KU LEUVEN

ARENBERG DOCTORAL SCHOOL
FACULTY OF ENGINEERING SCIENCE

Middleware Support for a Dynamic and Safe Internet of Things

Wilfried Daniels

Supervisors:
Prof. dr. D. Hughes
Prof. dr. ir. W. Joosen

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

June 2018

Middleware Support for a Dynamic and Safe Internet of Things

Wilfried DANIELS

Examination committee:
Prof. dr. A. Bultheel, chair
Prof. dr. D. Hughes, supervisor
Prof. dr. ir. W. Joosen, supervisor
Prof. dr. ir. H. Bruyninckx
Dr. S. Michiels
Prof. dr. K. Verbert
Prof. dr. K. Lee
(Nottingham Trent University, UK)
Prof. dr. A. Yasar
(Universiteit Hasselt)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

June 2018

© 2018 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Wilfried Daniels, Celestijnenlaan 200A box 2402, B-3001 Leuven (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

The dissertation embodies the results of over five years of research. While this work represents an important milestone in my professional career, it is important to look back to the long and winding road and thank the people that helped me along the way.

First of all I want to thank my supervisors, Danny Hughes and Wouter Joosen, for giving me the opportunity to pursue a PhD in DistriNet. Wouter always gave me solid advice during key moments of my PhD trajectory, while giving me the freedom to do my own thing. Danny and I worked closely together on a daily basis, discussing new paper ideas or just general interesting developments from our field. I thoroughly enjoyed our collaboration and learned a great deal from it. Thank you for your excellent guidance and support throughout my PhD.

I would also like to extend my gratitude to all members of my supervisory committee and jury, for their ongoing interest in my research and the useful discussion and suggestions I received to improve my work. Thank you Herman Bruyninckx, Sam Michiels, Katrien Verbert, Kevin Lee and Ansar Yasar for taking the time to read my dissertation and attend the defense. A special thanks goes to Sam, whose valuable feedback and insights helped me from day one into this journey. Lastly, I would like to thank Adhemar Bultheel for chairing the jury.

When I started my PhD I expected it to be a very solitary task, but I quickly realized that the opposite is true: research is never done alone. A few months after I started working here as a researcher, intense teamwork resulted in a first published paper. I would like to thank my colleagues from DistriNet, and in particular the Networked Embedded Software (NES) taskforce for their productive collaboration, useful exchanges of ideas and friendship that made working here a pleasure. Additional thanks go out to the people from the DistriNet Project Office and the secretaries, their help with the practical and administrative aspects of my PhD is greatly appreciated.

To all my friends, I would like to thank you as well. After a hectic workday, I could always count on you to make time for a relaxing chat and a beer, keeping me motivated during the hard times and making my time in Leuven among the best years of my life.

Finally all of this couldn't have been possible without my parents, my family and my in-laws. I will always be grateful for their continuous counsel, kind words and encouragement that kept me on my path.

Lastly I would like to thank my fiancée Karen. I would not have gotten this far without her unconditional love, care and support.

– Wilfried Daniels
Leuven, June 2018

This research is partially funded by the Agency for Innovation by Science and Technology in Flanders (IWT).

Abstract

The Internet of Things (IoT) bridges the physical world with the virtual world by embedding everyday objects with computation, communication and sensing abilities, interconnecting them with each other and the internet. As the IoT rapidly gains in popularity, the size and complexity of deployments is also increasing. Prime examples are smart cities and Industry 4.0, where the return on investment of large-scale infrastructure is improved by running multiple applications with ever changing requirements concurrently throughout the networks lifetime. Furthermore, with IoT technology becoming ubiquitous in our daily lives, security becomes a primary concern that is likely to grow in complexity as we move to more dynamic environments.

Dynamism at large scale requires fundamental support for software evolution and adaptation from the underlying software stack. State-of-the-art programming abstractions and reconfiguration approaches partially accommodate this through modular software development and runtime remote reconfiguration of distributed applications. While the advantages are clear, these solutions fall short when managing complex decentralized configurations. Incompatible configuration causes faults and downtime, and remote inspection and reconfiguration of applications incurs a considerable message overhead and energy cost on low power wireless networks. Contemporary solutions to secure the IoT require in-depth hardware modifications, raising the cost of roll-out for the millions of devices already active in the field and preventing their adoption.

This dissertation presents three contributions towards a dynamic and safe Internet of Things. The first contribution focuses on detecting and avoiding inconsistent configuration of distributed IoT applications. *Safe reparametrization* offers a descriptive language to developers for expressing distributed configuration dependencies, and uses a network protocol to resolve and enforce these dependencies at runtime. The overhead of this approach on representative hardware platforms is minimal, while management effort is significantly lowered when compared with exhaustive methods of ensuring safe

and correct reconfiguration.

The second contribution reduces the overhead and complexity of inspecting and reconfiguring distributed IoT applications through two novel approaches. *Refraction* selectively augments application data flows with meta-data, which travels to refractive pools without additional network traffic. Refractive pools serve as natural loci of low-cost inspection and control for distributed applications. Reactive policies deployed to these pools allow automatic reconfiguration based on changes in application state. *Tomography* further reduces overhead by recognizing that software components bound together are often queried in groups, using probes to traverse component compositions to collect meta-data and enact change efficiently. Prototype implementations have been evaluated within a smart office deployment, and show notable improvements in terms of management complexity and network traffic.

The final contribution looks at cost-effective security for the IoT. The *Security MicroVisor* is a pure-software approach that provides memory isolation for low-end IoT devices through partial virtualization of the instruction set. Subsequently, this memory isolation is used to safeguard the integrity of critical secure operations and secret key material while concurrently running insecure user applications, effectively providing a Trusted Computing Base. Evaluation shows that for typical usage patterns of IoT applications, virtualization latencies are imperceivable, while resource overhead is small.

Beknopte samenvatting

Het Internet der Dingen (Engels: Internet of Things, afkorting: IoT) slaat een brug tussen de fysieke wereld en de virtuele wereld door alledaagse objecten te verrijken met ingebedde informatieverwerking, communicatiemogelijkheden en sensoren, en ze te verbinden met elkaar en het internet. Terwijl IoT aan populariteit wint, worden de omvang en complexiteit van netwerken ook groter. Concrete voorbeelden zijn smart cities en Industry 4.0, waar het rendement op investeringen van grootschalige infrastructuur verbeterd wordt door meerdere toepassingen met continu veranderende vereisten gelijktijdig uit te rollen. Een bijkomend probleem dat gepaard gaat met de alomtegenwoordigheid van IoT-technologie is de beveiliging ervan, een probleem dat complexer wordt naarmate software dynamischer moet zijn.

Dynamiek op grote schaal vereist fundamentele ondersteuning voor software-evolutie en adaptatie van de onderliggende softwarestack. Actuele programmeerabstracties en methodologieën voor herconfiguratie vervullen deze vereisten gedeeltelijk met behulp van modulaire softwareontwikkeling en runtime herconfiguratie van op afstand voor gedistribueerde applicaties. Hoewel de voordelen duidelijk zijn, schieten deze oplossingen tekort bij het beheren van complexe gedecentraliseerde configuratie. Incompatibele configuratie veroorzaakt fouten en uitvalling, en herconfiguratie van op afstand zorgt voor een zware belasting van het netwerk met nefaste gevolgen voor het energieverbruik van IoT-apparaten. Hedendaagse oplossingen om de IoT te beveiligen berusten op diepgaande hardwaremodificaties, waardoor de kost te hoog is voor de beveiliging van de miljoenen bestaande apparaten die al in gebruik zijn.

Dit proefschrift presenteert drie bijdragen voor een dynamisch en veilig IoT. De eerste bijdrage focust op de detectie en vermijding van inconsistente configuratie van gedistribueerde IoT-toepassingen. *Veilige herparametrisatie* biedt een descriptieve taal aan voor ontwikkelaars om afhankelijkheden in gedistribueerde configuratie uit te drukken en gebruikt een netwerkprotocol om deze af te dwingen in een productieomgeving. De kost van deze aanpak op

een representatief platform is minimaal, terwijl de beheerinspanning significant verlaagd is ten opzichte van exhaustieve methoden die een veilige en correcte herconfiguratie garanderen.

De tweede bijdrage reduceert de kost en complexiteit van inspectie en herconfiguratie van gedistribueerde IoT-applicaties met twee nieuwe aanpakken. *Refraction* voegt selectief meta-data toe aan bestaande applicatiestromen, die vervolgens verzameld wordt in refractive pools zonder bijkomende netwerktransmissies. Refractive pools dienen als natuurlijke locaties voor lage kost inspectie en controle van gedistribueerde applicaties. Reactive policies staan automatische herconfiguratie toe, gebaseerd op veranderingen in de toestand van applicaties. *Tomography* reduceert de kosten verder door te erkennen dat softwarecomponenten die samengebonden zijn vaak in groep beheerd worden. Er wordt gebruikgemaakt van sondes die zich door componentcomposities verplaatsen om efficiënt meta-data te verzamelen en veranderingen door te voeren. Prototype implementaties werden geëvalueerd in een smart office scenario, en illustreren een duidelijke verbetering in beheerscomplexiteit en netwerkverkeer.

De laatste bijdrage spitst zich toe op de kosteneffectieve beveiliging van IoT-technologie. De *Security MicroVisor* is een op software gebaseerde oplossing die geheugenisolatie biedt voor low-end IoT-apparaten door middel van gedeeltelijke virtualisatie van de instructieset. De geheugenisolatie wordt vervolgens aangewend om de integriteit van kritische veilige operaties en geheime sleutels te garanderen, terwijl gelijktijdig onveilige gebruikersapplicaties uitgevoerd worden. Evaluatie laat zien dat voor normale gebruikspatronen van IoT-toepassingen vertragingen door virtualisatie onwaarneembaar zijn, terwijl het gebruik van systeembronnen klein is.

Contents

Abstract iii

Beknopte samenvatting v

Contents vii

List of Figures xiii

List of Tables xv

1 Introduction 1

1.1 Context 2

1.1.1 The IoT is resource constrained 2

1.1.2 The IoT is highly distributed 3

1.1.3 The IoT is dynamic 3

1.1.4 The IoT must be secured 4

1.2 Problem statement 5

1.3 Contributions 5

1.4 Overview of the thesis 7

2 Background 11

2.1 Software development abstractions 12
2.1.1 Network-oriented abstractions 12
2.1.2 Node-oriented abstractions 13
2.2 Runtime reconfiguration 14
2.2.1 Structural reconfiguration 15
2.2.2 Behavioural reconfiguration 16
2.2.3 Reflective component models 16
2.3 Security 18
2.3.1 Trusted Computing Base 18
2.3.2 Trusted embedded computing 19
2.3.3 Software-only solutions 19
2.4 LooCI: a reflective component model 20
2.4.1 The LooCI architecture 21
2.4.2 Reconfiguring a component-based application 22
2.4.3 SecLooCI 23
2.5 Barriers to adoption 23
2.5.1 Review of the state-of-the-art 24
2.5.2 Research opportunities 26
2.6 Summary 27

3 Safe reparametrization for distributed IoT applications 29

3.1 Introduction 30
3.2 Reparametrization in component compositions 30
3.2.1 Experiment description 31
3.2.2 Classes of component compositions 32
3.3 Prerequisites 32
3.3.1 Component model requirements 33

3.3.2 Component roles . . . 33
3.3.3 Constraints . . . 34
3.4 Design . . . 34
3.4.1 Language annotations . . . 35
3.4.2 Constraint propagation . . . 36
3.4.3 Constraint enforcement . . . 38
3.5 Implementation and evaluation . . . 39
3.5.1 Component size overhead . . . 40
3.5.2 Static middleware overhead . . . 40
3.5.3 Dynamic middleware overhead . . . 40
3.5.4 Network overhead . . . 41
3.5.5 Management overhead . . . 42
3.6 Summary . . . 43

4 Refraction: lowering the cost of reflection in the IoT 45

4.1 Introduction . . . 46
4.2 Reflection and its shortcomings in IoT systems . . . 47
4.3 Refraction in principle . . . 48
4.3.1 The refractive meta-model of RxCom . . . 50
4.3.2 Specifying refractive policies . . . 52
4.3.3 Specifying reactive policies . . . 53
4.4 Refraction in practice . . . 55
4.5 Evaluation . . . 57
4.5.1 Performance and overhead analysis . . . 58
4.5.2 Configuration repair case-study . . . 58
4.6 Summary . . . 63

