security Requirements

This section goes over each application, and identifies the minimal security requirements for each application. For the security requirements, we will apply a basic premise that all data and commands which trigger actions either by actuators in the environment, or by notifications in the back-end, must be protected minimally with authenticity protection. All messages which can be used to identify persons in the lab must be kept confidential. The remainder of this section will go through the applications, and identify the security requirements for each message.

The environmental monitoring application has minimal security requirements. There is no actuation coupled with the environmental monitoring, and is currently only to provide an indication of the current office environment. As such, the security requirements of this data are minimal, so we use this data as a baseline set of applications with no security policies installed.

The motion application is clearly a more security sensitive application. It can potentially be used to detect illegal presence in the room, and as such should have at least authenticity protection, to ensure that an illegal visitor cannot spoof the motion messages in an attempt to hide his presence. Since motion data cannot be used to identify persons, the policy only requires us to send it authenticated, but in order to test all security policies, we apply a full data encryption policy to this application.

Finally, the window and light monitoring application can trigger emails in the back-end that notify the building administrator of a potential security threat, or unnecessary energy usage. As such the policy requires the data to be protected using authenticity protection.

7.2.3 Evaluation Goal

The goal of the evaluation of the smart office deployment is to evaluate the runtime overhead of the SecLooCI infrastructure on the embedded network infrastructure. As such this evaluation will only look at the following three data flows: 1) application communication, 2) application management, and 3) service usage. The network initialisation data flow and application deployment data flow are omitted, since they only happen in a very brief window at the start of the deployment. This evaluation will only look at the stable state. Each of the three observed data flows will be evaluated based on communication overhead, and processing overhead. Memory overhead of the platform is omitted since this already discussed in Chapter 6.

The secure application communication overhead is observed in two ways: the first aspect is the additional communication overhead due to security associations and security headers that need to be added for MACs and security information. The additional processing time measured is the time it takes to evaluate the security policies, and perform encryption, if necessary.

The secure application management overhead is observed as follows: 1) the network overhead due to data sent and received, 2) the processing overhead due to encrypting, decrypting and authorising the management events, 3) the processing overhead due to authorisation, and 4) the overhead of actually querying the necessary runtime information.

The monitored application service usage is observed as follows: 1) the communication overhead of sending the monitoring data to the back-end server, 2) the processing overhead for aggregating and sending the monitoring messages, and 3) the cost of calling the monitoring framework.

So to conclude, the smart office will evaluate the following three data flows in a continuously running live environment: 1) the application communication data flow, 2) the application management data flow, and 3) the service usage data flow. For each of these data flows, the applicable communication and processing overhead is observed.

7.2.4 Metrics

This section presents a list of observations made during the operation of the smart office deployment. It presents mainly the processing overhead of the applications. In addition it also list the communication overhead and memory overhead of the applications. The figures shown are the average hourly overhead of the different systems.

The hardware used for the evaluation is the AVR Zigduino [4], running Contiki 2.7 and LooCI v2, with the SecLooCI node security middleware. For authenticity, the system uses AES128-CMAC with 16 bytes MAC. For confidentiality, the setup uses AES128-CCM with an 8 byte MAC. Both algorithms are implemented in software. The CPU time is measured by the monitoring system, which measure the system ticks for performing certain operation. Each system tick is 64 usec, so this is about the maximum accuracy of the measurements. The readings are taken during several days, and then the hourly average of these readings is shown in the different tables.

This section first looks at the application overhead, and then the security overhead for these applications. Table 7.2 shows for each application the

amount of events sent per hour, the average size per event, the LooCI overhead, the security overhead, the number of CPU ticks the component used, the amount of RAM the component required, and the number of CPU ticks the application security component used to provide the required level of encryption.

The table shows several interesting observations. A first non-security related observation is that the temperature and humidity component consumes a lot of CPU, both compared to other components or compared to any security overhead. This is because the component which produces temperature and humidity communicates with the sensor using a custom serial protocol, which is similar to I2C. However, it is sufficiently different so that the nodes' I2C support cannot be used. The pressure component uses the I2C interface to the sensor, and as such requires significantly less processing. When compared with the window and the motion component: the window component uses active polling every 10 seconds to read the current value of the ADC. This is a fast process, but still takes some processing time. The motion component lastly operates based on a callback interface, and only gets warned when motion is detected.

Next this section looks at the overhead of the application communication security system. First, when looking at the communication overhead of application communication security system, we can see that authenticity requires some extra overhead: 16 bytes for the MAC, and another 7 bytes for various headers etc. Confidentiality requires slightly less overhead, since we use only an 8B MAC. However, the overhead is only 4 B lower because to ensure confidentiality the 4 B LooCI header is also encrypted.

Second, this section looks at the processing overhead of the application communication module. The setup shows that even for no security, a small amount of time is spent in the application communication module, about 58 microseconds an hour. This is likely due to some validation and policy checking statements, which then show that no security is needed. Ensuring authenticity requires ca 15.9 milliseconds an hour, or 1.33 ms per message. To ensure confidentiality requires about 19.1 milliseconds per hour, or ca 1.59 ms per message. The difference between authenticity and confidentiality is not large, due to the selected algorithms. AES128-CMAC also encrypts every block, but does not write the encrypted text back, which likely saves a few milliseconds. Using different algorithms can change these figures significantly.

For the second security evaluation, this section looks at the hourly overhead of the application manager. This shows the overhead of managing the applications and the properties. The monitoring system observes the introspection of the entire system, so we cannot attribute introspection to a single application. We have set up four different nodes with a different application setup to

Table 7.2: Hourly application usage.

Application name	Temp + Humid	Pressure	Window	Motion
Security requirement	No sec	No sec	Authenticity	Confidentiality
Used Ram (B)	26	24	38	43
Nr events sent hourly	24	12	12	14,3
Average packet size (B)	6	6	6	6
Average LooCI overhead (B)	4	4	4	4
Average Security overhead(B)	0	0	23	19
CPU time component (ms)	3773	151.3	63.5	4.07
CPU time security (ms)	0.060	0.030	15.76	22.2
CPU time security per msg (ms)	0.0025	0.0025	1.31	1.55

manage: node 1: two non-secure components (temperature-humidity and pressure component), node 2: 1 component with confidentiality (motion), node 3: 3 components of the same application which requires authenticity (window), node 4: 2 different application components, one requiring confidentiality (motion), and one requiring authenticity (light).

Table 7.3 shows the average hourly usage of the management component infrastructure. Note that the system validates the entire deployment every hour, so this shows the average overhead of a single cycle of application validations on the node. The tables show that the introspection requires a significant number of messages in all cases. The number of hourly messages of validation surpasses the number of functional messages. The average size of the messages is also slightly bigger, due to often requiring more content in introspection or reconfiguration. The time that the management component operates however is fairly similar the time the functional components operate.

Table 7.3 also shows that the time to encrypt the messages is similar to the time it took the confidentiality policies to be enforced at ca 1.71 ms per message. The slightly higher average time is likely because certain management messages are longer than a single block, requiring a second crypto block operation. Depending on the amount of long messages, the average time will be a bit longer. The second observation is that the average proxy overhead is only about 2 percent of the average cryptography overhead, showing that cryptography is definitely the largest burden. Hence this shows that a larger authorisation part, with multi-user support, can be added to these constrained embedded nodes with minimal additional processing overhead.

Thirdly we look at the monitoring overhead of the monitoring and enforcement system. Table 7.4 shows for each node the amount of components present on that node, the amount of logging messages sent per node, the average monitoring event size, and the average CPU time for the monitoring component. First, to explain the high number of components: each node has 4 default components:

Table 7.3: Hourly management usage.

Node	node 1	node 2	node 3	node 4
Nr Components	2	1	3	2
Nr Applications	2	1	1	2
Security requirement	no sec	conf	auth	conf-auth
Nr events received	18	14	28.2	25
Average received event size (B)	7	9.3	7.1	9,7
Nr events sent	18	14	28.2	25
Average sent event size (B)	11.5	9.9	9.3	9.88
Average LooCI overhead per msg (B)	9	9	9	9
Average Security overhead (B)	19	19	19	19
CPU management component (ms)	5.96	4.61	9.09	8.35
CPU management component per rec msg (ms)	0.331	0.330	0.323	0.334
CPU authentication (ms)	62.4	47.6	95.4	85.93
CPU authentication per msg (ms)	1.73	1,70	1.69	1,72
CPU authorisation (ms)	0.651	0.448	0.795	0.757
CPU authorisation per rec msg (ms)	0.0361	0.0321	0.0282	0.0303

one middleware management component, one security management component, one OS monitoring component, and one application monitoring component. Next to those components, each node has the application components installed as described previously. The table shows that the average event size is linearly related to the amount of components installed on the node. Each component requires an additional 13.5 bytes per message on average. The second observation is that the processing time of the monitoring component is fairly low compared to the crypto components, and lower than most application components. Note that we did not add any application security policies on the logging data, to keep the readings of the security framework purely focused on the application functionality.

When looking at the time required for the acquisition of the data, we observe that this is currently less than the time required to perform the reporting. Note that the time of data acquisition depends on the frequency of the monitored subject, so in this case the execution and publication occurrences of application components, while the average CPU for sending depends mostly on the frequency of reporting. However, these figures show that the monitoring infrastructure adds about 13 ticks of execution overhead per application component per hour. When compared to most application components, this is less than 10 percent of the application execution time. This does not include sending and receiving data though.

Finally we would like to frame the figures from the previous experiments with some general figures from the node middleware with regards to the network system, the event bus and the memory usage as shown in table 7.5. The first

Table 7.4: Monitoring overhead per hour.

node	Node 1	Node 2	Node 3	Node 4
Nr Components on node	6	5	7	6
Nr monitor events	4	4	4	4
Nr total events	76	46.25	96.3	82.8
Average mon event size (ms)	110.5	97	124	110.5
CPU data acquisition (ms)	1.89	0.539	2.47	1.98
CPU acquisition per event	0.0248	0.0116	0.0256	0.0239
CPU data publication (ms)	2.90	2.84	3.14	3.04

thing to point out is the very large receiving overhead. Currently the node uses active listening, and this clearly shows in the amount of receiving overhead. It is about 1000 times more compared to almost any other processing overhead. The second observation is that the CPU time spent sending is about 2 times the amount of time spent encrypting the various messages. So again this puts the overhead due to security into perspective.

Next we look at the time spent using the sensors. The current middleware monitors reading of ADC, using the I2C module of the Zigduino, and setting and getting of binary pins. The sensing time spent on node 1 is due to the pressure sensor, which uses the I2C module. This clearly shows the efficiency of the module compared to the manual serial implementation of the temperature and humidity component. Node 2 is a motion detection node, and uses almost no sensor time. Node 3 is the window sensing node, and while it samples the binary pin fairly frequently, polling a pin incurs virtually no overhead. Finally node 4 has a motion sampling and light sampling component. The sensor time incurred is therefore likely alone from the light sampling component, which uses the ADC converter on the node. Reading from ADC thus clearly requires some more time compared to reading a digital pin.