5 Tomography: efficient regional reflection for the IoT 65

5.1 Introduction 66
5.2 Reflection on regions of compositions 67
5.3 Tomography 69
5.3.1 Regions of component compositions 69
5.3.2 Specifying regions 69
5.3.3 Using tomography 70
5.3.4 Probe propagation 71
5.4 Evaluation 73
5.4.1 Querying a region 73
5.4.2 Setting up a region 74
5.4.3 Management overhead 75
5.4.4 Smart lab case-study 75
5.5 Summary 77

6 Securing dynamic IoT systems **79**

6.1 Introduction 79
6.2 Design of SμV, the Security MicroVisor 81
6.2.1 The platform requirements of SμV 81
6.2.2 Architecture of SμV 82
6.2.3 SμV toolchain modifications 86
6.3 Remote attestation 87
6.4 Secure deployment 89
6.5 Implementation 90
6.5.1 Modified Harvard architecture 91
6.5.2 Bootloader 92
6.5.3 Initialization of data memory 92
6.5.4 Two word instructions 93

6.5.5 Remote attestation and secure deployment 93
6.6 Evaluation 94
6.6.1 Development overhead 95
6.6.2 Deployment overhead 95
6.6.3 Runtime overhead 98
6.7 Discussion 102
6.7.1 Promoting the adoption of MicroVisors 102
6.7.2 SµV and component-based middleware 103
6.8 Summary 104

7 Conclusion 105

7.1 Contributions in review 105
7.2 Future research and industrialisation 107
7.3 Outlook 109

Bibliography 111

List of publications 123

List of Figures

1.1 Contributions in the context of the IoT software stack 7

2.1 LooCI middleware architecture. 21

2.2 Example distributed component composition. 22

3.1 Classes of compositions with implicit parameter dependencies. . 33

3.2 Component roles and constraint types. 35

3.3 Step-by-step constraint resolution of component compositions . 37

3.4 Evaluation scenario: parameter dependencies in a smart office deployment. 39

3.5 Breakdown of average dynamic RAM overhead for each scenario. 41

4.1 Running example illustrating reflection. 47

4.2 Replacing reflective with refractive operations on the running example. 49

4.3 Formal specification of the basic RxCom meta-model. 50

4.4 Extension of RxCom's meta-model with *refractive* policies. . . 52

4.5 Extension of RxCom's meta-model with *reactive* policies. . . . 53

4.6 The LooCI architecture with RxCom extensions. 56

4.7 Evaluation scenario: configuration repair in a real-world smart lab. 59

4.8 Average latency of configuration repair for the evaluation scenario. 62

5.1 Running example illustrating reflection on regions of component compositions. 67
5.2 Querying the RFID coffee control region of the running example using tomography. 71
5.3 Splitting of tomography probes 72
5.4 Merging of tomography probes 72
5.5 The effect of component topology on tomography message overhead. 74
5.6 Evaluation scenario: smart lab component composition with regions. 76

6.1 Standard and SµV memory map 83
6.2 The modified toolchain for SµV 86
6.3 SµV memory map for modified Harvard architectures. 91
6.4 End-to-end secure deployment overhead. 98
6.5 Graph showing estimated battery life of applications 100

List of Tables

3.1 Number of messages sent over the network during reparametrization. 42

3.2 Number of commands issued by the user during reparametrization. 43

4.1 Performance overhead of aggregation and policy evaluation. . . 58

4.2 Network-wide messages per hour for 1 repair in the evaluation scenario. 62

5.1 Message overhead when querying the regions of the smart lab. . 77

6.1 Overhead of SµV on over-the-air binary image sizes. 96

6.2 Overhead of SµV on node-local load times. 97

6.3 Overhead of SµV on the execution times of the sample applications. 99

6.4 Flash memory utilization 102

Chapter 1

Introduction

The Internet of Things (IoT) can be defined as the interconnection of physical things to existing network infrastructure to enable cyber-physical services. Today, the IoT is moving out of the lab and into the real-world, where it is being applied at large scale in diverse application scenarios. Common examples include *smart homes*, where products such as Google Nest [70, 71] and Philips Hue [76] automate heating and lighting, *smart cities*, such as the Amsterdam Smart City initiative [4] and the SmartSantander project [86], where large-scale networks monitor the environment, transportation systems, waste management, and other community services, increasing efficiency and quality of life. The IoT is also critical to *Industry 4.0*, where the entire value chain is optimised by interconnecting physical processes with each other and the cloud, significantly reducing operational costs [53].

The number of IoT devices in the field has increased exponentially over the last few years, and projections indicate this trend will accelerate in the future. In 2017, 17.5 billion IoT devices were actively deployed [32]. By 2023, this number is expected to further increase to 31.6 billion devices. By definition, the IoT does not limit itself to one specific class of devices and promotes heterogeneous networks, however, the use of tiny battery-powered devices enables embedding processing and communication within the physical world at low cost and fine granularity [69]. These platforms are extremely resource constrained, often equipped with a low-power radio and run on a single battery charge for extended periods of time.

The rapid increase in IoT scale combined with the resource constraints of representative IoT platforms gives rise to a complex set of problems. This dissertation provides novel solutions to efficiently manage and secure large-scale

dynamic IoT networks built on resource constrained devices. The following sections analyse the characteristics and requirements of IoT applications, the key problems of the domain and the contributions made in this dissertation to address those problems.

1.1 Context

Building applications for IoT systems is notoriously difficult. IoT deployments have a unique properties that require special attention: i) IoT devices are resource constrained, ii) deployments are highly distributed, iii) applications are dynamic and require runtime reconfiguration, and lastly iv) the infrastructure and software running on it need to be secure.

1.1.1 The IoT is resource constrained

The current landscape of IoT hardware is extremely heterogeneous, with vast differences regarding system resources and capabilities. In the high-end of the spectrum are devices such as smartphones and tablets, which have ample memory in the GB range, processing power, fast 4G or WiFi network connections and a flexible energy budget with frequent battery recharges. The low-end is comprised by devices ranging from parking meters to thermostats, which are embedded with sensors, actuators and a microcontroller. These devices are extremely limited in resources: an IETF class 1 constrained device has ~10 KB of RAM, ~100 KB of flash and rudimentary low-power network capabilities with a bandwidth of less than 250 kilobit per second [12]. One of the biggest constraints, however, is that these devices often have to run on a single battery charge for years while deployed at large scale and in unreachable locations, making battery recharges or swaps unfeasible. This requires careful energy budgeting, mainly for tasks such as radio transmissions and computation [44].

Embedded resource constrained devices form the backbone of any IoT application, as they connect the physical world to the internet. The annual Ericsson Mobility Report shows us that wireless embedded devices currently account for more than half of the deployed IoT devices worldwide [32], a number that is bound to further increase as growth in the IoT install base is almost solely driven by them. Smartphones and tablets have an annual growth rate that has now stagnated between 0% and 3% percent, while small pervasive IoT devices have projected growth rates between 20% and 30% [32]. As a result, the importance of embedded devices and the resource constrains that they imply are an inherent part of designing any IoT platform or application.

1.1.2 The IoT is highly distributed

The IoT paradigm is inherently highly distributed: physical objects equipped with computational capabilities form a network together with back-end infrastructure, running distributed applications providing added benefit. These applications are characterized by the usual characteristics of distributed computing: concurrency, lack of global clock and independent failure of components. While contemporary middleware for IoT platforms has already taken these difficulties into account, support for recovering from software or hardware failures is still limited.

An added domain-specific difficulty is that IoT applications are often deployed at large scale and in inaccessible locations. A prime example are smart cities [33], where millions of devices embedded in streetlights, waste bins and parking meters are deployed over a city. The massive scale in combination with the impracticality to go in the field to fix problems requires that the software running on this distributed system has to be extremely scalable as well as robust.

1.1.3 The IoT is dynamic

Early IoT deployments consisted of simple single-purpose monitoring applications and devices deployed in the field ran a static software image with dedicated application logic for their entire useful life [68]. Software was preinstalled on the devices and remote reconfiguration or updating of the application after physical deployment was not possible.

As IoT applications become more complex and dynamic with requirements that change during their life cycle, the need for dynamic system reconfiguration rises. A trend further increasing dynamism in IoT is the advent of shared sensing infrastructures such as Smart Cities [33] and Smart Offices [55]. IoT deployments are evolving to become more open and multi-purpose [83, 49]. A single IoT infrastructure can host multiple applications at the same time that are managed by multiple actors, improving the return on investment as the cost of deploying and purchasing hardware can be spread over multiple applications, each generating their own independent value.

Handling such dynamism requires fundamental support for software evolution in the underlying operating system and middleware layers of the software stack, enabling applications to be reconfigured and updated at runtime while sharing the same physical devices concurrently. Furthermore, as IoT devices are often deployed at large scale in inaccessible places, it is important reconfiguration can

be enacted remotely. While contemporary operating systems and middleware have made advances in promoting remote reconfiguration in IoT, they have yet to result in commonly used platforms and paradigms [17, 96, 18, 47].

A largely untackled problem is the general practicality of managing a large IoT deployment remotely with multiple actors. Remote reconfiguration in its current form is a manual process, where incompatible reconfigurations or operator error can cause breakage and down-time of the IoT network. Furthermore, introspection and modification of configuration incur a considerable message overhead and energy cost, impacting battery life.

1.1.4 The IoT must be secured

Malware is a critical and growing threat to the IoT. The StuxNet [54] worm was the first high-profile example. StuxNet damaged an Iranian uranium enrichment facility by spoofing rotation sensor data from an enriching centrifuge, causing a motor actuator to increase its speed until the centrifuges it controlled were destroyed. More recently, the Mirai malware used IoT devices to create a botnet that mounted a massive scale Denial of Service (DoS) attack, which peaked at over 1 Terabit per second (Tbps) [103].

Despite the clear danger, the vast majority of deployed IoT products provide little or no protection against malware and even basic features such as memory protection, are typically not available on constrained IoT devices. Ronen et al. recently showcased these weaknesses through the creation of a rapidly spreading worm for the Philips Hue smart light bulbs [84].

Furthermore, adding advanced reconfiguration capabilities to support dynamism further increases attack surface for IoT devices. The ability to remotely reconfigure and deploy software has to be effectively secured to avoid misuse.

A common way to add security to these inherently insecure devices, is by either adding a trusted piece of hardware like a TPM to perform secure operations, or by modifying the microcontroller to provide security primitives or the memory isolation required to implement them in software. Both solutions rely on extensive hardware modifications, and do not provide a solution for the millions of devices already deployed in the field. Securing these devices through a hardware solution is practically infeasible and extremely costly.

1.2 Problem statement

While the added value of a far-reaching integration of IoT technology in our society is apparent today, efficiently configuring and securing these devices is not straightforward. This dissertation focuses on providing generally applicable solutions and paradigms to solve these problems, while considering the extreme resource constraints of typical IoT platforms.

More formally, this relates to the following problem statements:

- *How to reconfigure distributed application logic of an IoT infrastructure in a multi-user environment without causing configuration conflicts and downtime?*
- *How to reduce the management and resource overhead associated with runtime reconfiguration of large scale IoT networks?*
- *How to cost effectively secure representative constrained IoT platforms without adding hardware?*

Finding appropriate solutions for these problems can be challenging, considering the characteristics of IoT deployments outlined in previous sections: limited system resources, a tight energy budget, low-bandwidth radios, increasing levels of software dynamism and large-scale networks. As a result, reusing pre-existing solutions from traditional IT systems is suboptimal. When solving these problems, a trade-off has to be made between functionality and resource consumption.

1.3 Contributions

This dissertation provides three key contributions to the state-of-the-art in the IoT, which address the challenges laid out in the previous section.

1. A methodology detecting and avoiding incompatible distributed configuration of an IoT application, providing safe runtime reconfiguration in a dynamic multi-user environment: *Safe reparametrization*.
2. Novel approaches that lower the cost and complexity of inspecting and reconfiguring IoT deployments at runtime: *Refraction* and *Tomography*.
3. The *Security MicroVisor* (SμV), a software security architecture that provides memory isolation and a secure execution environment on low-end

microcontrollers. Two secure operations are showcased: *remote attestation* and *secure deployment*.

This work builds on existing software solutions that support dynamism in the IoT, more specifically reconfigurable component-based middleware. Component-based middleware has proven to be a promising solution for managing the complexity of developing IoT applications [49]. Examples of runtime reconfigurable component middleware include OpenCOM [18], RUNES [17], OSGi [82], REMORA [96] and LooCI [47]. These systems provide the capabilities required to manage component life-cycle, configuration, introspection, and assembly at runtime. An essential feature of component-based IoT infrastructures is software reuse, where one component can offer functionality to multiple applications which are formed by component compositions, promoting resource sharing and code reuse.

While contemporary component-based systems offer dynamic runtime reconfiguration and code reuse, the problems outlined in the previous paragraph still remain. Misconfiguration due to implicit configuration parameter dependencies or operator error are commonplace, causing down-time and loss of data. Introspection and reconfiguration of component compositions are still very inefficient, causing significant network overhead and battery drain. Lastly, the remote reconfiguration and software deployment features offered by middleware is inherently insecure and open the IoT platform to malware. The contributions presented in this dissertation focus on providing a solution for these problems while building on the merits of component-based middleware.

Safe reparametrization is the first contribution, offering composition-safe reparametrization of applications. This is accomplished by offering language annotations that allow component developers to make dependencies explicit and network protocols to resolve and enforce parameter constraints. As a result, misconfiguration breaking running applications is avoided, and reconfiguration is greatly simplified while imposing minimal runtime overhead.

Refraction lowers the runtime cost of introspection and reconfiguration by selectively augmenting application data flows with their reflective meta-data, which travels at low cost to refractive pools, serving as loci of inspection and control for the distributed application. Reactive policies are introduced, providing a mechanism to trigger reconfigurations based on incoming reflective meta-data. *Tomography* further reduces the runtime overhead of reconfiguration by reimagining the visitor design pattern for distributed component compositions. *Tomography* reduces both the number of explicit queries and the volume of network messages, significantly reducing management effort and energy consumption.

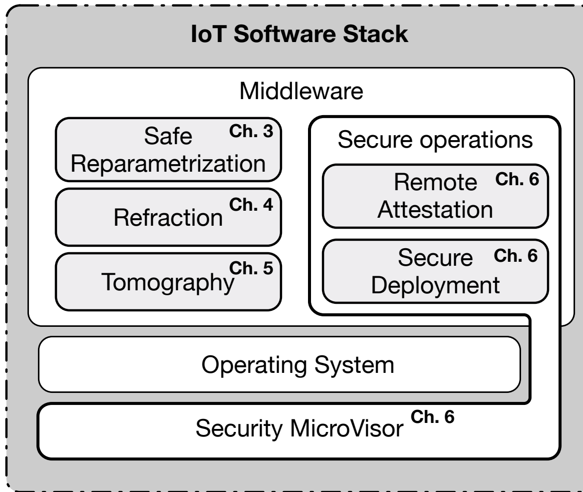


Figure 1.1: Contributions in the context of the IoT software stack

A last contribution is the *Security MicroVisor* (SμV). SμV is a software-only solution that provides memory isolation for low-end microcontrollers by selectively virtualizing machine instructions. The memory isolation provided by SμV is used to implement two key security features: *remote attestation* and *secure deployment*. These features guarantee the integrity of the operating system, middleware and applications running on the microcontroller and functions as an anti-malware mechanism, even when reconfigurable software engineering techniques are applied.

1.4 Overview of the thesis

The remainder of this dissertation is structured as follows:

Chapter 2 - Background The background chapter reviews work that supports or relates to the body of work presented in this dissertation. More specifically, relevant software development methodologies, reconfiguration frameworks, component-based middleware and low-level security architectures for IoT devices are surveyed. The chapter concludes with an analysis of the barriers to adoption of IoT technology.

Chapter 3 - Safe reparametrization for distributed IoT applications This chapter focuses on the first contribution of this dissertation: *Safe reparametrization* for component-based IoT systems with a decentralized configuration space. This chapter builds and expands on work presented in the following papers:

Daniels, W., del Cid Garcia, P. J., Hughes, D., Michiels, S., Blondia, C., and Joosen, W. Composition-safe Re-parametrization in Distributed Component-based WSN Applications. In *IEEE 12th International Symposium on Network Computing and Applications (NCA)* (aug 2013), IEEE, pp. 153–156

Daniels, W., del Cid Garcia, P. J., Joosen, W., and Hughes, D. Safe Reparametrization of Component-Based WSNs. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services (MobiQuitous)* (2014), Springer International Publishing, pp. 524–536

Chapter 4 - Refraction: lowering the cost of reflection in the IoT This chapter presents *Refraction*, a solution that selectively centralizes the meta-model of distributed IoT systems and allows for automatic reconfiguration through a rule-based language. This system directly translates to the second contribution of this dissertation: to reduce the costs and complexity querying and reconfiguring running IoT applications. This work was previously presented in the following paper:

Daniels, W., Proença, J., Clarke, D., Joosen, W., and Hughes, D. Refraction: Low-Cost Management of Reflective Meta-Data in Pervasive Component-Based Applications. In *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)* (New York, New York, USA, 2015), ACM Press, pp. 27–36

Chapter 5 - Tomography: efficient regional reflection for the IoT This chapter introduces *Tomography*, a novel way to efficiently query and reconfigure distributed IoT systems by launching probe packets into a network, progressively collecting data and enacting change when visiting devices on its path. Tomography is part of the second contribution of reducing the runtime overhead and complexity of managing IoT applications. Tomography was first discussed in the following paper:

Daniels, W., Proença, J., Matthys, N., Joosen, W., and Hughes, D. Tomography: Lowering Management Overhead for Distributed Component-Based Applications. In *Proceedings of the 2nd Workshop on Middleware for Context-Aware Applications in the IoT (M4IoT)* (New York, New York, USA, 2015), ACM Press, pp. 13–18

Chapter 6 - Securing dynamic IoT systems This chapter describes a software security architecture for embedded IoT devices: the *Security MicroVisor* (SµV). SµV secures microcontrollers by partially virtualizing instructions, thereby restricting unauthorized and unsafe operations. This effectively provides memory isolation for the low-end devices that dominate IoT deployments. Two secure operations are implemented on top of SµV: *remote attestation* and *secure deployment*. These security primitives can be used to guarantee the integrity of running applications. This work was initially published in the following paper:

Daniels, W., Hughes, D., Ammar, M., Crispo, B., Matthys, N., and Joosen, W. SμV - the Security MicroVisor: a Virtualisation-based Security Middleware for the Internet of Things. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track* (New York, New York, USA, 2017), ACM Press, pp. 36–42

Chapter 7 - Conclusion The final chapter concludes the dissertation, discusses how elements of the thesis integrate with past and present academic and industrial work, and provides an outlook on opportunities for future research.

Chapter 2

Background

Early Internet of Things deployments relied on custom firmware images, realised as static domain-specific software that closely integrated with the deepest layers of the software stack and required detailed knowledge of hardware platforms. The high complexity and inflexibility of developing software like this severely hampered early uptake of IoT technology. With modern deployments spanning cities and devices lasting up to 10 years on a single battery charge, the IoT is continuously pushed to be more agile and dynamic, moving from static software to supporting multiple concurrent applications and users throughout a network's life cycle. This calls for appropriate programming models and runtime reconfiguration to simplify the development of distributed applications and to efficiently manage them in the field. Securing the IoT is an orthogonal concern that will grow in importance as IoT technology permeates our society. The IoT security landscape is also likely to grow more complex, as we move from static to dynamic software. This chapter gives an overview of existing technologies that tackle these challenges and upon which the contributions of this dissertation are built.

The remainder of this chapter is structured as follows. Section 2.1 looks at relevant software development abstractions and programming approaches that facilitate building distributed IoT applications. Section 2.2 surveys solutions that enable software evolution and dynamism through runtime reconfiguration. Section 2.3 gives an overview of related work securing the IoT through trusted computing. Section 2.4 showcases LooCI, a state-of-the-art reconfigurable component model. Finally, Section 2.5 analyzes the shortcomings of contemporary IoT technology, and identifies the research gap that this dissertation aims to fill.

2.1 Software development abstractions

Efficient development support for distributed IoT applications is essential for IoT adoption. Many real-world IoT deployments still run software that is monolithic and tightly coupled with low-level system software, thereby obfuscating core application logic and increasing complexity [69]. This section discusses relevant state-of-the-art software development abstractions that address this problem by reducing complexity and promoting dynamism and flexibility. These programming models can be split up in network and node-oriented abstractions.

2.1.1 Network-oriented abstractions

A first group of programming models attempt to mitigate the complexity of distribution by treating an IoT network as a single (distributed) computer for which the developer writes software. Network-oriented abstractions are motivated by the notion that micromanaging each node individually is not be feasible for large networks and that addressing them as a group reduces this overhead.

In the most simplistic models, IoT devices sample data periodically and forward it to a gateway. The gateway will act as a data sink for the IoT devices, and at the same time offer APIs to query and process the data. In this model, all application logic resides at the gateway. Examples of this approach are Janus [29], Hourglass [92], Global Sensor Networks (GSN) [1] and the IBM edge servers [85]. A similar group of solutions abstract the network away behind a database interface. Applications can interact with the devices in the network by querying the distributed database. TinyDB [61], DSWare [56] and Cougar [112] exemplify this approach.

Macro-programming approaches such as Kairos [40] and Rulecaster [10] propose new high-level language constructs to define applications running over groups of IoT devices. During deployment, the application is automatically split into tasks and assigned to the nodes.

Neighbourhood programming abstractions such as Hood [110] and Abstract Regions [108] introduce a programming model for groups of nodes in a sensor network. A number of general-purpose primitives are provided for node addressing, data sharing and data reduction in local regions of the network. The goal of this new programming model is to alleviate development overhead for distributed sensing applications.

The main goal of contemporary network-oriented approaches is translating high-level global goals to low-level individual goals. A core difficulty that limits

the application of these solutions is the high level of abstraction, often to the point of a single application objective. The result is a reduction of fine-grained local control that may be necessary for more complex applications.

2.1.2 Node-oriented abstractions

Another category of software development methodologies focus on implementing applications from the perspective of the individual nodes in a network. The main goal of these approaches is to provide applications with modularity and separation from underlaying system logic at the single node level, while facilitating the creation and management of distributed relationships.

Contiki [30] is a modular IoT operating system that is realized in C. Contiki introduces the notion of application 'modules', which are coarse-grained units of software functionality that can be dynamically replaced at runtime, allowing a node to host multiple independently updatable applications. Contiki reduces the overhead of small-scale changes in comparison to monolithic programming models. While this approach offers more efficient support for software evolution, Contiki modules do not make their software dependencies externally visible and therefore it is not safe to deploy them without a complete understanding of the distributed applications in which they will be used.

Virtual machines (VMs) go one step further by adding an extra abstraction layer. While the core runtime environment is native code, application modules are compiled to intermediate byte-code and are interpreted at runtime. The primary advantages of this approach are portability between platforms and more compact modules. Byte-code is more expressive then native code and allows for a more concise representation of application logic. Disadvantages are that interpretation of byte-code can impose a significant performance overhead compared to native execution.

A plethora of solutions focus on bringing the object-oriented Java programming language to IoT platforms by implementing a Java VM (JVM). Squawk [93], Darjeeling [14] and CerberOS [2] are examples of this. Squawk provides the full Java Micro Edition compliant Squawk JVM running on a platform with a more powerful and power-hungry processor (180 MHz 32bit ARM core), making it less useful for applications requiring long battery lifetimes. JVMs for low-power 8-bit microcontrollers are more simplistic, offering a partial implementation of Java. Darjeeling and CerberOS are prime examples, with CerberOS adding the enforcement of resource utilization policies for application modules.

Lastly, component-based software engineering (CBSE) further refines modularization by moving from arbitrary software modules to components with

clearly defined *provided* and *required* interfaces. A full (distributed) application can be built by creating a composition of these components. The strengths of CBSE are a clear separation of concerns, reduction of coupling and the ease with which components and code can be reused in different applications. Implementations of CBSE for general computing include the CORBA [104] component model, focusing on interoperability and distribution as well, and OSGi [73], a Java-based component-oriented system.

NesC [36] was the first component-based IoT programming model and was used to build the TinyOS operating system [41]. NesC allows developers to compose applications from generic and re-usable software building blocks. At compile time, NesC performs whole-program optimization and produces a monolithic binary that includes both application and OS components. As NesC code is compiled to a single binary file, it is not possible to replace individual components at runtime and therefore runtime adaptation requires the replacement of the entire system image.