Finally we look at the LooCI communication overhead. The overhead of the event bus, which allows policy based event routing across the network based on event semantics and producer, is only about 2 percent of the actual time spent sending the events across the network. This shows that policy based routing is definitely viable energy wise. A related observation is that in the current system most of the event sending is clearly done by the LooCI networking bus. However, there is also a bit of transmission time of which we currently do not know the origin. It is suspected that this is IPv6 routing traffic, but this is not verified.

To summarize these observation, the smart office deployment shows with regards to security that: 1) the application communication security enforcement requires

Table 7.5: Operating system overhead.

Node	Node 1	Node 2	Node 3	Node 4
Hourly events sent	58	32.3	68.2	57.8
hourly events received	18	14.0	28,2	25
CPU sending (ms)	236.5	190.4	294.1	263.0
CPU receiving (ms)	67 385	63 409	105 449	87 490
CPU sensing (ms)	7.19	0	0	69.1
unused RAM (B)	4 974	5 269	5101	5 222
never used RAM (B)	3 979	4 326	4095	4 217
CPU LooCI event bus (ms)	3.95	2.19	4.18	4.53
CPU LooCI network layer (ms)	206.6	141.2	277.3	237.4

a significant amount of additional bytes to be sent, but since the packet sizes are small, it causes no packet fragmentation, so the actual impact is likely low, 2) the application communication security system requires processing time which is in many cases about the same time as the components require to process data. It is about a tenth of the time compared to the time spent sending, and even much less compared to the time spent receiving. 3) the application management subsystem which is used to verify the system every hour (so one 12th frequency) consumes more processing time due the large number of messages sent, and the fact that both sent and received messages require encryption operations. Hence the overhead of the continuous validation is significant. It also showed that the overhead of the authorisation module is only about 1 percent of the cryptography operations, so clearly multi-user interactions are feasible in these constrained embedded system, and 4) the total monitoring system in most cases uses less processing time than the applications, and sends less messages, but on average much larger messages, and 5) a fully operational system with the Contiki OS, LooCI and SecLooCI middleware still has ca 4K of unused RAM, which is about 25 percent of the total available RAM. This shows that the SecLooCI secure node middleware is able to operate on these constrained embedded nodes, while still providing sufficient memory for applications.

Note that the sampling frequency, monitoring frequency and application validation frequency are variable, and in this case chosen to produce a sufficient amount of certainty and visibility of the system. As such, most of these readings should be interpreted relatively to the sampling frequency. So by significantly lowering the validation frequency of the applications, the absolute overhead of the management system would be significantly lowered.

Also a few non security related observations: 1) active listening takes orders of magnitude more processing time compared to anything else, 2) custom

I2C implementations require significantly more energy compared to using I2C hardware support, and 3) reading ADC values also requires significant processing time, but less compared to custom I2C interfaces.

7.2.5 Observations

During the construction and operation of this smart office environment, we made two general observations which might prove interesting for related work: 1) while powered nodes do no tend to fail often, there are occurrences that cause nodes to reboot, and 2) to easily deploy embedded systems, nodes must run multiple applications.

All nodes in the smart office environment are connected to the power grid. This provides them with long term power, and causes them to almost never reboot. However, during one occurrence, the power of half the lab was accidentally shut down. This event was quickly detected by the application owner server, since suddenly a large number of nodes was no longer responding to requests. The application owner was notified, and took action to attempt to solve the issue. Eventually the issue turned out to be a fuse which was automatically turned off. This issue was easily fixed, and the application owner server automatically reinstalled the necessary components. While a minor issue, this event showed two things: 1) nodes will reboot on unforeseen times, and 2) by having a monitoring and management infrastructure, these events are noticed faster due to automated reporting, and fixed easier by automatic state reenactment.

The second observation is related to the application running on the nodes. To easily set up an office deployment, you want to quickly put a few nodes on certain places, and instrument them with the necessary sensors. In many cases, the nodes were instrumented with multiple pieces of hardware, and thus running multiple pieces of software to support that hardware. For example one node was involved in both the temperature monitoring and the motion detection application. An other node was involved in the motion and light monitoring application. The motion and temperature monitoring applications are nice examples of applications that can use multiple nodes, and instead of having to set up individual nodes for each of these applications, reusing nodes allows for quicker and easier setup.

7.3 Summary

This chapter has presented the evaluation of the SecLooCI infrastructure and SecLooCI node security framework. First it presented a virtual test scenario based on the smart logistics use case which went through the lifecycle, presented how the SecLooCI infrastructure operates, and identified the overhead of the different phases of the application deployment and operation. This showed that the overhead of secure dynamic course grained evolution of embedded networks is similar to about one day of application execution.

Second, this chapter provided an in-depth evaluation of a real world smart office deployment. Several embedded applications have been deployed in a smart office testbed, to monitor environment variables such as temperature, humidity, window state, and lights. This deployment showed that the SecLooCI node middleware does pose some overhead, but that it is in the same order of magnitude as the execution time of certain applications. It also showed that most of the processing time of a node is spent receiving, and only a fraction actually doing other activities such as security. Finally it showed that active verification of embedded systems by frequently inspecting the current state is quite costly, but active monitoring of running applications only has a limited overhead compared to most application functionality.

Chapter 8

Discussion

This chapter discusses the proposed SecLooCI security infrastructure for Shared Networked Embedded Systems. First this section goes over the functional and non-functional requirements and details that all requirements have been met. Next this chapter discusses how the architecture prevents the different attacks. Finally this chapter discusses the trust requirements, privacy impact, energy impact, features and trade-offs of the security infrastructure, and the framework approach to designing the infrastructure.

8.1 Non Functional Requirements Discussion

This section looks at how the non functional requirements, as presented in Section 2.5.2 have been fulfilled.

8.1.1 Evolvability of Infrastructure

The SecLooCI infrastructure handles the evolvability of the underlying infrastructure during the different stages: it handles the joining of new nodes, changing connected hardware of nodes, and the disappearance of nodes, depending on the requirements and exact scenario.

Node joining is handled by the Secure Network Initialisation data flow. When a new node joins the network, it must contact the network gateway. The network owner can then negotiate policies with the platform owner and allow the node

to security join the network, as discussed in Section 6.2. Once this is done, the network owner can add the new node to its node repository, and potentially inform interested clients that a new node has joined.

The infrastructure also handles adding new hardware, be it adding nodes to the system, or hardware to nodes. Adding nodes to the system simply requires the platform owner to add a description of the nodes, together with the initially installed hardware to the platform owner server. Potentially node vendors can even provide a specification document, which the platform owner simply has to import in his server. Adding new hardware requires the platform owner to add the necessary resources and parameters of that hardware to the node model. Again this can potentially be automated by the vendors providing the necessary specs, which can be automatically loaded in the end-user tool, and deployed on the platform owner server.

The disappearance of nodes can also be handled by the infrastructure. During the application monitoring, the application owner server can notice that a node that it expects to be present is no longer available. Depending on the requirements of the user, the application owner server can then notify the application owner to update the application, or it can automatically select another node to deploy the necessary functionality based on application requirements. Currently only the notification is available in the SecLooCI infrastructure, but related work [49] has shown that it is possible to adapt deployment based on high level application requirements.

8.1.2 Evolvability of Software

The SecLooCI infrastructure supports the evolvability of software both in the back-end and on the node middleware, for both creating and securely deploying new applications, or securely modifying and removing existing applications.

The end-user tool enables users to easily create new application deployments based on pre-existing generic application models, and to create their own new applications. It allows them to easily add security requirements to their applications based on the semantics of the data transmitted, and to set the functional parameters of their applications, such as limits or sampling frequency. These specifications are then sent to the application owner server, which automatically rolls out these applications on to the relevant embedded nodes. By using the secure application deployment system, it can securely deploy new application components when required on these embedded nodes.

The end-user tool also provides users with an easy way to change their security or functional requirements by simply downloading the current deployment

specification, altering the required parameters, and commit the new specification back to the application owner server. The application owner server will then automatically attempt to change the underlying system to achieve the new desired state, and it will automatically clean up any policies or settings that are no longer necessary. The application owner server can securely enact these changed requirements on the embedded nodes using the Secure Application Management System. Additionally when removing the application, the application owner server can again use the Secure Application Management system to clean up all obsolete components, configurations and policies.

8.1.3 Heterogeneity of Infrastructure

The SecLooCI infrastructure is able to support nodes with a wide variety of sensing and actuation capabilities, or node types. The platform owner node model allows the platform owners to encode any resource that is available. The tool provides a subset of standard available resources, but the model is not restricted to only using these resources. platform owners can append any resource they feel is necessary.

The node model can similarly support multiple different node types. The current model provides a limited list of available node types. However, again the platform owner can add any new node type that he wants. However for both the node types and available resources, care must be taken that the provided resources as modeled by the platform owner, match the required resources as modeled in the application by the application owner. This could potentially be solved by a global repository of node types, sensing and actuation resources. This has however currently not been implemented.

The end-user tool aids the application owner to deploy components easily on such heterogeneous infrastructure. Based on the available implementations, required resources of those implementations, the available nodes and the provided resources of those nodes, the tool can filter out all nodes which are unable to support a certain application. This allows the tool to filter the potentially large set of available nodes so the application owner can select the preferred nodes from a list of only suitable nodes.

8.1.4 Transparency of Heterogeneity

The SecLooCI infrastructure handles the heterogeneity of performance by providing the same modelling abstractions for describing both resource constrained nodes as well as resource rich nodes. Modelling and assigning

similar behaviour is identical for resource constrained nodes and resource rich nodes.

By using the same modelling abstractions, the user uses the same set of abstractions to declare that two resource constrained nodes are exchanging data, or two resource rich nodes, or a resource constrained node is sending data to a resource rich node. The user does not have to be aware of the difference between the nodes, nor does he have to use multiple different abstractions to interact with different types of nodes.

The SecLooCI infrastructure also provides users with a single set of security abstractions to model the security requirements, which is independent of the underlying platforms. Users do not have to distinguish between which platform is running underneath, or which type of communication pattern is used by the application. Rather the user can express their security requirements based on the semantic type of the data that is being transmitted.

8.1.5 Transparency of Security

The SecLooCI framework handles the security transparently for the application on multiple different levels. First, the end-user tool encodes the security requirements in a unique security policy document, that exists next to the actual application deployment model.

The node security middleware also handles the security transparently for the actual application implementation on the node itself. As long as the implementations of the application behaviour follows the best practices of of the application middleware, such as clearly and uniquely encoding the semantic type of the data transmitted, and using the monitored proxy services for accessing node OS services, the middleware can transparently secure the messages transmitted using the Secure Application Communication System, and transparently monitor the service usage of the application using the Application Monitoring System.

8.1.6 Flexibility of Communication

Applications can have multiple different modes of communication, such as one-to-one, one-to-many, and opportunistic communication. The SecLooCI infrastructure allows all these communication patterns. The application owner can encode which type of communication should be used in his application using the Application Modelling abstractions. Additionally the application owner must then specify the security policy based on the type of data. These policies

are then automatically parsed and enacted by the application owner server and installed as policies in the Secure Application Communication system on the embedded nodes.

The Secure Application Communication system on the nodes has a very flexible security mechanism. Based on outgoing encryption policies, the data transmitted by the node is sent if necessary over a secure channel which either provides authenticated or confidential transmission. Incoming encrypted data is then checked based on the channel identification, and the necessary decryption and validation is performed. Additionally incoming security policies can ensure that for certain types of data, certain security guarantees are met. Using the flexible security policies provides the users with a very flexible model able to secure most if not all communication patterns.

8.1.7 Flexibility of Security

Since security incurs a cost, it should be possible to decide which kind of data must be protected with which kind of security protocol. This is provided by the application security policy modelling abstractions, and enforced by the application communication security middleware.

The application security policy model allows users to declare which types of data should be secured with which level of security. For example users can declare that environment monitoring data does not have to be secured, motion detection data must be authenticated, and personally identifiable data must be encrypted. This can be done on a per application, and a per user level. So different users can decide for themselves how their application communication should be secured. To provide simple abstractions, users can only decide between no security, authenticity, or confidentiality on this level.

The policies encoded in the application security policies are then enforced by the SecLooCI node security middleware. The node security middleware also supports even more fine-grained configuration of the level of security, by allowing the increase or decrease of the length of the MACs. This potentially reduces the amount of communication overhead, however the amount of processing overhead remains the same, since in most proposed algorithms, sending a smaller MAC still requires the calculation of the full length MAC, but the long MAC is just truncated to the first bytes.

The node security framework also supports multiple different algorithms. This could allow certain users to use lighter authentication or encryption algorithms, or experiment with newer version which reduce the overhead on the embedded node. These algorithms can be deployed as necessary on the sensor nodes, which

would enable algorithm negotiation and ad-hoc deployment. However, to keep the user abstractions simple, the length of the MAC and the algorithm selection are made by default by the SecLooCI server infrastructure.

8.2 Security Discussion

This section presents the security analysis by identifying the possible attacks and possible mitigation options of the four different types of attackers as presented in Section 2.5.1: the outside attacker (OA), the Network Attacker (NA), the physical attacker (PA), and the insider attacker (IA). To identify the possible attacks, we apply the STRIDE threat model [112] to these attacker and assign the different threats to the relevant actors. This approach is an industry standard proposed by Microsoft, and recommended by OWASP [86], and the Open Web Access Security, a consortium to promote the security of the web, with members such as HP, IBM, and Oracle. OWASP recommends the STRIDE model because it works well for addressing the unique challenges facing web application security and is simple to learn and adopt by designers, developers, code reviewers and quality assurance.

The list of threats we have identified is: spoofing identity (OA + NA + PA), tampering with data (OA + NA + PA), repudiation (IA), information disclosure (OA + NA + PA + IA), denial of service (OA + NA), and elevation of privilege (IA). The remainder of this sections discusses each of the types of attackers, explains how they pose a threat to the system, and how the SecLooCI security infrastructure is able to mitigate the risk.

8.2.1 Outside Attacker

The outside attacker is an attacker who is not part of the network, and who can listen to the network, intercept messages, and change messages, but not break cryptographic protocols, according to the Dolev Yao model [30]. He can potentially 1) spoof the identity of users, 2) tamper with application or management data, 3) disclose information, and 4) perform denial of service attacks. The SecLooCI middleware prevents the first three attacks, and offers potential mitigation against the fourth attack.

The secure management system, and the secure application communication system prevent network attackers from spoofing the identity of nodes and users. All management communication is encrypted using a symmetric key cryptography protocol, where the secret key is only known to the nodes and

users. Since the model dictates that a NA cannot break encryption algorithms, the NA cannot perform identity spoofing. The secure network layer provides full network authenticity and encryption protection, which prevents the NA from tampering with all application or management data. The secure network layer also prevents any data disclosure for parties outside of the network. Finally the SecLooCI infrastructure cannot prevent denial of service attacks.

SecLooCI does have tools that can potentially detect such attacks when they occur. For example the application owner server validates that nodes can be reached, and that messages can be received. The application owner server is a final endpoint of data, and can validate that data is coming in when expected. Additionally the platform owner server should receive monitoring data from his nodes. When either of these mechanisms fail, the relevant parties can investigate what is interfering with their communication, and take action. From a node point of view, the monitoring and enforcement system can detect when a significant amount of invalid messages arrive. In such cases, it can go into an energy saving mode by for example turning the network stack off for some time.

8.2.2 Network Attacker

The network attacker is an attacker who is part of the network of the network owner, but who has no permissions on the node system, nor to access any data produced by the node. As such he can perform all the attacks the OA can, and has extra capabilities to do so. As the outside attacker he can: 1) spoof the identity of users, 2) tamper with application and management data, 3) disclose information, and 4) perform denial of service attacks.

As with the outside attackers, the secure management system prevent the NA from spoofing users. Since all management operations are authenticated and encrypted end-to-end and the network attacker does not receive these credentials, he cannot perform such attacks. He also cannot tamper nor disclose this management data.

A network attacker however can tamper with application data if the application owner does not require end-to-end security for his application data, and disclose application information if the application owner has not required an end-to-end confidentiality for his data. In such cases, the application owner permits other members in the network to read the data, and in case of no security policy, even to potentially modify or spoof the data. This is a choice the application owner has to make. However, if the application owner has high security requirements for certain types of data, the SecLooCI infrastructure allows him to specify these policies, and enforces them by applying an additional layer of end-to-end encryption.

Finally the network attacker can also perform Denial of Service attacks, and can do these more efficiently, and can perform other attacks, such as the sinkhole attack. In this kind of attack, a node will fake being the best next hop for all nodes, and when routes are established, not forward the traffic, denying service to the nodes. Again, the SecLooCI infrastructure has no countermeasures for such attacks, but as with the OA, it has the necessary tools for detection. We also consider routing security out of scope for this thesis, and refer to related work where ample countermeasures have been proposed for this and other attacks.

8.2.3 Physical Attacker

The physical attacker can physically probe a node, and as such retrieve all node credentials and key material that is contained within the node. When such a PA performs a successful probe, all network and application security keys will be available for the PA. He can thus perform the following attacks: 1) spoofing identity, 2) tampering with data, and 3) disclose information.

The physical attacker can try to spoof the identity of the owner of the node based on the owner key of the node. However, the SecLooCI node middleware requires that each node has a unique secret key shared with the platform owner. As such the PA cannot spoof the identity of the owner. Any user key that is present on the node is also disclosed to the PA. The SecLooCI middleware prefers that each user key is unique, however it does not strictly require it, because by allowing a single user key to be installed on multiple nodes, the user could perhaps perform group management commands, which save significant resources. When a single user key is used on multiple nodes, this can be disclosed and all other nodes are temporarily vulnerable. However, when the attack is detected, the platform owners can use their owner key to remove compromised users, and install new keys.

The physical attacker can tamper with and disclose information of those data streams in which the disclosed node participated. Since the disclosed node was a valid participant in the node communication, he needed to be able to create and read traffic, and as such when disclosed the attacker can do the same. However the Application Communication system only discloses keys to those parties that need it, so not all data streams in the network are affected. While the SecLooCI infrastructure currently does not attempt to detect these breaches, it can recover from this disclosure. Again the platform owner or application owner of the other affected nodes can use their unique user keys to update the cryptographic material on the nodes, and exclude the disclosed node. Since the

PA cannot read the rekeying messages, the new keys are not disclosed and the system can be returned to a safe state.

8.2.4 Internal Attacker

The inside attacker is a network owner or application owner who is trusted by the platform owner of the attacked node. As such the IA has valid credentials to contact the node, and perform certain management operations on the node. This enables him to do the following attacks: 1) repudiation, and 2) elevation of privilege.

The secure application management system ensure non-repudiation since the IA cannot deny performing an operation. Each user receives unique keys for the nodes of the platform owner. As such, the key is only known to the platform owner, the node and the IA. When performing an operation, the IA cannot deny having requested that operation to the platform owner. The secure application management system also prevents elevation of privilege. The management system has an access control list which specifies which users are allowed which operations. As such, the IA can only perform those operations which he is allowed to perform.

8.2.5 Other Threats

Recent revelations have shown that governments and other organisation have the capabilities to and actively engage in spying on the communications of virtually everyone on earth, actively attack network providers and add spyware into the hardware that is invisible for the software running on top. These actors form an attacker that has almost unlimited resources and unknown capabilities. However, we can look at these attackers based on the previous attacker model.

When these attackers target the hardware, we can assume that they have full control of the hardware, and thus basically act as a physical attacker. We stated previously that the software framework can do very little to prevent this kind of attacks, but if it can be detected, the framework can potentially mitigate the attack and remove the broken nodes from use.

When these attackers target the network, we can assume they have full control of the network, but still can not break cryptographic primitives, similar to the network attacker. As such, the framework can offer certain protection against these kinds of attacks: they cannot spoof users, nor can they perform management actions on the platform. Users can also still send confidential data over the network, which can not be read by these attackers. The network

attacker can however still see when communication takes places, and potentially with whom the devices are communicating. However solving these issues is beyond the scope of this thesis.

8.2.6 Summary

The SecLooCI security infrastructure prevents many possible attacks from outside, network, physical and inside attackers. While most of the attacks are prevented by the security middleware, the framework cannot prevent certain kinds of attacks such as network based attacks and physical attacks. However, the SecLooCI infrastructure does provide the necessary tools to potentially detect and mitigate these attacks.

8.3 General Discussion

This section provides a general discussion of the SecLooCI infrastructure for the secure sharing of embedded networks. First it looks at the trust requirements of the infrastructure, and identifies the different ways trust can be required and transferred in the system. Second it discusses the privacy implications of the framework. Next it considers the energy consumption of the framework. Then, it looks at the current limitations of the infrastructure, and lists some of the trade-offs made in the construction of the prototype. Last it discusses why the system fits the definition of a framework.

8.3.1 Trust Requirements

The SecLooCI infrastructure enables multiple different parties to securely cooperate in embedded networks. As such, the primary requirement to cooperate is that the different parties trust each other enough so they are willing to cooperate. Currently, the system assumes that network owners, application owners and platform owners will only offer their services to parties they trust, and only use services from parties they trust.

Currently the prototype requires that each network owner, platform owner and application owner explicitly lists those parties with whom it is willing to collaborate, and is able to retrieve valid certificates for those parties. However, this is mostly an implementation issue. In a mature environment, trust will likely be established by third party trust providers, which will likely also operate as certificate authorities. Each party must then register with one or more of

these trust providers, and trust the trust provider to only permit trustworthy parties. This significantly reduces the burden of establishing trust relationships, and opens up the system for ubiquitous usage in a very scalable way. These third parties could also act as payment aggregator, paying platform owners for all associated usage, and billing application owners for said usage. Parties still can chose which services to share with which parties or groups, and distinguish in offered services between different trust providers. The SecLooCI infrastructure currently already allows parties to differentiate available resources based on groups.

At the moment the system requires an explicit trust relations to exist between the parties, however this is optional. For example, a network owner might be willing to offer his network to all users, but with severe limitations, or a platform owner might be willing to open up certain nodes to everyone, since they provide very limited value to him. In such cases no trust relation would be required. This is currently not implemented, but could potentially be a part of a mature system. However when dealing with critical systems, the parties involved will want strong security guarantees. For example in the smart logistics scenario, it is critical that only certain users are allowed to access for example the current container manifest, or the container lock service.