GridKit [46] was the first runtime reconfigurable component-based middleware for IoT and was implemented using the OpenCOM [18] component model, to which it adds support for building distributed relationships using the Open Overlays [39] pattern. RUNES [17] is a component-based middleware that supports the creation of reconfigurable application compositions on resource constrained devices, however, it is a local component model which provides no support for the creation of distributed relationships. REMORA [96] provides a C-like programming language to specify component interfaces and application compositions. At runtime these components execute on the REMORA platform abstraction layer, providing a uniform interface to low-level hardware functionality without requiring platform specific knowledge. LooCI [47] is an IoT middleware which provides a reconfigurable component model, loosely coupled binding model and a platform independent execution environment. LooCI explicitly supports resource constrained hardware and includes distribution support as a first class concern, making it is possible to reify and reconfigure distributed relationships at runtime.

2.2 Runtime reconfiguration

In parallel to streamlining the development of applications through appropriate programming abstractions, numerous developments towards optimizing runtime management have been proposed in literature. With hardware getting more energy-efficient, batteries holding more charge and IoT deployments growing ever larger, large-scale sensor deployments lasting up to ten years are a reality.

Manually reprogramming these hard-to-reach IoT devices in the field throughout their software life cycle is unfeasible, calling for solutions that allow runtime reconfiguration. State-of-the-art solutions in this field can be categorized by the scope of the reconfiguration: structural reconfiguration replaces larger chunks of functionality, while behavioural reconfiguration is more fine-grained and merely modifies the behaviour of existing applications through parameters [8].

Reconfiguring a resource constrained IoT device comes at a certain energy cost. Radio transceivers are the primary energy consumers on contemporary IoT devices [44]. As a result, reconfiguration by means of a full monolithic image update will have a much larger impact on battery life than changing a single parameter. Another important metric is the flexibility of a reconfiguration solution: a full image update can enact an arbitrary amount of change, while the modification of a parameter is more limited in that regard.

2.2.1 Structural reconfiguration

The most flexible structural reconfiguration solution is the use of *monolithic images* to replace all software on a device. A monolithic image is a single binary file that contains the operating system, drivers, middleware and applications linked together. Because of this, any aspect of the running software can be changed. While reconfiguring devices this way is extremely flexible, the size of monolithic images causes difficulties with distribution and has a big energy impact. A seminal example of monolithic reconfiguration is Deluge [48]. Deluge extends TinyOS [41] with a dissemination protocol for transmitting system images to all nodes in a multi-hop network reliably. A more recent example is the firmware update capability proposed for Mbed OS [51], an industry leading IoT operating system developed by Arm and its partners. Here multicast is used to disseminate a fragmented monolithic image. Packet loss is mitigated by transmitting redundant fragments with error correction data. A full firmware image can be reconstructed node-local using the extra fragments and an error-correction algorithm.

A more fine-grained approach to structural reconfiguration is loading smaller *software modules* containing native code and linking them at runtime to the operating system and libraries present on the device. This way, only functionality that needs to be modified will be transmitted over the network and replaced. Contiki [30] is a prime example of modular reconfiguration. Contiki relies on a compact version of the Executable and Linking Format (ELF) [28] object format to provide loading, relocation and linking of new application modules at runtime. The embedded variant of the LooCI [47] reconfigurable component middleware is built on the Contiki OS and uses its ELF loader to dynamically

load new components. As discussed in Section 2.1, an added advantage here is that the updated functionality is clearly delineated by means of a component with declared interfaces.

Virtual Machine-based approaches such as Squawk [93] and Darjeeling [14] similarly offer reconfiguration through modular updates. The primary advantage when compared to native software modules is that any interaction with the preinstalled functionality is provided by the JVM. This avoids relocation and linking of code and data, thereby drastically simplifying the loading process. In terms of flexibility, interpreted languages tend to be more limited then their native equivalents, as they tend to have incomplete VM implementations and abstract away lower-level hardware control which is needed in some cases [14, 2].

2.2.2 Behavioural reconfiguration

The simplest form of reconfiguration is behavioural reconfiguration through *parameters*. Behavioural reconfiguration relies on the application developers to expose parameters that can be modified at runtime and change the behaviour of running software. A simple example would be changing a *sample rate* parameter on an IoT device measuring temperature. In this case it allows runtime modification of the sample rate at which the sensor is sampled. From this example it is apparent that parameter reconfiguration incurs a very low network overhead but is extremely limited in what changes can be enacted.

Generally speaking, any IoT device with bi-directional communication is able to perform behavioral reconfiguration through parameters as long as the software developer explicitly exposes them. However, some platforms incorporate support for parameters directly in their runtime environment. The LooCI [47] reconfigurable component model allows developers to define and externalize component parameters that can be used for runtime reconfiguration through a dedicated API. Sensor Network Management System (SNMS) [100] similarly extends TinyOS components with explicit support for parametrization, and has a special focus on distribution protocols to manage parameters over a multi-hop network.

2.2.3 Reflective component models

Component models take above reconfiguration approaches further by systemizing them through a reflective meta-model. Reflection is the capacity of a software system to inspect itself *(i.e. introspection)* and modify its structure, properties and behaviour at runtime *(i.e. reconfiguration)* [94].

The application of reflective programming techniques to distributed component-based systems was pioneered by Blair et al. [11], who advocated for a systematic approach to reflection based on a per-component *meta-model* in which selected elements of the component implementation are reified. This meta-model externalises elements of the component implementation, such as its incoming and outgoing connections and its parameters. The meta-model is causally connected to the component implementation, which allows it to support both *introspection* (reading the meta-model) and *reconfiguration* (modifying the meta-model). The use of a per-component meta-model ensures that the scope of reconfiguration actions is bounded. Typical actions are adding and removing components, changing the connections between them and modifying their properties. Distributed applications are built by connecting components and forming a component composition.

The meta-model provides a well-defined way of introspecting and reconfiguring components, and is backed up by structural and behavioural reconfiguration approaches listed above. For example, adding a new component to a device requires the loading of a software module (*structural reconfiguration*), while modifying connections between components and changing their properties is similar to reconfiguration using parameters (*behavioural reconfiguration*).

OpenCOM [18] is a generic reflective component model that has been applied to build pervasive sensing applications [46]. OpenCOM is platform and language independent and supports runtime reconfiguration and introspection of the component meta-model. The local OpenCOM component model can be extended with binding model plug-ins to support distributed application composition. RUNES [17] brings OpenCOM-like functionality to resource constrained systems.

LooCI [47], as described in detail in Section 2.4, is a language and platform independent component model designed to support distributed IoT applications. LooCI supports both introspection and reconfiguration at runtime. In contrast to the models discussed above, which need extensions to support distribution, the LooCI runtime inherently supports the creation of distributed component bindings.

While the merits of reflective component models are apparent, when reconfiguring distributed applications it is frequently necessary to work in a coordinated fashion with the meta-model of multiple distributed components. This leads to increased development complexity and message passing overhead.

2.3 Security

Previous sections have focused on various technologies that enable dynamism and runtime adaptation on resource constrained IoT platforms. While these developments are a natural evolution and a necessity to drive adoption of the IoT, the increased complexity of both middleware and underlaying operating systems increase attack surface significantly. More specifically, platform facilities such as remote software deployment and reconfiguration pose a threat if left unsecured and are primary attack vectors for malware and tampering. This section takes a look at relevant work that focuses on securing low-cost and resource constrained embedded devices, ensuring they are uncompromised by attackers and behave as intended.

2.3.1 Trusted Computing Base

A Trusted Computing Base (TCB) is a combination of hardware and software that forms the foundation of any security policy on a platform [102]. The goal of a TCB is protection against system-level attacks, and to provide a trusted base upon which security related functionality can be built.

Early TCBs utilize a static chain of trusted integrity checks, such as secure boot [5, 74] and Trusted Platform Modules (TPMs) [101]. Secure boot relies on immutable bootstrap code with hard-coded public keys to establish a chain of trust during the boot process. TPMs on the other hand are secure crypto co-processors isolated from the main processor, embedding cryptographic keys and providing authentication, attestation and encryption.

An important concept in trusted computing is remote attestation [34]. Remote attestation allows an external verifier to detect unauthorized changes and tampering to devices in the field and guarantees platform integrity at runtime. The TPM standard for example specifies Trusted Execution Technology [101] for this purpose.

While nowadays TCBs are quite common in more powerful commodity hardware (smartphones, laptops), it is a relatively new technology for the embedded devices prevalent in the IoT. Existing solutions such as the industry standard TPM are often considered too complex, expensive and power-hungry for low-end embedded devices.

2.3.2 Trusted embedded computing

To cover this end of the spectrum, a new wave of solutions have been proposed for low-end microcontrollers, some of which provide much more than just a TCB. The most relevant are SMART [31], SANCUS [72] and TyTAN [13].

SMART [31] is a hybrid hardware-software approach. Hardware modifications to the memory bus addressing logic of the embedded microcontroller guarantee isolation of code and data belonging to critical secure operations. SMART specifically focuses on providing remote attestation. A symmetric key is stored in the isolated data memory, while the isolated instruction memory contains routines for computing a message authentication code (MAC) of the entire state of the device. The result is compared by an external verifier with the MAC computed over the expected state, detecting any deviation or tampering on the device.

SANCUS [72] is a full-featured security architecture for IoT devices, relying on a pure hardware root of trust. SANCUS provides not only application isolation, but also integrity, authentication, and dynamic remote attestation. SANCUS operates by modifying the Memory Access Logic (MAL) circuit of an MSP430 processor, enforcing access rights for software modules to achieve isolation and key protection. The result is a full TCB on a microcontroller. Similarly to SMART, SANCUS relies on computationally less complex symmetric key cryptography in order to minimize hardware costs.

TyTAN [13] extends the Trustlite architecture [52] with dynamic loading, and both local and remote attestation guarantees for isolated software modules from mutually untrusted stakeholders. It is based on an Execution-aware Memory Protection Unit (EA-MPU), a hardware component that provides memory access control enforcement based on the identity of code that attempts to access a data region. Compared to SANCUS and SMART, TyTAN adds interruptibility of the attestation process, a property that is often required in hard real-time applications.

2.3.3 Software-only solutions

While SMART, SANCUS and TyTAN have much lower hardware requirements than a TPM, they are still costly to implement. Millions of devices already deployed would require hardware modification or replacement to implement these solutions. As a result, there is a stream of research that focuses on bringing software-based remote attestation to resource constrained devices *without* the use of a full TCB.

SWATT [89] is a time-based attestation technique that relies on response timing to identify compromised nodes. A similar concept is used in many other software-based solutions [57, 58, 87, 88]. These methods rely on the estimated upper-bound time required by a given configuration of the device to freshly compute the correct answer for the verifier. If the computation takes longer, then the presence of an attacker can be inferred. The inherent limitation of time-based assumptions have been discussed in the literature [90] and several concrete attacks have also been published [15]. Therefore, these approaches are not considered to be secure.

PoSE [75] uses the bounded amount of storage in the target platform as a means of providing software-based remote attestation. PoSE can only remotely attest the state of a device in conjunction with a software update. During the update process, a stream of random data is sent along with the new software, combined together exactly the size of the entire memory of the target device. Next, a MAC is computed over the full memory contents of the device using a fresh key included in the payload. The MAC is sent back to the verifier/updater, and if correct the state of the device (i.e. every byte in the memory) is proved. Two obvious drawbacks of this approach are: i) the lack of authentication, any adversary with access to the network can install new software, and ii) the high network overhead and associated battery life impact incurred by sending large amounts of random data over a low-power wireless network.

2.4 LooCI: a reflective component model

This last section takes a closer look at a state-of-the-art component-based middleware. Earlier in this chapter, Section 2.1 underlined the advantages of componentization in the software development stages. Component-based software engineering increases code-reuse and decreases coupling by encapsulating functionality in components with clearly defined interfaces and building applications through component composition. Section 2.2 further highlighted how these systems can be reconfigured with reflection, which extends component-based systems with the capability to do runtime reconfiguration and introspection through a meta-model.

In order to better understand how all these technologies work together, this section takes an in-depth look at an existing reflective component model: LooCI, the Loosely-coupled Component Infrastructure [47]. First, the architectural components that make up LooCI are explained. Next, the process of building and managing distributed applications with LooCI is illustrated through a

running example. Lastly, SecLooCI [62] is discussed, an extension of the LooCI component model that focuses on security.