8.3.2 Privacy

Privacy has gained a lot of attention in recent years. While the definition of privacy is difficult and many people define it differently, this thesis will take the definition provided by Westin et al. [120]: ”the right of the individual to decide what information about himself should be communicated to others and under what circumstances”. With the arrival of more pervasive means to track individuals, it is clear that in the context of networked embedded systems, privacy is a valid concern. While the provided framework does not provide full privacy, it can aid in attaining more privacy.

The framework allows owners of the platforms to decide which resources, and which data they share with whom. As such, the framework is able to restrict access to information resources only to trusted parties. It also allows users to specify which data must be communicated confidential. This ensures that no other parties in the network can read potentially identifiable data, such as location data or personal identification codes, during transmission.

However, the framework only provides a small piece of privacy puzzle. There are still many issues. The first issue is that all devices are currently identifiable when they are present in a network, due to their unique and persistent MAC address. This allows attackers to potentially track devices, and the users or goods they

are attached too. Second, due to the strong dependency on interaction with a cloud infrastructure, each device must know the identity of the owner, and transmits this identity when connecting to a new network, which can be a privacy problem. These problems can be mitigated: the first problem can partially be mitigated by MAC address cycling, where on joining a new gateway, or after a certain time frame, the node changes its MAC address [42]. The second problem can be partially mitigated by using anonymous connections such as TOR anonymous services [29], or allowing a trusted third party to manage the nodes of multiple parties, which potentially brings its own privacy problems.

Next to these issues, there are several others. One of the main issues is the question of privacy when the devices of other people are monitoring the behaviour of and potentially identifying other users. While this is a valid privacy question, answering such questions could potentially fill another dissertation. As such this and other privacy questions are considered out of scope.

8.3.3 Energy

In wireless battery powered embedded environments, energy is a crucial concern. As such the cost of encrypting and monitoring the system should be carefully considered. The remainder of this section discuses first why we did not do in depth energy measurements, and second discusses the energy concerns of the different security operations based on the smart office usage case: 1) encryption overhead, 2) authorisation, 3) application validation, and 4) node application monitoring.

First, during this thesis, there were no in depth energy measurements done of the SecLooCI middleware for the following reasons. First, any energy measurement is very dependent on the used hardware system, so even small differences of a node's hardware composition might have a large influences on the energy consumed per action. Second, doing energy measurements is typically done using a special energy measurement setup, which often suffers from limited visibility of what actually occurs on the node, and is very labor intensive to achieve significant depth of results. Third, in order to create a reasonable and insightful study, significant work has to be done to create baseline measurements of the non-secure middleware, to frame the results. And last, monitoring results have shown in related work to fairly accurately reflect energy usage, and has the advantages of being able to be done in a real-world setting, and offering much more detail in the results [32]. As such this thesis only performed a resource requirements evaluation based on the monitored smart office deployment, instead of probing the system in an energy measurement setup.

The smart office deployment currently already shows that adding security policies requires some additional processing and network overhead, but that this overhead is significantly lower compared to the total time the system is listening to the network. This matches with findings of De Meulenaer et al. [27], who found that of all the processing done on a embedded network, only ca 5% of time is spent doing security operations compared to network operations, at least when using symmetric cryptography. Our findings corroborate that. Special note should be made that currently, all encryption protocols are executed in software. By changing them to hardware, the overhead of encryption can be significantly reduced.

Second the smart office deployment shows that the authorisation overhead of the secure management system is much smaller compared to the time spent performing the encryption and decryption, only about 1%. The authorisation overhead can be considered one of the main differentiaters between a single user and a multi-user system. In a single user system, the main user still needs to ensure security, and encrypt all management requests. The difference is that the user is by default authorised to all functions. Showing that the current authorisation system can be implementated with minimal runtime, and thus energy overhead, greatly supports the hypothesis that these devices can be shared.

Third, the smart office deployment clearly shows that active validation of applications requires a large amount of resources. The current smart office deployment validates each hour that all configurations and applications are still running as required. This involves sending a large amount of introspection commands. Due to the high frequency of validation, the overhead of these commands is in most cases higher than the time spent on the application itself. This shows that active validation is likely not ideal for energy constrained environments, and other means of validating deployments should be investigated, such as outcome monitoring.

Finally, the smart office deployment shows that the overhead of monitoring the node is small compared to the even the functional behaviour. Currently node monitoring data is sent to the platform owner every 30 minutes, in the form of 2 messages: one for node monitoring, and one for application monitoring. Applications currently trigger every 5 min. This means that 2 messages are sent for every 6 of an application. However, sharing nodes significantly increases this number. For example the window node monitors 3 windows, and as such sends 18 messages every 30 min, which means the monitoring only incurs a 10 % overhead in sending. Naturally in more energy constrained environments, the monitoring frequency can be reduced signficantly to reduce overhead.

8.3.4 Limitations and Trade-offs

The creation of the SecLooCI infrastructure, the SecLooCI node middleware architecture and prototype were significantly constrained due to our goal of implementing and designing the system for constrained embedded nodes. During the creation of the prototype, the biggest constraint experienced was no doubt the memory limitations of both RAM and ROM.

All security elements, from userId, to permissions, party information, security policies etc. all require some amount of memory. The Zigduino sensor node has only 128 KB of ROM memory, and a mere 16 KB of RAM memory, which is very low to current standards. The current implementation still has about 4 Kb of unused RAM with a limited amount of users and policies installed. However, this causes the node to only be able to support about 20 concurrent users and parties per node, with about a dozen components, each having a few resource consumption policies. The ROM memory is equally an issue. Currently only about 36Kb of the 128Kb is available, which is 28 percent of the available space.

These limitations did have a significant impact on the design decisions of the SecLooCI node middleware. The two largest design decisions made were the choice not to use asymmetric cryptography for these nodes, and the choice of having a hard coded role based access control infrastructure.

To support the resource constrained hardware, we decided not to use asymmetric cryptography. Most current generation asymmetric cryptography implementations both require a significant amount of memory for storing the credentials, require a significant amount of time to validate and create signatures, and still require the necessary infrastructure to update and change certificates. For example RSA signatures require 128 bytes for the certificate at least, and 128 bytes for the signature. ECC requires less transmission and has smaller certificates for the same security level, namely about 20 bytes. However, to perform asymmetric cryptography still takes a significant amount of time. An optimised ECC algorithm [77] still requires 55 seconds to verify signatures, which is a significant energy drain on nodes. While solely using symmetric key cryptography sacrifices forward security if the key is broken, as analysed by O'Hanlon et al [83], it does reduce the overhead on the node and network.

The second trade-off is the way the current prototype implements permissions. Currently each service verifies that the user has a certain role, either as a global user, or as a user of the party offering the service. These requirements are hard coded in the offered services, and cannot be changed. This allows for efficient storage and low runtime overhead, but reduces the access control policy options.

Note that none of these limitations are fundamental to the architecture, but

rather are trade-offs made to implement the architecture on the target resource constrained embedded devices. The architecture could potentially be scaled upwards in favor of more expressive policies and more flexibility on the node, and perhaps easier management of node security credentials using asymmetric cryptography. However the current prototype shows that the target resource constrained devices can offer the necessary services to enable secure shared usage.

8.3.5 Security Framework

This dissertation presented a software security framework. We decided to build a framework, because it provides an flexible system which expert users can extend, but still offers a secure and complete environment for end-users to use. Because the framework decides how to handle network or client request, users don't have to burden themselves with specifying and implementing this behaviour. The users of the framework can modify how these operations are handled by adding policies, or other user specified code, but the framework dictates the standard way of handling the data flow.

The framework also provides a set of clear user extension points, where users of the framework can add their own customized code or policies, while still operating withing the security framework. Clear examples of this approach are 1) the secure application management system, where users can add access control rules, policies and roles to suit their specific use case, 2) the secure application communication system, where users can add new cryptographic algorithms or protocols to secure their data within the flow of the framework, 3) the monitoring and enforcement system, where users can add additional interception points and monitoring limitations to monitor arbitrary functionality, and 4) the meta-data provided with the resource and parameter modelling abstractions, which offer key value pair descriptors to arbitrary extend the given functionality.

The framework also offers a level of flexibility. Users can optionally chose to add or remove systems from the larger infrastructure, either omitting them to reduce size and overhead, or decide to change the implementation to optimize to a specific application. The security framework provides a clear set of interfaces, and an architectural overview which has to be adhered to, but specific subsystems can be reimplemented, allowing for greater customisation.

This approach also provides a level of crypto-agility: the secure application communication system allows users to deploy their own crypto algorithms, replacing the previously installed algorithms. Currently this approach is only implemented in the secure application communication system, because here we see there could be more value in changing cryptographic algorithms. In

future, it could also prove to be necessary to change the used cryptographic algorithms in the other security protocols. While possible, this would require a small addition to the protocols or management infrastructure to take the used protocol into account. The decision to not support algorithm evolution was made to maximally reduce overhead of the protocols. Since almost no other embedded security protocols provide this feature, adding it would prove an additional overhead, making comparison harder. The framework does have the big advantage that the secure application deployment system allows the deployment of new code modules, making future code evolution already possible.

8.4 Summary

This section provided a discussion of the SecLooCI secure shared embedded network infrastructure. Section 8.1 showed that the requirements as identified in Section 2.5.2 are met. Section 8.2 went over the four kinds of attackers identified in Section 2.5.1: the outside attacker, the network attacker, the physical attacker, and the inside attacker. It showed that the current SecLooCI infrastructure prevents many attacks by these different kinds of attackers, and that for those attacks that it cannot prevent, it does have the necessary tools and infrastructure to detect the attack, and to mitigate its effects. Finally Section 8.3 provided a general discussion about the architecture with regards to the trust requirements of the infrastructure, the privacy implications, the energy consumption, the current trade-offs and limitations of the system, and the rationale behind building a framework.

Chapter 9

Conclusion

This chapter presents the conclusions. First it presents a global summary of this thesis and emphasizes the contributions. Second it formulates the most important lessons learned during the thesis. Third it identifies two avenues of future work, and finally it provides an outlook for network embedded systems in general, and security in particular.

9.1 Summary and Contributions

The goal of this thesis is to enable the secure sharing of constrained networked embedded systems. The need for such a system comes from the observation that embedded networks have evolved and still are evolving towards multi-party, multi-application and dynamic environments. Most use cases first and foremost show that multiple parties are interested in the data produced within the network, and that often, the network consists of multiple devices owned by multiple different parties. These networks typically are not used for a single application, but rather run multiple applications that gather data and offer actuation for multiple different parties. Lastly, there is a large amount of dynamism in these networks both in hardware and software: new devices and applications are continuously joining the network, existing applications are updated, and old applications and nodes are removed from the network. To ensure the continuous operation of these shared networks, it is crucial that the necessary security infrastructure is present. This need is even more pressing considering the pervasive nature of the nodes in the ecosystem, the sensitivity of the data measured, and the impact of the actuators on the environment.