2.4.1 The LooCI architecture

Figure 2.1 shows the layered architecture of the LooCI middleware. LooCI provides a platform-independent execution environment for various *underlaying platforms*, such as Contiki (embedded), Squawk (embedded VM), OSGi (back-end) and Android (smartphones). The Network Framework abstracts any platform specific network interaction away and provides a uniform interface for the upper layers. While the APIs are consistent between platforms, components are still written in the language best suited for the target platform and compiled natively to reduce runtime overhead.

The distribution of application data both locally or remotely is handled transparently by the Event Manager, which connect all components across nodes through a distributed event bus. LooCI utilizes an asynchronous, event-

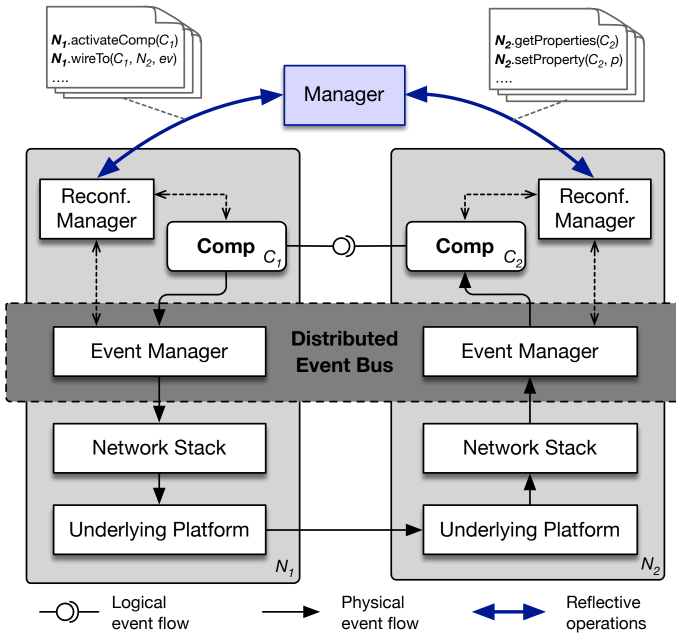


Figure 2.1: LooCI middleware architecture.

based interaction model that requires components with matching event types to be explicitly wired together through bindings.

The Reconfiguration Manager handles remote introspection and reconfiguration of locally deployed components (i.e. activation, changing of properties, ...), and manages any local or remote component bindings. Figure 2.1 shows how an external Manager interacts with the Reconfiguration Manager module residing on the devices, using reflective operations to introspect and reconfigure deployed components and bindings.

2.4.2 Reconfiguring a component-based application

Figure 2.2 shows an example LooCI software composition that is used in a real-world IoT deployment for detecting motion in a room. In this example, two Motion Detector components —residing on N_1 and N_2—transmit their sensed data to a Motion Aggregator component on N_3, that aggregates motion readings and forwards processed data to a Motion Reporter component. Motion Reporter resides on a resource rich node N_4 and pushes the data to a web platform for viewing. Gray boxes represent computational platforms, where components are deployed and executed. Software components are shown as white boxes with solid black lines, which publish values via their *provided interfaces* (—o), and receive values from via their *required interfaces* (—( ). Components also have key-value pairs of properties that can be used to parametrize their behaviour.

The application is controlled by an external Manager entity, which issues *reflective operations* to the component model kernel running on each node in order to *introspect* the software system, perform *structural reconfiguration* by connecting or disconnecting interfaces, and *behavioural reconfiguration* by modifying component properties. The interaction between the Manager and the reflective middleware is depicted with thicker arrows in Figure 2.2. Examples of reflective operations that can be used by the Manager are shown in Listing 2.1.

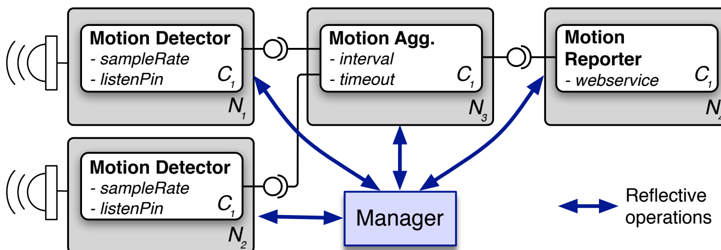


Figure 2.2: Example distributed component composition.

```
1  // introspection operations
2  N1.getProperties(C1)
3  N1.getProperty(C1,sampleRate)
4  N3.getWiresFrom(C1)
5  // reconfiguration operations
6  N3.setProperty(C1,timeout=30s)
7  N4.activateComponent(C1)
8  N3.wireTo(C1,N4,motionEvent)
9  N4.wireFrom(N3,C1,C1,motionEvent)
```

Listing 2.1: Example of reflective operations on the example.

2.4.3 SecLooCI

SecLooCI [62] extends the LooCI middleware with security features aimed at multi-user and multi-application environments. SecLooCI augments reflective operations with role-based access control, provides secure deployment of new components and encrypts application data. SecLooCI makes use of well-known cryptographic primitives relying on symmetric pre-shared keys to enforce access control.

The primary weakness of this approach is the lack of a Trusted Computing Base. Among other hardware platforms, SecLooCI also targets an 8-bit microcontroller without memory isolation. Crucial security routines and secret key-material reside in the main memory alongside remotely deployed components. Any compromised component executing on the platform is able to read or modify keys and routines, hereby bypassing the security features offered by SecLooCI. As discussed in Section 2.3, by isolating security-critical functionality and data inside a TCB, these problems can be avoided.

2.5 Barriers to adoption

Previous sections surveyed a range of state-of-the-art solutions focusing on solving challenges associated with building, managing and securing distributed software for the IoT.

This section takes a critical look at their strengths and weaknesses, and identifies gaps that must be addressed to ensure their adoption.

2.5.1 Review of the state-of-the-art

The analysis starts with reviewing how the state-of-the-art caters to simplifying development, life-cycle management and security in the IoT.

Simplifying development and runtime configuration

While previous sections introduced software development abstractions and reconfiguration systems separately, it is clear that often these go hand in hand to create holistic solutions in the shape of operating systems, virtual machines and component models.

Two prominent operating systems for the IoT are Contiki [30], and TinyOS [41]. Both support run-time reconfiguration, but do so in different ways. TinyOS is more limited: while it is modular in the development phase, it specializes the effective dissemination of full-image updates to devices in the field. Contiki on the other hand allows developers to create arbitrary modules which are over-the-air deployable. Both provide no inherent support for additional reconfiguration or distributed interactions, requiring the application developer to implement their own non-standard solutions.

Solutions based on a JVM like Squawk [93] offer similar functionality by compiling application modules to intermediary Java-byte code. Application modules can then be remotely deployed and executed. Support is provided for local interaction between modules, yet remote interaction and finer-grained remote reconfiguration needs to be provided by the developer.

The biggest limitation of both OS and VM solutions is that while they allow modular software development, they do not offer software paradigms to enable the specification of meta-data that defines and bounds reconfiguration. Reflective component models improve on this by providing developers with the means to explicitly declare required and provided interfaces, as well as the type and properties of a component through a meta-model. By externalizing these aspects during development, introspection and reconfiguration is well-defined and standardized.

Representative reflective component models include RUNES [17], REMORA [96] and LooCI [47]. Reflective component models have introduced unprecedented flexibility for developing and managing IoT applications, certainly when compared to the earliest systems that were built and deployed statically without reconfiguration in mind. Advantages are low coupling and high code-reuse due to the componentization of applications, and fine-grained runtime reconfiguration through reflection. All three approaches use dynamically loadable modules

to remotely deploy new components. Explicitly defined interfaces allow the composition of components to build applications. LooCI differentiates itself in this regard by allowing remote bindings through a distributed event bus, while RUNES and REMORA only support local bindings.

While the advantages are clear, contemporary component models come with a new set of problems. Considering the example shown in Figure 2.2, two shortcomings of reflection quickly become evident. Firstly, reflection requires the transmission of many messages to query and reconfigure remote components. This is very problematic, as research [44] has shown that radio transmissions are the primary source of energy consumption for IoT devices. Secondly, writing reflective code for distributed management, as shown in Listing 4.1 is complex and error-prone. Errors in the configuration of a single component can make the distributed application malfunction and are extremely difficult to track down. This issue becomes more severe if multiple users manage a large-scale shared deployment and therefore use reflective APIs in parallel.

Security in the IoT

As the IoT takes a prominent place in our society, we rely more and more on the correct and secure functioning of embedded devices. The developments discussed in the paragraph above indicate a future of open and dynamic systems, however, features such as in-depth remote reconfiguration provide larger attack surfaces for adversaries.

In order to secure critical secure operations such as encryption of application data and authentication of reconfiguration, it is essential to utilise a Trusted Computing Base as a root of trust.

For resource constrained devices, the most relevant solutions are SMART [31], SANCUS [72] and TyTAN [13]. All these solutions rely on modifying 8-bit microcontrollers on a hardware level to provide functionality like isolation and memory protection. These hardware-based solutions have a proven track record, but securing the millions of devices out in the field today would require expensive modification or replacement of hardware.

This problem has long been discussed in literature, and solutions attempting to provide secure primitives such as *Remote attestation* without a hardware TCB have been proposed. The most prominent ones are SWATT [89] and PoSE [75], which attempt to provide attestation by relying on physical deterministic properties of the microcontroller, such as time required for execution and storage space. Not only are these approaches extremely limited due to the lack of a TCB, they also are insecure [15] or impractical [90].

2.5.2 Research opportunities

From the in-depth analysis of the state-of-the-art in the previous section, four barriers to adoption are apparent. The first three specifically focus on reconfiguration and management of component-based models, as these provide the basic dynamism and flexibility contemporary deployments need. The last point focuses on security for resource constrained devices in general.

Achieving consistent reconfiguration is too complex While reflective component-models offer flexible and well-defined reconfiguration, the amount of configuration parameters in larger compositions in conjunction with the distributed nature of the meta-model in a multi-user environment causes inconsistencies in configuration, which are hard to track down and cause downtime.

The overhead of control is too high When reconfiguring distributed applications, it is frequently necessary to work in a coordinated fashion with the meta-model of multiple distributed components. This significantly increases overhead in terms of configuration messages passed, energy spent and the time required to enact change.

The overhead of inspection is too high Introspection of a distributed component-composition is often required to gather a full overview of an IoT application. Because the meta-model is distributed over all participating devices, the network overhead and energy spent to carry out introspection is substantial.

Security is expensive and cannot be retrofitted onto existing devices All current methods of securing the IoT rely on extensive hardware modifications to provide memory isolation and a secure execution environment. While research has been conducted towards cost-effective software-only solutions, no viable alternatives have been developed.

2.6 Summary

This chapter gave an overview of work presented in literature related to the focus of this dissertation: promoting dynamism, flexibility and security in the IoT. First, existing solutions were surveyed focusing on software development paradigms, dynamic reconfiguration and security for low-power IoT devices. Next, LooCI was presented as a case study of a reflective component model, combining the principle of component-based software development with powerful runtime reconfiguration and introspection. The chapter concluded with an in-depth analysis of the presented solutions. While reflective component models and hardware-based security fill an important gap in the evolution towards a dynamic and safe IoT, four areas of improvement were identified.

The following chapters present contributions specifically focusing on these challenges. Chapter 3 introduces *Safe reparametrization*, a methodology that lowers the complexity of achieving consistent reconfiguration by automatically resolving implicit configuration dependencies in distributed component-based applications. Chapter 4 and Chapter 5 aim to reduce the overhead and complexity of reflection in component models by introducing *Refraction* and *Tomography* respectively. Refraction augments existing application data with reflective data, thereby reducing the amount of explicit messages required for introspection, while reconfiguration is simplified through policies. Tomography introduces the concept of regions of components in a composition, and provides a way to inspect and control them efficiently. Lastly, Chapter 6 focuses on cost-effective security for the IoT, and proposes *MicroVisor*, the first software-only Trusted Computing Base for low-end IoT devices.

Chapter 3

Safe reparametrization for distributed IoT applications

This chapter introduces the first contribution of this work: *Safe reparametrization* of distributed IoT applications. A common challenge with large-scale IoT infrastructures running network-wide applications is maintaining correct configuration. Chapter 2 introduced reflective component models as an effective way to manage the complexity of developing IoT applications. Distributed applications can be configured and inspected at runtime through a per-component meta-model. While the advantages of component models are clear, misconfiguration of a single component in an application is difficult to track down and causes malfunctions and downtime.

Safe reparametrization extends reflective component-based middleware in two key ways: i) by offering a descriptive language for component developers to express implicit configuration dependencies between components, and ii) by providing a network protocol that efficiently resolves, monitors and enforces these dependencies over a distributed component composition, effectively minimizing misconfiguration, downtime and management overhead.