To gain a clear understanding of the problem, Chapter 2 first listed two prototypical use cases: 1) the smart office use case where a building operator wants to monitor the offices, and offer smart services to the users of the office space, and 2) the smart logistics use case, where logistics providers instrument their containers to gain greater visibility of the transport chain, and offer this data and node services to different actors in the logistics ecosystem. Next, from these two use cases, Chapter 2 identified the three different roles a party can take on in these embedded networks: 1) the Platform Owner, who owns the embedded nodes, and wants to share their node services with other parties, 2) the Network Owner, who provides a wireless network for migrating nodes of the platform owner, and 3) the Application Owner, who wants to use the node services of other parties to reduce the necessary investment of getting data. Then, based on the use cases and the role model, Chapter 2 provided an overview of a prototypical embedded network application lifecycle. Finally, it identified the two main issues to currently enable this work: 1) the different parties need to be able to declare their policies and applications, and 2) these policies must be enforced at a node local layer by a node security infrastructure.

Chapter 3 provided an overview of the current state-of-the-art solutions for 1) abstractions for expressing the security and application requirements of the different parties as identified in Chapter 2, and 2) current solutions for providing the necessary node security systems during the entire lifecycle of the embedded application. The survey showed that currently little work exist that adequately provides the necessary abstractions to model the applications, combined with the semantic data flows, and underlying systems. The lack of such models also prevents expressing the necessary security policies, which either need to secure the data flows based on data semantics, or to restrict usage of the underlying system. Second, an overview of the current related work with regards to embedded network security solutions showed that there are currently many security subsystems, that provide some limited security for a certain limited part of the lifecycle. However, most solutions focus on only very specific security problems, and those solutions targeting very constrained environment only consider single party environments, and most solutions targeting multi-user environments require resource rich environments. Especially, currently no one has proposed a single comprehensive node security framework, able to run on resource constrained devices, and that support shared environments.

Chapter 4 provided the top level architecture of the SecLooCI secure embedded network infrastructure. It proposed the 5 top level architectural elements in the shared embedded ecosystem: 1) the end-user tool, which offers an interface for users to create and model their embedded applications, and to express their security requirements for their node infrastructure, and their application data, 2) the application owner server, which manages and monitors active applications

and stores long term security policies and data, 3) the platform owner server, which provides information about and permissions for node usage, and monitors the platform owners' sensor nodes, 4) the network owner gateway, which stores network owner security policies, and offers nodes an entry point to join the network, and access the Internet, and 5) the embedded node, which offers shared services for different parties in the network to enact their application functionality. Second, Chapter 4 also provided an overview of a basic non-secure node architecture, and the security modules required to provide the necessary security infrastructure.

Chapter 5 presented the first contribution of this thesis: a set of security management abstractions that allow the different stakeholders to express their security requirements, and a set of supporting abstractions which model the system and applications. The abstractions for the platform owner are: 1) the node abstractions, which specify for each node the present and available resources, and the limits and cost of using those resources, 2) the application owner abstractions, which encode the different application owners or groups the platform owner trusts, which nodes and resources he allows them to use, and with which specific limits and cost, and 3) the network owner abstractions, which describe which network owners the platform owner trusts, and which network cost requirements the platform owner has for networking. Second, the chapter presented the abstractions for the application owner which are: 1) an application specification model which describes the required resources and the semantics of the data produced and consumed of each application, and 2) a deployment specification model, which instantiates an application into a specific deployment, with an associated set of data security requirements. Finally, Chapter 5 presented the network owner abstractions, which express which nodes are currently present in the network, which zones are present in the network, and which parties are allowed to use the network, at which cost and with which zone permissions.

Chapter 6 presented the second contribution of this thesis: a holistic node security middleware for resource constrained embedded nodes, which provides security for the full lifecycle for a networked embedded application in a shared embedded network. The SecLooCI node security middleware is comprised of the following subsystems: 1) the secure network setup system, which allows embedded nodes to securely set up a shared embedded network, 2) the secure application deployment system, which allows multiple different parties to securely deploy new application functionality on the shared embedded nodes, 3) the secure application management system, which allows different users to manage their applications, their security policies and inspect the underlying node platform, 4) the secure application communication system, which provides policy driven secure application communication, based on the

application semantics and the security requirements of the different parties, and 5) the monitoring and enforcement system, which monitors the node usage of the different applications and users. These systems were integrated and evaluated in a single prototype, which showed that current generation embedded nodes can support the necessary security systems to enable secure sharing.

Chapter 7 evaluated and validated the SecLooCI security infrastructure based on two scenarios derived from the two use cases presented in Chapter 2: the smart logistics scenario, and the smart office scenario. The smart logistics test scenario provided a virtual evaluation scenario which showcased the entire embedded application lifecycle and provided an overview of the overhead of the SecLooCI security node middleware. It provided a theoretical evaluation which lists for each relevant step in the embedded lifecycle how the SecLooCI architecture operates, and what the network and node overhead is. Second, the smart office scenario provided an evaluation based on a real world deployment. It looked at the overhead incurred by the security middleware during the runtime phase, more specifically at the overhead of the secure management system, the secure application communication system, and the monitoring system. These figures showed that for this small testbed, the incurred overhead is significant, but manageable, again showing the validity of the proposed approach.

Finally Chapter 8 provided an in-depth discussion of the security middleware. First it went over all the non-functional requirements as proposed in Chapter 3, and detailed how the SecLooCI infrastructure met all the requirements. Second it provided a security analysis of the SecLooCI infrastructure: it applied a threat model to the embedded network environment based on the four main potential adversaries. The threat model identified how the SecLooCI infrastructure prevented most of the possible attacks, and if unable to prevent an attack, how it could be used to detect and mitigate the attack. Finally Chapter 8 provided a general discussion of the SecLooCI infrastructure, where it looked at the trust requirements of the environment, privacy concerns of the environment, the energy requirement of the security middleware, the most important restrictions and trade-offs made in the architecture, and rationale behind building a framework.

9.2 Lessons Learned

The first important lesson learned is that building a flexible policy driven security environment, in the form of the end-user tool and the security abstractions, depends in the first place on a good model of the underlying environment. Without having a clear model of the applications, the resources they require, the nodes, the resources they provide, and the data that is produced, it becomes very

difficult to express any policy which aims at securing data based on semantics, or to restrict and/or predict the application usage. When starting to work on the security policy part of this thesis, we looked at the possible application modelling techniques for embedded applications, and found they were few, and seldom used. When looking at potential related work in modelling application security or secure node management, we found virtually no work done, which may be explained by lack of application models. As such, in this thesis we extended a standard application model (SCA [16]) to support the necessary application abstraction we needed, and built our security abstractions from there.

The second lesson learned is that in order to roll out a secure system, you need more than a chain of implementations of single secure subsystems. Many embedded systems focus solely on a single problem, such as security, and often during a very specific point in the lifecycle, such as application communication security. However, what this thesis taught me, was that in order to roll out a fairly secure deployment, one can not just append different security systems, one needs an integrated security middleware, with the necessary abstractions to actually manage the middleware. For example, to enable an application owner to enforce that his location data sent confidentially depends on a chain of systems: it depends on the network being initialised, having keys initialised, having the application modeled so the system knows where location data is transmitted and received, on having the systems in place to enforce the security policy and finally on having the abstractions to express the policy. Targeting a full holistic view of the system and its security requirements enabled us to create a full, usable and streamlined security infrastructure which was able to be rolled out in a real world deployment.

The third lesson learned is the importance of software evolution, and decent support for such evolution on these tiny embedded devices, both during development, and operation. During development, a developer often wants to test his components to ensure they work well. Often, to perform end-to-end tests, one needs to be able to deploy these components onto a running system. An environment where a developer can easily update deployed code to fix issues greatly speeds up the development process. Also during operation, evolution is crucial for both functional and security concerns. The testbed has shown, that often unforeseen circumstances require small changes in the running systems. Software evolution allows users to easily roll out these small updates. Also from a security perspective, evolution is crucial. If events in the past year should have taught us anything, it is that there are bugs in every system which threaten the security of that system. Easy software evolution is essential to quickly mitigate potential risks due to faulty software.

The fourth and final lesson learned in this thesis is the value of a continuously

running prototype, coupled with a strong monitoring solutions. The smart office testbed provided us with a test environment with real functional concerns: people wanted to have monitoring data about the environment, and wanted to be able to use the coffee machine at any time. This required the system to operate continuously without interruptions. A permanent operating system requires a lot more stability of the system, compared to the often used prototype solutions. This led to a lot of fixes in all layers of the software, which led to a more stable systems. To identify such issues, the monitoring system was a great help. By clearly identifying the relative workload of different parts of the system, and by continuously being able to monitor memory, processing and communication, problems can be very quickly identified. The second way in which the testbed deployment aided the research was that it forced us to use the modelling solutions we developed, in a system that comes as close to a production system as it gets in research. This allowed us to iterate on the solutions and abstractions offered, and led us to clearly validate the importance of the modelling approaches to ensure the continuous stable environment.

9.3 Future Work

This research proposed an infrastructure to enable the secured sharing of embedded networks. This work presented a set of security abstractions which the different parties can use during the lifecycle of the embedded application, and a sensor network security middleware which is able to enforce these abstractions. However, while this work is a significant step forward with regards to securing shared embedded networks, there are still areas outside of the core focus of this work, which are required to create end-user friendly, fully secure, shareable embedded networks, namely: 1) improvements can be made to the underlying node hardware and operating system to ensure the safe, isolated and secure collaboration of the different parts of these embedded networks, and 2) further work should be done with regards to raising the abstraction level of the management of these embedded networks to enable non-specialist to easily set up secure embedded networks. This section will now detail these two avenues of related work.

9.3.1 Secure Underlying Environment

This thesis proposed a node security middleware that aims to allow the sharing of embedded networks at a node level. The current middleware allows those applications to be securely deployed, to communicate securely, and to securely manage those applications. However, what the node middleware cannot enforce,

is the secure and isolated execution of these applications while using these shared platforms, nor the secure communication with the peripherals.

Future work should look at how these embedded applications can execute securely and isolated from each other. Ideally they would operate similarly as current cloud based virtual machines, where the actions of one virtual machine can have virtually no influence on the operation of another virtual machine. However, current generation of embedded devices offer little to none of the memory protection, and safe multi-threading that current generation CPUs offer. Without such protection, it will be near impossible to enforce resource restrictions and processing limitations, or event that the deployed code cannot alter other parts of the system. Related work, partially done within DistriNet, has provided some solutions that potentially provides this isolation, such as Sancus [82] and Fides [103]. However currently it is still unclear how these systems can be used to build large secure evolvable software environments and truly apply these systems in real world deployments.

A second avenue of future work is to look at how to securely connect different peripherals with the embedded nodes. For many security applications, it is vital that the entire application flow from for example identification through RFID card or biometric scanner, to the unlocking of a door with smart lock, is entirely secure, and cannot be intercepted by attackers. This would ensure that any data received from a sensor is authentic, and would ensure that only authentic commands to an actuator would be executed.

9.3.2 End-user Support

To truly unlock the vast opportunities that shared embedded networks offer, they must become as ubiquitous as currently the computer or the smart phone. The average end-user needs to be able to deploy and manage such embedded networks, with little to no specialised knowledge of the underlying system and middleware. However, to get there, there still needs to be significant improvements in many different areas of embedded networks: 1) the deployment and setup needs to be made easier, 2) the creation and deployment of applications must be as intuitive as downloading an app for the smart phone while ensuring security, and 3) security and trust must be easily and intuitively set up, with strong default settings with regards to data and communication security.