In this chapter, Section 3.1 describes the problem, and provides a general overview of the solution. Next, Section 3.2 delves deeper into the real-world problem that motivated this research. Section 3.3 provides a taxonomy of component roles, dependencies and constraints. Section 3.4 presents the descriptive language used along with the mechanisms behind the resolution and enforcement of dependencies. Section 3.5 discusses the proof of concept implementation on a representative IoT platform, alongside a real-world evaluation. Lastly, Section 3.6 concludes this chapter.

3.1 Introduction

Reconfigurable component models have proven to be a promising solution for managing the complexity of developing IoT applications. Examples of runtime reconfigurable component systems have been thoroughly discussed in Chapter 2, and include OpenCOM [18], RUNES [17], REMORA [96] and LooCI [47]. These systems provide the capabilities required to manage component life-cycle, configuration, introspection, and assembly at runtime. An essential feature of component-based middleware is component reuse, where one component can offer functionality to multiple application compositions. This way, both platform resources (i.e. flash, RAM) and code are shared between applications.

Contemporary component-based systems have some drawbacks when sharing component instances. Configuration conflicts can arise due to resource contention. While component binding dependencies are explicit in the form of interfaces and receptacles, implicit dependencies between component parameters emerge in a composition due to application level constraints. Consider a software composition that detects vehicle motion, using a composition formed from a magnetometer component and a motion detection component. The magnetometer component must sample at 2 Hz in order for the motion detection component to function properly. This configuration generates an implicit parameter dependency between the magnetometer and motion detection components that is not expressible with state-of-the-art component models.

Manually resolving these implicit dependencies is difficult and error-prone. In a multi-purpose IoT infrastructure where multiple actors reuse and reconfigure components, no single actor has an accurate understanding of all existing compositions and parameter dependencies. Existing compositions have to be introspected remotely, which incurs developer overhead and message passing overhead. Failure to resolve an implicit parameter dependency will cause disruption due to erroneous reconfigurations of existing applications.

Before investigating solutions, the problem is first motivated more through an in-depth case-study conducted on a real-world testbed.

3.2 Reparametrization in component compositions

Over the years multiple component-based IoT testbeds have been built in the DistriNet research group. Most notably a smart office and a smart lab environment, running multiple applications concurrently. Within this testbed, state-of-the-art component-based middleware is thoroughly tested running

various dynamic applications. Examples are: facility management, workforce management, security and workplace safety, each of which is managed by a different stakeholder. For these applications, data was logged continuously and analyzed for reconfiguration effort and latency. Analysis revealed the previously unknown problem of implicit dependencies between parametrized components in a composition. To further investigate, a series of informal experiments were conducted that were designed to better understand the impact that implicit dependencies have on reconfiguration effort and latency.

3.2.1 Experiment description

During the experiments, seven experienced component developers were tasked with a series of reconfiguration exercises to be conducted on the smart office. In each exercise the component assembler had to plan and enact an extension to one of the running applications. Throughout the experiment, *reconfiguration effort* (commands issued), *latency* (time required) and *disruption* (configuration errors made) were tracked.

No inter-stakeholder coordination of reconfiguration plans is assumed, and there is no up to date global view of the system. The assemblers all started out having limited information regarding the configuration and state of all the running component instances and the applications.

The reconfiguration typically happens in 4 steps:

1. **Interpretation:** Reading and understanding the required changes.
2. **Introspection:** Identifying and remotely inspecting the components of interest to check if and how they are reusable.
3. **Analysis:** Checking if the desired reconfiguration adversely affects any of the running applications.
4. **Reconfiguration:** Creating and executing a reconfiguration plan.

Analysis revealed that reconfiguration involving the deployment of components or the modification of bindings between them did not create any disruption and only required moderate reconfiguration effort and latencies. Modifying component parameters, on the other hand, caused high reconfiguration effort, latency and in most cases disruption of existing applications. The reason for this is implicit parameter dependencies that arise every time the consumer component has application requirements that constrain the possible values used to configure the functionality implemented by the producer component.

3.2.2 Classes of component compositions

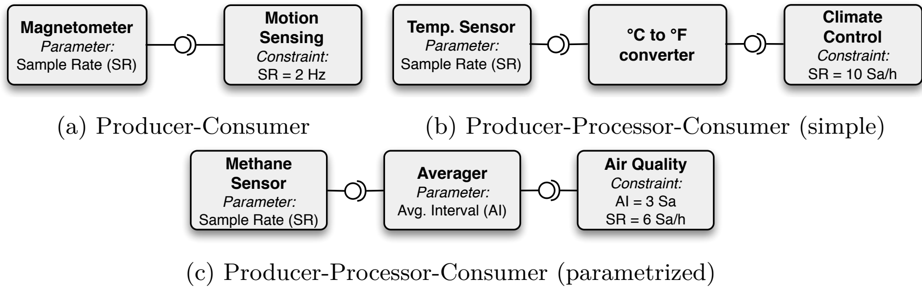
Three classes of component compositions were identified in the smart office environment where implicit dependencies are generated, see Figure 3.1.

- **Producer-Consumer:** Figure 3.1a depicts a Producer-Consumer composition. This is the implementation of the scenario previously explained in the introduction, where a Motion Sensing component requires a Magnetometer to sample at 2 Hz. In this case, the Motion Sensing component has an implicit dependency with the sample interval of the Magnetometer. Therefore the application requirements reified in the Motion Sensing component constrain the valid values for configuration property in the Magnetometer.
- **Producer-Processor-Consumer (simple):** Figure 3.1b exemplifies a Producer-Processor-Consumer composition where the implicit dependency is propagated from producer to consumer through a data processing component without any configurable parameters, thus only relaying implicit dependencies. A Temp. Sensor component sample rate is 10 Sa/h (samples per hour), the reading's units are converted and consumed by a Climate Control actuator component.
- **Producer-Processor-Consumer (parametrized):** In Figure 3.1c, a Producer-Processor-Consumer composition is shown with a parametrized processing component. The Methane Sensor component samples at a rate of 6 Sa/h, then an Averager component aggregates the data of 3 samples, which is consumed by an Air Quality component. Averager is a data processing component which has a configurable parameter that is constrained by the consumer, i.e. Air Quality. Both the Methane Sensor and Averager have implicit parameter dependencies with the consumer component because Air Quality has requirements on the temporal resolution of the readings. This constrains the valid parameter values for both Averager and Methane Sensor.

The software compositions classified above can be arbitrarily long, for instance having multiple Processors interconnected in a composition.

3.3 Prerequisites

This section discusses the requirements of safe reparametrization on component-based middleware. Next, the roles that a component can play during



(a) Producer-Consumer

(b) Producer-Processor-Consumer (simple)

(c) Producer-Processor-Consumer (parametrized)

Figure 3.1: Classes of compositions with implicit parameter dependencies.

composition-safe reparametrization are enumerated. Finally, a classification of constraint types is provided.

3.3.1 Component model requirements

Safe reparametrization requires a component model that at least has the following characteristics: i) explicitly defined interfaces and receptacles, ii) typed interface and receptacles through e.g. a **u**nique **id**entifier (UID), and iii) support for reparametrization of running components.

These requirements are met by all reflective components models previously discussed in Chapter 2: RUNES [17], REMORA [96], OpenCOM [18] and LooCI [47].

3.3.2 Component roles

Analysis revealed three component roles, each of which must be considered when resolving distributed parametrization dependencies. An overview of each of these roles is given, with reference to the example smart office compositions shown in Figure 3.1.

1. **Constrained components:** These are components which produce events differently based upon their parametrization. A concrete example of such a component is the Temp. Sensor component shown in Figure 3.1b. The Sampling Rate parameter (SR) influences how often temperature data is sensed and transmitted. which is constrained by the Climate Control component.

2. **Relaying components:** Relay components do not have constrained parameters, or constrain the parametrization of other components. They do however serve as a relay of parameter constraints along the chain of components in a composition. In Figure 3.1b, the °C to °F converter is an example of a relaying component. It relays data between a constrained component Temp. Sensor and the constraining component Climate Control.

3. **Constraining components:** Constraining components consume data that is produced and processed by other components in the composition. Constraining components require a specific parametrization of components producing and processing the data. The Climate Control component shown in Figure 3.1b is an example of a constraining component. It requires that the sampling rate of the Temp. Sensor component has a fixed value.

A component can play multiple roles at the same time. For example, the Averager component in Figure 3.1c relays a parameter dependency from the Methane Sensor (Sampling Rate) and also has a constrained parameter (Averaging Interval). Both of these parameters are constrained by the Air Quality component.

3.3.3 Constraints

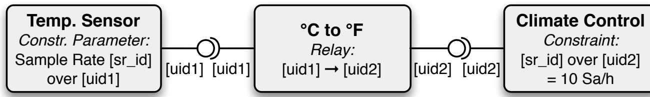
Constraining components impose two categories of parameter constraints: *Locking* and *Synchronizing*.

1. **Locking constraints** lock constrained parameters to a specific value or range. An example of this is the constraint imposed by the Climate Control component in Figure 3.2a. This constraint locks the Sampling Rate parameter of Temp. Sensor to 10 Sa/h. Another possibility is constraining the range of acceptable parametrization, e.g. 8 to 14 Sa/h.

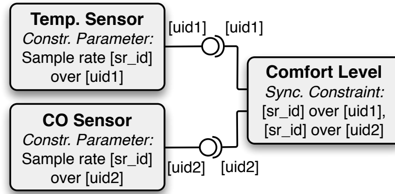
2. **Synchronizing constraints** require the synchronization of multiple parameters in the composition. This is illustrated in Figure 3.2b, where the sampling rate of the Temp. Sensor and CO Sensor components has to be time synchronized in order for the Comfort Level component to aggregate the data and function properly.

3.4 Design

Safe reparametrization is comprised of 3 elements: language annotations used by component developers, a constraint propagation protocol and a constraint enforcement protocol. This section takes a closer look at the role they fulfill.



(a) Locking constraint



(b) Synchronizing constraint

Figure 3.2: Component roles and constraint types.

3.4.1 Language annotations

In order to offer composition-safe reparametrization, component developers must specify parameter dependencies, relaying behaviour and constraints. To achieve this, a set of language annotations is introduced.

For **constrained components**, developers use the syntax shown in Listing 3.1 to identify constrained parameters. Constrained parameters are assigned an id, which is used by the constraint resolution protocol to reference the parameter. The component developer must specify both the id of the constrained parameter and the UID of associated the outgoing interface. Figure 3.2a shows an annotated version of the composition visualized in Figure 3.1b. In this scenario, the Temp. Sensor component specifies `ConstrainedParameter(sr_id, uid1)` to define the constrained parameter.

For **relaying components**, developers must specify the UID of the incoming receptacle and the outgoing interface over which the dependencies have to be relayed. Listing 3.1 shows the syntax that is used by relaying components. For example, the °C to °F component in Figure 3.2a specifies `DependencyRelay(uid1, uid2)`.

For **constraining components**, developers use the syntax shown in Listing 3.1. Constrained parameters are designated by their parameter id together with the UID of the incoming receptacle. The tuple formed by this pair of values uniquely specifies a constrained parameter. Lock constraints are specified by appending the parameter identifying tuple with the constraint itself, which

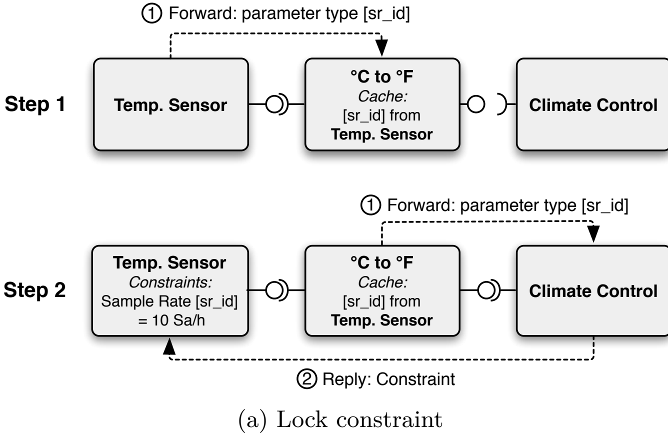
```
1  ConstrainedParameter(
2     parameter-id, //Reference to constrained parameter
3     interface-uid //Outgoing interface uid
4  );
5
6  DependencyRelay(
7     receptacle-uid, //Incoming receptacle uid
8     interface-uid //Outgoing interface uid
9  );
10
11 ParameterLock(
12    parameter-id, //Reference to constrained parameter
13    receptacle-uid, //Incoming receptacle uid
14    constraint //Open/closed interval or value
15 );
16
17 ParameterSync(
18    {
19       parameter-id, //Reference to constrained parameter
20       receptacle-uid //Incoming receptacle uid
21    },
22    {paramter-id, receptacle-uid},
23    ...
24 );
```

Listing 3.1: Language annotations defining implicit parameter dependencies.

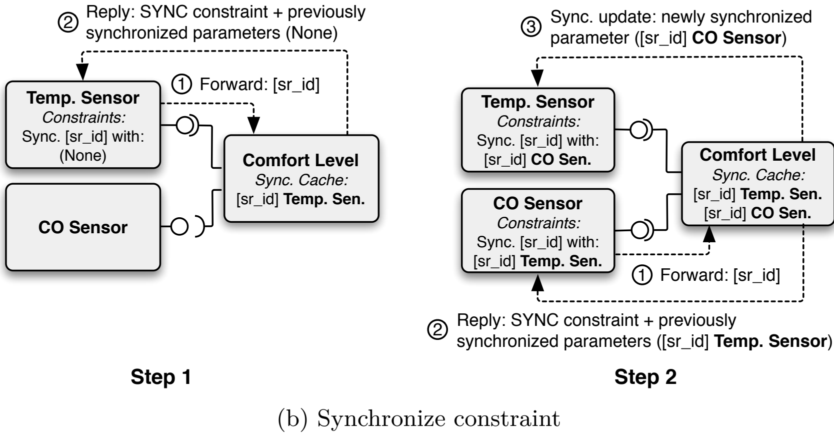
is either a range or a single value. Synchronizing constraints are specified by providing a number of parameter-identifying tuples, the values of which must remain the same. The example lock constraint of the Climate Control component shown in Figure 3.2a is expressed as follows `ParameterLock(sr_id, uid2, 10)`. The example synchronizing constraint of the Comfort Level component shown in Figure 3.2b is expressed as `ParameterSync({sr_id, uid1}, {sr_id, uid2})`.

3.4.2 Constraint propagation

Constraint propagation happens through a network protocol that efficiently relays appropriate constraints from constraining components to constrained components. Bindings can be local or can cross node boundaries, requiring the transmission of a radio message. The constraint propagation protocol has two phases. In *phase one*, descriptions of constrained parameters are propagated along the chain of components as bindings are made. A caching mechanism



(a) Lock constraint



(b) Synchronize constraint

Figure 3.3: Step-by-step constraint resolution of compositions from Fig. 3.2.

is used at every component along the chain from the constrained component to the constraining component. Every time a new binding is made between two components, the local cache of the component with the providing interface is checked for constrained parameters that should be forwarded. These cache entries are then forwarded along the chain as far as existing bindings allow. Using local caching by intermediate components as opposed to a stateless method serves two purposes: i) network traffic is reduced, as constraints do not have to travel all the way through a composition whenever a single binding is modified, and ii) it allows the bindings of an application to be made in any sequence, with messaging overhead remaining the same regardless of the order. In *phase two*, any downstream constraining component sends its constraints

back directly to the constrained components. Looped component compositions should generally not occur, but duplicate constrained parameter propagations due to loops can easily be detected and filtered out.

The process of locking constraint propagation for the composition shown in Figure 3.2a is shown in Figure 3.3a. When the first binding is made, Temp. Sensor forwards its constrained parameter to the °C to °F component, which caches it. Next, when the last binding is made, °C to °F forwards this constrained parameter to Climate Control, which matches it with a lock constraint and sends this constraint back directly. This mechanism would similarly work if the components were bound together in the opposite order: when °C to °F and Climate Control are bound together, the cache is still empty and nothing is forwarded. Next, when Temp. Sensor is bound to °C to °F, its constrained parameter is forwarded as far as possible, first passing by °C to °F where it is cached, and next to Climate Control, which replies immediately with a constraint.

A synchronizing constraint must be propagated to all synchronized components and thus the constraining components must store references to each synchronized parameter and its associated component. Figure 3.3b illustrates how synchronizing constraints are propagated when building the composition of Figure 3.2b. When the first binding is made, Temp. Sensor forwards its constrained parameter to Comfort Level, where it is matched with a synchronizing constraint. As there are currently no other synchronized parameters, none are sent back with the reply containing the synchronizing constraint. When CO Sensor is bound, the same happens except this time a reference to `sr_id` on Temp. Sensor is sent back with the synchronizing constraint to CO Sensor. Lastly, an update with a reference to `sr_id` on CO Sensor is sent to Temp. Sensor.

3.4.3 Constraint enforcement

The constraint enforcement protocol ensures parameter constraints are maintained during reparametrization. This is accomplished by checking the constraints specified in the constraining component every time a parameter on a constrained component is set.

This requires 3 steps: i) check all lock constraints on the parameter, and return a *Constraint not met* error message if one is broken, ii) if no lock constraints are broken, tentatively store the new value for the parameter (NPV), iii) enforce all synchronizing constraints by setting all synchronized parameters on remote components to NPV. This recursively triggers the same 3 step constraint check on the remote component. If any remote component returns a *Constraint not met* error message, the local change is rolled back and the same error is returned.

If all parametrizations succeed, set the local parameter to NPV and return a *Success* message to the parametrizing entity.

3.5 Implementation and evaluation

A prototype was implemented on the LooCI [47] middleware, previously discussed in-depth in Chapter 2. LooCI is a middleware for building distributed component-based IoT applications. It complies with all of the requirements listed in section 3.3. LooCI supports a number of platforms. Safe reparametrization was implemented on the Contiki [30] based AVR Raven [6] port of LooCI. The Raven mote offers a 16 MHz Atmel microcontroller, 16 KB of RAM and 128 KB of flash memory. All of the functionality necessary was implemented using LooCI components, requiring no modifications to the middleware.

The performance of safe reparametrization was evaluated against the original version of LooCI in terms of both middleware overhead and commands issued. Performance was assessed in 3 specific scenarios inspired by the smart office deployment introduced previously. The scenarios build on top of each other, and Figure 3.4 shows the final complete distributed application composition. In every scenario, functionality is added by deploying new components and binding them to existing ones.

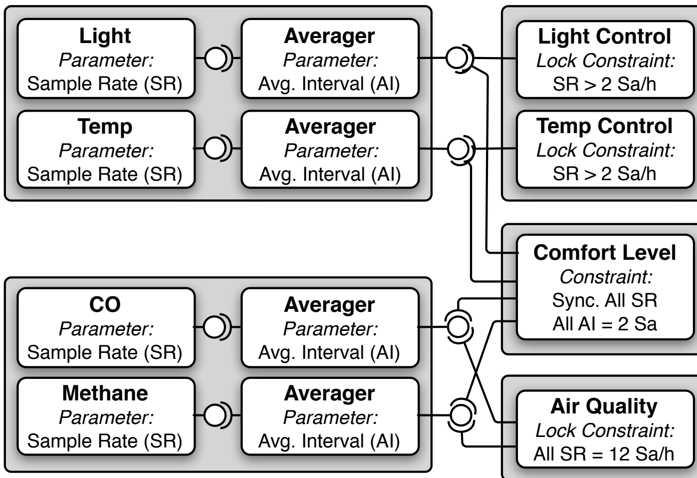


Figure 3.4: Evaluation scenario: parameter dependencies in a smart office deployment.

In **scenario 1**, an automatic light/climate control system is deployed over 2 nodes. A first node collects the light and temperature readings and averages them. The controller components are deployed on a separate node and constrain the sensor sampling rates. **Scenario 2** expands upon the first scenario by introducing a node which samples and averages CO and Methane readings. All 4 sensor readings are aggregated in a Comfort Level component deployed on a separate node, which evaluates the level of comfort in an office. Comfort Level imposes a synchronizing constraint on all sensors, together with a lock constraint on the averaging window of all averager components. **Scenario 3** adds the Air Quality component, which uses the existing CO and Methane sensors. This component imposes additional constraints on the sampling rate of the sensors.

3.5.1 Component size overhead

Component sizes in reconfigurable component models are significant because they largely determine energy expenditure during deployment due to radio usage, and as a result impact node lifetime [44]. Safe reparametrization requires components to be annotated with meta-data. The storage of this meta-data incurs overhead on the size of the deployable components. On average, **scenario 1** incurs a component size overhead of 18.3 bytes, **scenario 2** incurs an extra 18.4 bytes and **scenario 3** incurs an extra 19 bytes. These increases in size are insignificant when compared to an average unannotated component size of 790 bytes (worst case overhead of 2.4%).

3.5.2 Static middleware overhead

Without any components deployed, LooCI still needs space in both flash and RAM memory in order to provide functionality. The additional components required to implement safe reparametrization increase this static overhead. Both flash and RAM usage are evaluated. Considering the 128 KB of flash and 16 KB of RAM available, the modified version uses respectively 5.3% (65130 bytes vs. 58162 bytes) and 2% (9351 bytes vs. 9030 bytes) more of the total flash and RAM than the original LooCI middleware. In conclusion, the overhead imposed by the modifications is minimal.

3.5.3 Dynamic middleware overhead

The execution of components results in the dynamic allocation of RAM on top of the base consumption discussed in the previous subsection. Figure 3.5

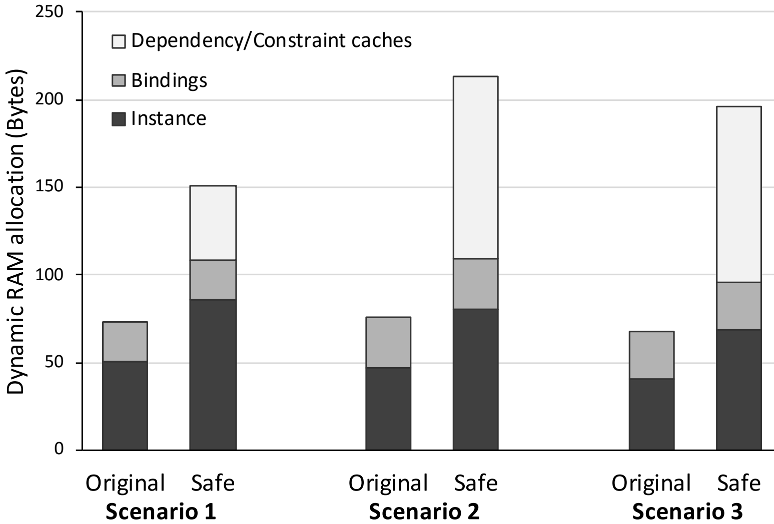


Figure 3.5: Breakdown of average dynamic RAM overhead for each scenario.

shows a detailed breakdown of the average allocated memory per node. The allocations can be split in three categories: memory used by component instance bookkeeping, memory used for storing component bindings and dependency and constraint caches. Note that **scenario 2** and **scenario 3** have a higher average overhead then the first scenario. This is due to the introduction of the synchronizing constraint in the second scenario, which requires more cache space on each node. Taking into account the 16 KB RAM available, in the worst case (**scenario 2**) this means on average 0.83% more RAM used on each node (213 bytes vs. 76.3 bytes).

3.5.4 Network overhead

Network overhead is an important performance indicator for any distributed software running on resource constrained IoT devices. Sending and receiving messages over the network causes high energy consumption, and increased network traffic directly impacts battery lifetime. The amount of messages sent over the network when reparametrizing and binding is measured, and the overhead with and without safe reparametrization is compared. When binding, component dependencies are propagated, generating some message overhead. In case of **scenario 2** and **scenario 3**, only the messages sent when expanding

the previous scenario are counted. For **scenario 1**, an increase from 6 to 12 messages is measured, **scenario 2** increases from 10 to 62 messages and in **scenario 3** message overhead increases from 4 to 10.

It is clear that the modified version has more overhead due to the constraint propagation protocol, which runs at bind time. This is most apparent in **scenario 2**, where a non-trivial amount of network traffic is generated keeping the synchronization caches consistent over all nodes. It is important to note that binding is less frequent than reparametrization, where the constraint enforcement protocol gives significant savings.

Table 3.1 gives an overview of the amount of messages sent when reparametrizing either one of the sampling rate parameters of the sensors, or the interval of one of the averagers. In the original version, the running component composition has to be remotely inspected to ensure that all constraints are met. This generates significant network usage. A worst case example is the reparametrization of the sampling rate in scenario 3, where 33 extra messages have to be sent for a full reparametrization. As a result, safe reparametrization is a significant improvement.