Currently the deployment of embedded networks requires a great deal of knowledge of many different subareas of embedded networks. To create a real deployment, the deployer must go through an entire cycle of steps. In short he must currently: 1) select a suitable embedded node, 2) download and deploy a custom operating system for that node, 3) choose sensor and

actuator hardware to enact the desired functionality, adapt the hardware to be able to perform the functionality, and connect to the embedded node, which often involves soldering, 4) write custom code for both the embedded node and back-end application server to adequately enact the functionality, 5) set up a wireless network and 6) hopefully add some security protocols to the code for communication or storage. Clearly this is outside the skill set of the average consumer, who wants to buy the networked device as is, or perhaps as a system that can be easily set up and connected, and then with minimal effort place it in the environment, and view the desired monitoring data securely on his pc. Within DistriNet we have partially tried to tackle these issues, such as the LooCI middleware environment on top of Contiki [51], which provides a ready to use operating system, networking stack and application interface, and the micro plug and play solution that allows users to easily attach plug and play hardware to sensors [121]. While these are vital steps towards ease of use, more work still needs to be done to streamline the deployment for non-expert end-users.

This also leads to the second issue, which is the creation and deployment of embedded software applications. As stated previously, currently it is often the application deployer who has to write his application from scratch. However, this is clearly not transferable to end-users. End-users should be able to download an application from some kind of application store, and enact it on his personal environment. The presented application and component modelling abstractions potentially provide the necessary tools for developers to create such generic applications, which can be deployed on any compatible system. However, in order for this approach to gain widespread traction, a much larger ecosystem must emerge where application developers can easily write and share embedded application, and application users can easily download and deploy said applications in a secure local environment.

This brings us to the last need for end-user support, which is most closely related to the thesis: currently the different parties must encode their security requirements and trust relationships using a simple tool. However, while the architecture can scale to large organisations or ecosystems, the provided interface will not. Users must easily be able to specify who they trust, and to what extent, and with which resources. Additionally they should somehow either be able to specify which resources and data they consider critical and confidential, and which data they are willing to share with whom. As long as end-users cannot easily and dependably dictate the conditions with which they share, it is unlikely that many shared environments will be deployed in the world.

9.4 Outlook

Shared networked embedded systems have the opportunity to impact our daily lives, similar to what computers, the Internet and smart phones have done over the past decades. It is a vision where people are continuously interacting wirelessly with many networked devices in the environment, that these systems would offer the necessary services to sense and enact the desired behaviour on the environment. In this mist of devices, we would be continuously monitoring our surroundings, and changing it to suit our needs better: an environment-as-a-service, similar to the current platform-as-a-service or infrastructure-as-a-service. However, before we get there we see many challenges: 1) how to deploy such systems and networks, 2) how to develop for such systems, 3) how to handle the data, how to 4) manage these systems, and 5) lastly, and perhaps most vitally, how do we secure such systems.

The deployment and setup of these networks is still very much uncertain. Currently significant research is still being done on how to optimally set up the low level network protocols, in a way which is often incompatible with every other current network protocol. In order for these shared networked embedded systems to become a reality, standards must emerge which would enable these devices not only to communicate packets with each other, but also allow them to exchange the full richness of information and services provided by these networked embedded systems

With regards to developing for networked embedded systems, there are currently still many possible solutions on how to develop for these systems, and deploy applications on these systems: will we use HTTP or CoAP [100] services, will we use component models, will we use agents or other software paradigms. Currently no standard has evolved but many proposals have been made, which shows the complexity of developing for and interacting with these systems. Additionally, in order to fully unlock the embedded ecosystem, it is not sufficiently to just be able to easily develop for these systems, an entire application ecosystem must emerge similar to the current app stores, where users can just download their smart application, which attempts to enable the desired embedded functionality.

Third, we can see that these networked embedded systems will send us a vast amount of continuous data streams, which need to processed in real time or at least as soon as possible, and decisions need to be made on how to adapt the system to our needs. There is still vast amounts of work to be done in this field, going from activity recognition, to environment discovery, to intent and context determination.

Fourth, the amount of networked embedded devices will not number in the thousands, nor millions, but there will be billions or even trillions of connected

devices. This means that every person on earth has hundreds to thousands of nodes under his or her personal control. There is first the obvious the physical management question of who places which node, and who cleans up which node, but second, also the question of how we can keep logical control or even how we interact with these systems. A large question is still how a simple end-user will actually interact with these systems, and how the user can actually imprint his desires to be enacted in the environment. While currently no standard interaction paradigms have emerged, I believe augmenting reality based on visual discovery of nodes in the environment is a promising avenue worth further study.

And lastly, and perhaps most importantly, a large question is still how we will ensure that these systems are and remain secure during their entire lifetime. These networked embedded systems will produce vast amounts of sensitive data which need to be communicated, stored and processed, making security a crucial enabler to ensure trust in the system, and create industry and social acceptance. This dissertation proposed a framework that would enable users to express trust relationships, model applications, restrict resource usage, and express security requirements on different kinds of data, and it presented a security middleware to enforce these policies. While this dissertation is a step towards the goal of secure shared embedded systems, work remains to be done to ensure that all end-users can easily set up networked embedded system environments, safe from attackers, where they can securely share data and services with trusted collaborators, friends, and family.

Bibliography

[1] Avr-crypto-lib, mar 2011. Available as http://avrcryptolib.das-labor.org.

[2] Aranha, D. F., Lopez, J., Oliveira, L. B., and Dahab, R. Efficient implementation of elliptic curves on sensor nodes. In *Conference on Hyperelliptic curves, discrete Logarithms, Encryption, etc., Frutillar, Chile* (2009).

[3] Atmel. AVR raven, 2008. Available as http://www.atmel.com/tools/AVRRAVEN.aspx.

[4] Atmel. Atmega1284p, 2009. Available as http://www.atmel.com/devices/atmega1284p.aspx.

[5] Atmel. Atmega128rfa1, 2009. Available as http://www.atmel.com/devices/atmega128rfa1.aspx.

[6] Awad, A., Nebel, R., German, R., and Dressler, F. On the need for passive monitoring in sensor networks. *Digital Systems Design, Euromicro Symposium on 0* (2008), 693–699.

[7] Benenson, Z. Realizing robust user authentication in sensor networks. In *Real-World Wireless Sensor Networks (REALWSN)* (2005).

[8] Bluetooth, S. Bluetooth specification, 2007.

[9] Burns, S. F. Threat modeling: A process to ensure application security. *GIAC Security Essentials Certification (GSEC) Practical Assignment* (2005).

[10] Camtepe, S. A., and Yener, B. Key distribution mechanisms for wireless sensor networks: a survey. Tech. rep., Rensselaer Polytechnic Institute, 2005.

[11] Cantor, S., Kemp, J., Philpott, R., and Maler, E. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0. Tech. rep., Mar. 2005.

[12] Cardell-Oliver, R., Kranz, M., Smettem, K., and Mayer, K. A reactive soil moisture sensor network: Design and field evaluation. *International Journal of Distributed Sensor Networks 1*, 2 (2005), 149–162.

[13] Case, J., Mundy, R., Partain, D., and Stewart, B. Introduction to Version 3 of the Internet-standard Network Management Framework. RFC 2570, Apr. 1999.

[14] Castelluccia, C., Francillon, A., Perito, D., and Soriente, C. On the difficulty of software-based attestation of embedded devices. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security* (New York, NY, USA, 2009), ACM Press, pp. 400–409.

[15] Chapin, P., and Skalka, C. SpartanRPC: Secure WSN middleware for cooperating domains. In *Mobile Adhoc and Sensor Systems (MASS), 2010 IEEE 7th International Conference on* (nov. 2010), pp. 61 –70.

[16] Chapman, M., Edwards, M., Beisiegel, M., Karmarkar, A., Patil, S., and Rowley, M. Service component architecture assembly model specification version 1.1. Tech. rep., OASIS, 2011.

[17] Chaudet, C., Fleury, E., Lassous, I. G., Rivano, H., and Voge, M.-E. Optimal positioning of active and passive monitoring devices. In *Proceedings of the 2005 ACM Conference on Emerging Network Experiment and Technology* (New York, NY, USA, 2005), CoNEXT '05, ACM, pp. 71–82.

[18] Chen, C.-Y., and Chao, H.-C. A survey of key distribution in wireless sensor networks. *Security and Communication Networks* (2011).

[19] Claycomb, W. R., and Shin, D. A novel node level security policy framework for wireless sensor networks. *Journal of Network and Computer Applications 34*, 1 (2011), 418 – 428.

[20] Consortium, O. G. Sensor model language, 2013.

[21] Cuomo, F., Abbagnale, A., and Cipollone, E. Cross-layer network formation for energy-efficient IEEE 802.15. 4/ZigBee wireless sensor networks. *Ad Hoc Networks 11*, 2 (2013), 672–686.

[22] Customs, U., and Protection, B. C-TPAT: Customs-Trade Partnership Against Terrorism, 2006. http://www.cbp.gov/xp/cgov/trade/cargo_security/ctpat/.

[23] Daemen, J., Peeters, M., Van Assche, G., and Rijmen, V. NESSIE proposal: NOEKEON. In *First Open NESSIE Workshop* (2000).

[24] Dahlman, E., Gudmundson, B., Nilsson, M., and Skold, A. UMTS/IMT-2000 based on wideband CDMA. *Communications Magazine, IEEE 36*, 9 (1998), 70–80.

[25] Dahlman, E., Parkvall, S., and Skold, J. *4G: LTE/LTE-advanced for mobile broadband*. Academic press, 2013.

[26] Damianou, N., Dulay, N., Lupu, E., and Sloman, M. The ponder policy specification language. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks* (London, UK, 2001), Springer-Verlag, pp. 18–38.

[27] de Meulenaer, G., Gosset, F., Standaert, O.-X., and Pereira, O. On the energy cost of communication and cryptography in wireless sensor networks. In *Networking and Communications, 2008. WIMOB '08. IEEE International Conference on Wireless and Mobile Computing,* (2008), pp. 580–585.

[28] Dierks, T., and Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor, August 2008.

[29] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The second-generation onion router. Tech. rep., DTIC Document, 2004.

[30] Dolev, D., and Yao, A. On the security of public key protocols. *Information Theory, IEEE Transactions on 29*, 2 (1983), 198–208.

[31] Dunkels, A., Gronvall, B., and Voigt, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 455–462.

[32] Dunkels, A., Osterlind, F., Tsiftes, N., and He, Z. Software-based on-line energy estimation for sensor nodes. In *Proceedings of the 4th Workshop on Embedded Networked Sensors* (New York, NY, USA, 2007), EmNets '07, ACM, pp. 28–32.

[33] Dyo, V., Ellwood, S. A., Macdonald, D. W., Markham, A., Trigoni, N., Wohlers, R., Mascolo, C., Pásztor, B., Scellato, S., and Yousef, K. Wildsensing: Design and deployment of a sustainable

sensor network for wildlife monitoring. *ACM Trans. Sen. Netw. 8*, 4 (Sept. 2012), 29:1–29:33.

[34] Eschenauer, L., and Gligor, V. D. A key-management scheme for distributed sensor networks. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security* (New York, NY, USA, 2002), ACM Press, pp. 41–47.

[35] Farahani, S. *ZigBee Wireless Networks and Transceivers.* Newnes, Newton, MA, USA, 2008.

[36] Freitas, L., Bispo, K., Rosa, N., and Cunha, P. SM-Sens: Security middleware for wireless sensor networks. In *Information Infrastructure Symposium, 2009. GIIS '09. Global* (june 2009), pp. 1 –7.

[37] Fysarakis, K., Papaefstathiou, I., Manifavas, C., Rantos, K., and Sultatos, O. Policy-based access control for DPWS-enabled ubiquitous devices. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE* (Sept 2014), pp. 1–8.

[38] Gamma, E., Helm, R., Johnson, R., and Vlisides, J. *Design patterns.* Addison Wesley, 2007.

[39] Godik, S., and Moses, T. Extensible access control markup language (xacml), v2.0. Tech. rep., OASIS, 2005.

[40] Gomez, C., Oller, J., and Paradells, J. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors 12*, 9 (2012), 11734–11753.

[41] Group, O. M. G. UML Specification, Version 2.0.

[42] Gruteser, M., and Grunwald, D. Enhancing location privacy in wireless lan through disposable interface identifiers: A quantitative analysis. *Mobile Networks and Applications 10*, 3 (2005), 315–325.

[43] Gungor, V., and Hancke, G. Industrial wireless sensor networks: Challenges, design principles, and technical approaches. *Industrial Electronics, IEEE Transactions on 56*, 10 (oct. 2009), 4258 –4265.

[44] Halperin, D., Heydt-Benjamin, T. S., Ransford, B., Clark, S. S., Defend, B., Morgan, W., Fu, K., Kohno, T., and Maisel, W. H. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (Washington, DC, USA, 18-22 2008), IEEE Computer Society, pp. 129 –142.

[45] He, D., Bu, J., Zhu, S., Chan, S., and Chen, C. Distributed access control with privacy support in wireless sensor networks. *Wireless Communications, IEEE Transactions on 10*, 10 (october 2011), 3472 –3481.

[46] He, D., Chen, C., Chan, S., and Bu, J. Sdrp: A secure and distributed reprogramming protocol for wireless sensor networks. *Industrial Electronics, IEEE Transactions on 59*, 11 (Nov 2012), 4155–4163.

[47] Healy, M., Newe, T., and Lewis, E. Efficiently securing data on a wireless sensor network. *Journal of Physics: Conference Series 76*, 1 (2007), 012063.

[48] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. System architecture directions for networked sensors. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2000), ACM Press, pp. 93–104.

[49] Horré, W., Michiels, S., Joosen, W., and Hughes, D. QARI: quality aware software deployment for wireless sensor networks. In *Seventh International Conference on Information Technology: New Generations, ITNG 2010, Las Vegas, Nevada, USA, 12-14 April 2010* (2010), pp. 642–647.

[50] Hu, W., Tan, H., Corke, P., Shih, W. C., and Jha, S. Toward trusted wireless sensor networks. *ACM Transactions on Sensor Networks 7*, 1 (2010), 1–25.

[51] Hughes, D., Thoelen, K., Maerien, J., Matthys, N., Horre, W., Del Cid, J., Huygens, C., Michiels, S., and Joosen, W. LooCI: the loosely-coupled component infrastructure. In *Network Computing and Applications(NCA), 2012* (2012), IEEE, pp. 236–243.

[52] Hui, J. W., and Culler, D. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04* (New York, NY, USA, 2004), ACM, pp. 81–94.

[53] Hyncica, O., Kucera, P., Honzik, P., and Fiedler, P. Performance evaluation of symmetric cryptography in embedded systems. In *Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2011 IEEE 6th International Conference on* (Sept 2011), vol. 1, pp. 277–282.

[54] Hyun, S., Ning, P., Liu, A., and Du, W. Seluge: Secure and dos-resistant code dissemination in wireless sensor networks. In *Information*

Processing in Sensor Networks, 2008. IPSN '08. International Conference on (Apr. 2008), pp. 445 –456.

[55] IEEE Std 802.11-2007. IEEE standard for information technology — telecommunications and information exchange between systems — local and metropolitan area networks — specific requirements — part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications, 2007.

[56] International, S. Architecture analysis and design language (AADL), 2012.

[57] Jelicic, V., Magno, M., Brunelli, D., Bilas, V., and Benini, L. Analytic comparison of wake-up receivers for WSNs and benefits over the wake-on radio scheme. In *Proceedings of the 7th ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks* (2012), ACM, pp. 99–106.

[58] Karlof, C., Sastry, N., and Wagner, D. TinySec: a link layer security architecture for WSNs. In *SenSys '04* (New York, NY, USA, 2004), ACM, pp. 162–175.

[59] Katz, J. Universally composable multi-party computation using tamper-proof hardware. In *Advances in Cryptology - EUROCRYPT 2007*, M. Naor, Ed., vol. 4515 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007, pp. 115–128.

[60] Kent, S., and Seo, K. Security Architecture for the Internet Protocol. RFC 4301, RFC Editor, December 2005.

[61] Kirk, J. Pacemaker hack can deliver deadly 830-volt jolt. *Computerworld 17* (2012).

[62] Kothmayr, T., Schmitt, C., Hu, W., Brunig, M., and Carle, G. Dtls based security and two-way authentication for the internet of things. *Ad Hoc Netw. 11*, 8 (Nov. 2013), 2710–2723.

[63] Kovatsch, M., Mayer, S., and Ostermaier, B. Moving application logic from the firmware to the cloud: Towards the thin server architecture for the internet of things. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on* (2012), pp. 751–756.

[64] Kuang, X., and Shen, J. Snds: A distributed monitoring and protocol analysis system for wireless sensor network. *Networks Security, Wireless Communications and Trusted Computing, International Conference on 2* (2010), 422–425.

[65] Lanigan, P. E., Gandhi, R., and Narasimhan, P. Sluice: Secure dissemination of code updates in sensor networks. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems* (Washington, DC, USA, 2006), IEEE Computer Society, p. 53.

[66] Liu, A., and Ning, P. Tinyecc: A configurable library for elliptic curve cryptography in wireless sensor networks. In *Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on* (April 2008), pp. 245–256.

[67] Liu, A., Ning, P., and Wang, C. Lightweight remote image management for secure code dissemination in wireless sensor networks. In *INFOCOM 2009, IEEE* (Apr. 2009), pp. 1242 –1250.

[68] Lynch, J. P., and Loh, K. J. A summary review of wireless sensors and sensor networks for structural health monitoring. *Shock and Vibration Digest 38*, 2 (2006), 91–130.

[69] Maerien, J., Agten, P., Huygens, C., and Joosen, W. Famos: A flexible active monitoring service for wireless sensor networks. *Distributed Applications and Interoperable Systems* (2012), 104–117.

[70] Maerien, J., Michiels, S., Hughes, D., Huygens, C., and Joosen, W. Seclooci: A comprehensive security middleware architecture for shared wireless sensor networks. *Ad Hoc Networks 25, Part A*, 0 (2015), 141 – 169.

[71] Maerien, J., Michiels, S., Huygens, C., Hughes, D., and Joosen, W. Access control in multi-party wireless sensor networks. In *Wireless Sensor Networks*, vol. 7772. Springer Berlin Heidelberg, 2013, pp. 34–49.

[72] Maerien, J., Michiels, S., Huygens, C., Hughes, D., and Joosen, W. Enabling resource sharing in heterogeneous wireless sensor networks. In *Proceedings of the 1st ACM Workshop on Middleware for Context-Aware Applications in the IoT* (New York, NY, USA, 2014), M4IOT '14, ACM, pp. 7–12.

[73] Maerien, J., Michiels, S., Huygens, C., and Joosen, W. Masy: Management of secret keys for federated mobile wireless sensor networks. In *Wireless and Mobile Computing, Networking and Communications, 2010. WIMOB 2010. IEEE International Conference on* (Washington, DC, USA, oct. 2010), IEEE Computer Society.

[74] Maerien, J., Michiels, S., Huygens, C., and Joosen, W. Sasha: A distributed protocol for secure application deployment in shared ad-hoc

wireless sensor networks. In *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on* (oct. 2011), pp. 43 –48.

[75] Maes, R., Van Herrewege, A., and Verbauwhede, I. Pufky: A fully functional puf-based cryptographic key generator. In *Cryptographic Hardware and Embedded Systems (CHES 2012)*, E. Prouff and P. Schaumont, Eds., vol. 7428 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 302–319.

[76] Mainwaring, A., Culler, D., Polastre, J., Szewczyk, R., and Anderson, J. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications* (New York, NY, USA, 2002), WSNA '02, ACM, pp. 88–97.

[77] Malan, D. J., Welsh, M., and Smith, M. D. Implementing public-key infrastructure for sensor networks. *ACM Transaction on Sensor Networks 4*, 4 (2008), 1–23.

[78] Matthys, N. *Software Technologies for Dynamic Sensor Networks*. PhD thesis, KULeuven, February 2014.

[79] Montenegro, G., Kushalnagar, N., Hui, J., and Culler, D. Transmission of IPv6 packets over IEEE 802.15. 4 networks. *Internet proposed standard RFC 4944* (2007).

[80] Motiev. Tmote sky, 2006. Available as `http://www.snm.ethz.ch/Projects/TmoteSky`.

[81] Neuman, B. C., and Ts'o, T. Kerberos: an authentication service for computer networks. *Communications Magazine, IEEE 32*, 9 (sep 1994), 33 –38.

[82] Noorman, J., Agten, P., Daniels, W., Strackx, R., Van Herrewege, A., Huygens, C., Preneel, B., Verbauwhede, I., and Piessens, F. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security* (2013), pp. 479–494.

[83] O Hanlon, P., Wright, J., Brown, I., and de Souza, T. Kemf: Key management for federated sensor networks, 2014.

[84] OSGi Alliance. *OSGi service platform, release 3*. IOS Press, Inc., 2003.

[85] Othman, M. F., and Shazali, K. Wireless sensor network applications: A study in environment monitoring system. *Procedia Engineering 41* (2012), 1204–1210.

[86] OWASP. Open web application security project, 2013. www.owasp.org.

[87] Pantelopoulos, A., and Bourbakis, N. G. Prognosis - a wearable health-monitoring system for people at risk: Methodology and modeling. *Information Technology in Biomedicine, IEEE Transactions on 14*, 3 (may 2010), 613 –621.

[88] Parliament, T. E., and of the Council. Authorised Economic Operator certificate, 2005. http://ec.europa.eu/taxation_customs/customs/policy_issues/customs_security/aeo/.

[89] Porter, B., and Coulson, G. Lorien: a pure dynamic component-based operating system for wireless sensor networks. In *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks* (2009), ACM, pp. 7–12.

[90] Postel, J. Rfc 768: User datagram protocol. IETF Request For Comments, Aug. 1980.

[91] Postel, J. Rfc 793: Transmission control protocol. IETF Request For Comments, Sept. 1981.

[92] Raij, A., Ghosh, A., Kumar, S., and Srivastava, M. Privacy risks emerging from the adoption of innocuous wearable sensors in the mobile environment. In *Proceedings of the 2011 annual conference on Human factors in computing systems* (New York, NY, USA, 2011), CHI '11, ACM, pp. 11–20.

[93] Ramachandran, B., Dwyer, J., Raup, B. H., and Kargel, J. S. Aster datasets and derived products for global glacier monitoring. In *Global Land Ice Measurements from Space*. Springer, 2014, pp. 145–162.

[94] Ramanathan, N., Chang, K., Kapur, R., Girod, L., Kohler, E., and Estrin, D. Sympathy for the sensor network debugger. In *Proceedings of the 3rd international conference on Embedded networked sensor systems* (New York, NY, USA, 2005), SenSys '05, ACM, pp. 255–267.

[95] Raza, S., Duquennoy, S., Chung, T., Yazar, D., Voigt, T., and Roedig, U. Securing communication in 6LoWPAN with compressed IPsec. In *Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on* (June 2011), pp. 1–8.

[96] Ren, K., Lou, W., and Zhang, Y. Multi-user broadcast authentication in wireless sensor networks. In *SECON '07* (june 2007), pp. 223 –232.

[97] Rescorla, E., and Modadugu, N. Datagram Transport Layer Security Version 1.2. RFC 6347, RFC Editor, January 2012.

[98] Roman, R., Lopez, J., Alcaraz, C., and Chen, H.-H. Sensekey – simplifying the selection of key management schemes for sensor networks. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on* (march 2011), pp. 789 –794.

[99] Sheet, T. S. D. Moteiv, San Francisco, CA, 2006, 2004.

[100] Shelby, Z., Hartke, K., and Bormann, C. Constrained application protocol (coap). Tech. rep., Internet Engineering Task Force, 2013. `http://tools.ietf.org/html/draft-ietf-core-coap-14`.

[101] Smits, E., Ratinckx, E., Thoen, V., Debackere, K., Monard, E., and Raspoet, D. Technologie en innovatie in vlaanderen : Prioriteiten. Tech. rep., Vlaamse Raad voor Wetenschapsbeleid, 2006.

[102] Sodano, H. A., Inman, D. J., and Park, G. A review of power harvesting from vibration using piezoelectric materials. *Shock and Vibration Digest 36*, 3 (2004), 197–206.

[103] Strackx, R., and Piessens, F. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 2–13.

[104] Studer, A., Shi, E., Bai, F., and Perrig, A. TACKing together efficient authentication, revocation, and privacy in VANETs. In *SECON'09: Proceedings of the 6th Annual IEEE communications society conference on Sensor, Mesh and Ad Hoc Communications and Networks* (Piscataway, NJ, USA, 2009), IEEE Press, pp. 484–492.

[105] Szczechowiak, P., Kargl, A., Scott, M., and Collier, M. On the application of pairing based cryptography to wireless sensor networks. In *WiSec '09: Proceedings of the second ACM conference on Wireless network security* (New York, NY, USA, 2009), ACM Press, pp. 1–12.

[106] Szczechowiak, P., Oliveira, L., Scott, M., Collier, M., and Dahab, R. NanoECC: Testing the limits of elliptic curve cryptography in sensor networks. In *Wireless Sensor Networks*, R. Verdone, Ed., vol. 4913 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 305–320.

[107] Taherkordi, A., Loiret, F., Abdolrazaghi, A., Rouvoy, R., Le-Trung, Q., and Eliassen, F. Programming sensor networks using remora component model. In *Distributed Computing in Sensor Systems*, vol. 6131. Springer Berlin Heidelberg, 2010, pp. 45–62.

[108] Tan, H., Ostry, D., Zic, J., and Jha, S. A confidential and DoS-resistant multi-hop code dissemination protocol for wireless sensor networks. In *WiSec '09: Proceedings of the second ACM conference on Wireless network security* (New York, NY, USA, 2009), ACM, pp. 245–252.

[109] Tanaka, T., Sonoda, K., Okochi, S., Chan, A., Nii, M., Kanda, K., Fujita, T., Higuchi, K., and Maenaka, K. Wearable health monitoring system and its applications. In *Emerging Trends in Engineering and Technology (ICETET), 2011 4th International Conference on* (Nov 2011), pp. 143–146.

[110] Tang, L., Sun, Y., Gurewitz, O., and Johnson, D. B. PW-MAC: An energy-efficient predictive-wakeup MAC protocol for wireless sensor networks. In *INFOCOM, 2011 Proceedings IEEE* (2011), IEEE, pp. 1305–1313.

[111] Thomson, S., Narten, T., and Jinmei, T. IPv6 stateless address autoconfiguration. RFC 4862 (Draft Standard), September 2007.

[112] Torr, P. Demystifying the threat modeling process. *Security Privacy, IEEE 3*, 5 (2005), 66–70.

[113] Tripathy, S. Effective pair-wise key establishment scheme for wireless sensor networks. In *SIN '09: Proceedings of the 2nd international conference on Security of information and networks* (New York, NY, USA, 2009), ACM, pp. 158–163.

[114] Van Herbruggen, J. COMACOD Control and Management of Constrained Devices, 2010. `https://distrinet.cs.kuleuven.be/research/projects/COMACOD`.

[115] Vollbrecht, J., Calhoun, P., Farrell, S., Gommans, L., Gross, G., de Bruijn, B., de Laat, C., Holdrege, M., and Spence, D. AAA Authorization Framework. Tech. Rep. 2904, August 2000.

[116] Want, R. An introduction to RFID technology. *Pervasive Computing, IEEE 5*, 1 (2006), 25–33.

[117] Want, R. Near field communication. *IEEE Pervasive Computing 10*, 3 (2011), 0004–7.

[118] Watteyne, T., Molinaro, A., Richichi, M. G., and Dohler, M. From MANET to IETF ROLL standardization: A paradigm shift in WSN routing protocols. *Communications Surveys & Tutorials, IEEE 13*, 4 (2011), 688–707.

[119] Werner, F., and Benenson, Z. Formally verified authenticated query dissemination in sensor networks. In *SPECTS'09* (2009), IEEE Press, pp. 154–161.

[120] Westin, A. F. *Privacy and Freedom.* Bodley Head, 1970.

[121] Yang, F., Matthys, N., Bachiller, R., Michiels, S., Joosen, W., and Hughes, D. μ pnp: plug and play peripherals for the internet of things. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 25.

[122] Yu, S., Ren, K., and Lou, W. Fdac: Toward fine-grained distributed data access control in wireless sensor networks. *Parallel and Distributed Systems, IEEE Transactions on 22*, 4 (april 2011), 673 –686.

[123] Zatout, Y., Campo, E., and Llibre, J.-F. WSN-HM: Energy-efficient WSN for home monitoring. In *ISSNIP* (Washington, DC, USA, dec. 2009), IEEE Computer Society, pp. 367 –372.

[124] Zhang, Z., Chen, Q., Bergarp, T., Norman, P., Wikstrom, M., Yan, X., and Zheng, L.-R. Wireless sensor networks for logistics and retail. In *INSS* (Washington, DC, USA, 17-19 2009), IEEE Computer Society, pp. 1 –4.

[125] Zhu, S., Setia, S., and Jajodia, S. Leap: efficient security mechanisms for large-scale distributed sensor networks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security* (New York, NY, USA, 2003), ACM, pp. 62–72.

List of publications

Journal articles

Jef Maerien, Sam Michiels, Danny Hughes, Christophe Huygens, and Wouter Joosen. 2015. SecLooCI: A comprehensive security middleware architecture for shared wireless sensor networks. In Ad Hoc Networks, Volume 25, Part A, February 2015, Pages 141-169, ISSN 1570-8705. Lead author, performed design, prototyping, and validation.

International conference papers

Jef Maerien, Sam Michiels, Christophe Huygens, Danny Hughes, and Wouter Joosen. 2013. Access Control in Multi-party Wireless Sensor Networks. In Wireless Sensor Networks Lecture Notes in Computer Science Volume 7772, 2013, pp 34-49. Gent, Belgium, 13-15 Feb 2013. Lead author, performed design, prototyping, and validation.

Danny Hughes, Eduardo Canete, Wilfried Daniels, Gowri Sankar Ramachandran, James Meneghello, Nelson Matthys, Jef Maerien, Sam Michiels, Christophe Huygens, Wouter Joosen, Maarten Wijnants, Wim Lamotte, Erik Hulsmans, Bart Lannoo, and Ingrid Moerman. 2013. Energy aware software evolution for Wireless Sensor Networks. In World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2013 IEEE 14th International Symposium and Workshops on a , pp 1-9. Madrid, Spain, 4-7 Jun 2013. Participated in text review.

Danny Hughes, Klaas Thoelen, Jef Maerien, Nelson Matthys, Javier Del Cid, Wouter Horre, Christhope Huygens, Sam Michiels, and Wouter Joosen. LooCI: The Loosely-coupled Component Infrastructure. In 2012 11th IEEE

International Symposium on Network Computing and Applications (NCA 2012), pp 236-243. Cambridge, Massachusetts, USA, 23-25 Aug 2012. Participated in prototyping and text review.

Jef Maerien, Pieter Agten, Christophe Huygens, and Wouter Joosen. 2012. FAMoS: A Flexible Active Monitoring Service for Wireless Sensor Networks. In Distributed Applications and Interoperable Systems. Lecture Notes in Computer Science Volume 7272, 2012, pp 104-117. Springer, Stockhold, Sweden, 13-16 Jun 2012. Lead author, participated in design and validation.

Jef Maerien, Sam Michiels, Christophe Huygens, and Wouter Joosen. 2011. SASHA: A Distributed Protocol for Secure Application Deployment in Shared Ad-Hoc Wireless Sensor Networks. In 8th IEEE International Conference on Mobile Adhoc and Sensor Systems (MASS 2011), pp 43-48. Valencia, Spain, 17-22 Oct 2011. Lead author, performed design, prototyping, and validation.

Christophe Huygens, Nelson Matthys, Jef Maerien, Wouter Joosen, and Danny Hughes. 2011. Building blocks for secure multiparty federated wireless sensor networks. In Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International, pp 2021-2027, 4-8 Jul 2011. Participated in prototyping and review.

Jef Maerien, Sam Michiels, Christophe Huygens, and Wouter Joosen. 2010. MASY: MAnagement of Secret keYs for federated mobile wireless sensor networks. In Wireless and Mobile Computing, Networking and Communications (WiMob), 2010 IEEE 6th International Conference on, pp 121-128. Niagara falls, Canada, 11-13 Oct 2010. Lead author, performed design, prototyping, and validation.

Jef Maerien, Sam Michiels, Stefan Van Baelen, Christophe Huygens, and Wouter Joosen. 2010. A secure multi-application platform for vehicle telematics. In IEEE vehicular technology conference: (VTC2010-Fall). Ottowa, Canada, 6-9 Sep 2010. Lead author, performed design, prototyping, and validation.

International workshop papers

Jef Maerien, Sam Michiels, Christophe Huygens, Danny Hughes, and Wouter Joosen. 2014. Enabling resource sharing in heterogeneous wireless sensor networks. In Proceedings of the 1st ACM Workshop on Middleware for Context-Aware Applications in the IoT (M4IOT '14), pp 7-12. Bordeaux, France, 8-12 Dec 2014. Lead author, performed design, prototyping, and validation.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Heverlee
jef.maerien@cs.kuleuven.be
http://www.distrinet.cs.kuleuven.be