Parameter	**Scenario 1**		**Scenario 2**		**Scenario 3**	
	Without	With	Without	With	Without	With
Sample rate	8 pkts	1 pkts	31 pkts	4 pkts	32 pkts	4 pkts
Avg. interval	4 pkts	1 pkts	6 pkts	1 pkts	6 pkts	1 pkts
Total gain		**10 pkts**		**32 pkts**		**33 pkts**

Table 3.1: Number of messages sent over the network during reparametrization.

3.5.5 Management overhead

The advantages of safe reparametrization are also most apparent when comparing the amount of commands an actor has to issue and the associated network traffic generated during reparametrization. Table 3.2 shows this data for each scenario when reparametrizing the sampling rates and the averager intervals. Because the original version has no automatic constraint enforcement, the actor has to introspect the composition manually to ensure that all constraints are met and that the reparametrization is composition-safe. In all cases this generates significantly more commands and messages (on average 24 times more).

Parameter	Scenario 1		Scenario 2		Scenario 3	
	Without	With	Without	With	Without	With
Sample rate	8 cmds	1 cmds	31 cmds	1 cmds	32 cmds	1 cmds
Avg. interval	4 cmds	1 cmds	6 cmds	1 cmds	6 cmds	1 cmds
Total gain		**10 cmds**		**35 cmds**		**36 cmds**

Table 3.2: Number of commands issued by the user during reparametrization.

3.6 Summary

This chapter presented the first technical contribution of this dissertation: *Safe reparametrization*. Implicit configuration dependencies within distributed IoT applications can cause inconsistent reconfiguration, resulting in down time and faulty behaviour. This chapter first investigated this problem for component-based systems through an experiment with experienced component developers. Analysis showed that while bindings between component are explicitly typed and cause no issues, implicit dependencies between the parameters of components introduce complexity.

Safe reparametrization solves this problem by leveraging annotated components to externalize implicit dependencies, which thereupon are enforced at runtime by a specialized network protocol. By automatically resolving and enforcing distributed parameter dependencies over a component composition, safe reparametrization avoids configuration conflicts and greatly improves the manageability of reflective component models. A prototype implementation for a representative resource constrained hardware platform was evaluated and showed that the overhead is acceptable, with minimal messaging over the network and acceptable memory costs. Furthermore, when compared to exhaustive introspection to ensure safe and correct reconfiguration, this approach greatly lowers messaging and management costs.

While the methodology presented in this chapter shows promising results, it is important to take a step back and reflect on how it can be enhanced to be more versatile. A possible improvement is to provide systematic support for more constraints besides the two proposed in this chapter. To accommodate for this, the solution can be extended with facilities that allow developers to define new types of constraints as well as how they should be enforced. When moving towards more complex constraints or even higher-order constraints (i.e. constraints on a set of constraints), the scalability of the in-network resolution and enforcement of constraints can be improved through a layered resolution of dependencies combined with a partial centralization of configuration data, a

concept which is further explored in the next chapter.

This concludes the first contribution of this dissertation. The next two chapters embody the second contribution of this thesis and aim to reduce the general overhead and complexity of reflection by introducing two novel approaches: *Refraction* and *Tomography*.

Chapter 4

Refraction: lowering the cost of reflection in the IoT

This chapter introduces the concept of *Refraction*, a principled means to lower the cost of managing reflective meta-data for the Internet of Things. The benefits of reflective component-based middleware for building open and reconfigurable applications, as argued in Chapter 2, are clear, but the cost of using remote reflective operations remains high. Refractive components address this problem by selectively augmenting application data flows with their reflective meta-data, which travels at low cost to *refractive pools*, serving as loci of inspection and control for distributed applications running in the network. Reconfiguration is further simplified by *reactive policies*, providing a mechanism to trigger reconfigurations based on incoming reflective meta-data.

The remainder of this chapter is structured as follows. Section 4.1 reviews the problems present in state-of-the-art component models and provides a high-level overview of how refraction solves them. Section 4.2 introduces a real-world running example that showcases standard reflective operations and their shortcomings. Section 4.3 outlines the principles of refraction. Section 4.4 applies these principles to realize a refractive component framework. Section 4.5 evaluates this framework in a real-world case-study scenario. Finally, Section 4.6 concludes.

4.1 Introduction

Building applications for IoT systems is notoriously difficult. In addition to the usual complexities of creating distributed applications, IoT systems are resource constrained, require a high degree of flexibility and dynamism, and often deployed at large scale and in inaccessible locations, such as flood plains [46] and volcanoes [109]. This requires that all configuration and management must be performed remotely.

Reflective component models [47, 18, 96] have a strong track record of realising adaptable and evolvable applications for IoT systems. As discussed in detail in Chapter 2, the advantages of such component models are numerous: the complexity of embedded software development is mitigated by the reuse of generic software components, customisable middleware [46] conserves system resources through the removal of redundant software functionality, and finally, reflection enables runtime reconfiguration and adaptation [39] after system deployment. A per-component meta-model structures what configuration can be read (i.e. *introspection*) and modified (i.e. *reconfiguration*), and bounds reflective operations. While reflective component models help in building open and reconfigurable distributed systems, reconfiguration can be complex. When reconfiguring distributed applications it is frequently necessary to work in a coordinated fashion with the meta-model of multiple distributed components. This leads to increased development complexity and message passing overhead.

This chapter proposes *refraction*, a principled means to lower the cost of reflection for IoT applications. Refraction minimises the need to inspect and reconfigure individual components by incorporating an efficient meta-data distribution mechanism in the component model kernel. Refractive components use this mechanism to selectively augment the application data that they process with elements of their own meta-model. When refractive components are bound together they naturally form *refractive streams* along which component meta-data can flow. These streams terminate at *refractive pools*, a network location where all components that contributed to the stream can be inspected and reconfigured. Reconfiguration is facilitated by the introduction of *reactive policies* which can be deployed on any node in the reactive stream or pool. These policies offer a mechanism to trigger reconfiguration based on incoming meta-data. The benefits of refraction are evaluated in a real-world IoT case-study: *configuration monitoring and automated repair*. The evaluation shows that refraction significantly reduces message transmissions while not increasing development overhead.

4.2 Reflection and its shortcomings in IoT systems

Before delving deeper into the mechanisms of refraction, the problem is first motivated with a running example that is used throughout this chapter. The attentive reader may recognize this example from Chapter 2, where it was used to introduce LooCI [47], a state-of-the-art reflective component model. This chapter builds its ideas and proof-of-concept on top of LooCI, although they are applicable to any contemporary reflective component model.

Figure 4.1 shows an example LooCI software composition, that is used in a real-world IoT deployment to detect motion in a room. In this example, two Motion Detector components —residing on N_1 and N_2—transmit sensor data to a Motion Aggregator component on N_3. Here the motion readings are processed, aggregated and forwarded to a Motion Reporter component residing on a resource rich back-end node N_4. Motion Reporter exposes the data to a web platform for storage and viewing. Gray boxes represent computational platforms, where components are deployed and executed. Software components are shown as white boxes with solid black lines, which publish values via their *provided interfaces* (—o), and receive values from via their *required interfaces* (—( ). Components also have key-value pairs of properties that can be used to parametrize their behaviour.

The application is controlled by an external Manager entity, which issues reflective operations to the component middleware running on each node in order to introspect the software system, perform *structural* reconfiguration by connecting or disconnecting required and provided interfaces and *behavioural* reconfiguration by modifying component properties. The interaction between the Manager and the reflective middleware is depicted with thicker blue arrows in Figure 4.1. Examples of the reflective operations that can be used by the Manager are shown in Listing 4.1.

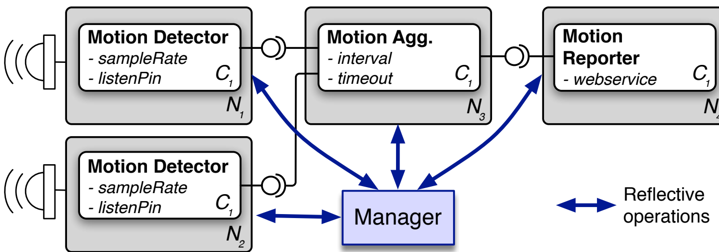


Figure 4.1: Running example illustrating reflection.

```
1 // introspection operations
2 N1.getProperties(C1)
3 N1.getProperty(C1,sampleRate)
4 N3.getWiresFrom(C1)
5 // reconfiguration operations
6 N3.setProperty(C1,timeout=30s)
7 N4.activateComponent(C1)
8 N3.wireTo(C1,N4,motionEvent)
9 N4.wireFrom(N3,C1,C1,motionEvent)
```

Listing 4.1: Reflective operations on the running example.

Considering the example shown in Figure 4.1, two shortcomings of reflection quickly become evident. Firstly, reflection requires the transmission of many messages to query and reconfigure remote components. This is very problematic, as research [44] has shown that radio transmissions are the primary source of energy consumption for IoT devices. Secondly, writing reflective code for distributed management, as shown in Listing 4.1 is complex. These problems could be mitigated by transmitting all meta-data to the managers, however, the memory and network resources consumed by storing reflective meta-data must also be minimised. This gives rise to three requirements that should be tackled by *refraction*:

- **Requirement 1:** The number of messages that are required to perform reflection should be minimised.
- **Requirement 2:** The subset of meta-information that is distributed should be customisable.
- **Requirement 3:** Mechanisms are required to specify in-network reconfiguration behaviour, that operates close to the point of action.

4.3 Refraction in principle

As described in Section 4.2, reflective software processes are empowered to reflect upon and modify their implementation. Inspired by the power of this metaphor, this chapter introduces the complementary metaphor of refraction. In the same way that characteristics of a material can be inferred from the light that traverses it, the characteristics of a refractive software component can be inferred from application data that it processes. Refracted meta-data

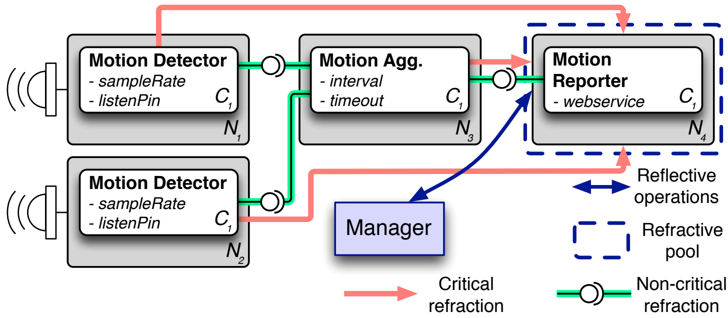


Figure 4.2: Replacing reflective with refractive operations on the running example.

travels together with application data across the distributed component graph, allowing the components that receive it to perform reflection at reduced cost.

Figure 4.2 shows how refraction distributes component meta-data across the network. Each node is extended with a *refraction engine*, which keeps relevant elements of the meta-model synchronised between nodes. A selected node, in this case N_4, collects all relevant meta-data and acts as a single-point of introspection for the 3 upstream nodes. This node is referred to as a *refractive pool*, surrounded by a blue dashed line in the figure. Two types of *refractive streams* are offered: *non-critical* streams, shown in green and *critical* streams, shown in red. Non-critical refractive streams transport meta-data in an opportunistic way by augmenting application data, while critical refractive streams transport important meta-data directly to the refractive pool. Critical streams are offered as an alternative when the rate of application data is too low or unpredictable and a minimum latency should be guaranteed. This extended version of LooCI with refraction support is called *RxCom*.

RxCom provides mechanisms to selectively aggregate reflective meta-data updates that describe component properties and bindings and transmit them using either critical or non-critical refractive streams. *Refractive policies* provide a mechanism to specify what meta-data should be refracted using which type of refractive stream and where it should be stored. *Reactive policies* provide a mechanism to trigger reconfigurations based on incoming refracted meta-data. This new architecture minimises message passing through the use of aggregation, while not increasing development overhead. The following subsections address three key questions for RxCom:

What does the basic meta-model describe? The basic meta-model of a software component allows for the introspection and reconfiguration of software functionality, structure and behaviour. RxCom extends the basic LooCI component model, which is described in Section 4.3.1.

How should the meta-model be refracted? To minimise memory footprint, only a subset of the meta-model should be refracted. Yet it is not possible to know a-priori which meta-data should be distributed to support introspection and reconfiguration. Therefore, customisable *refractive policies* are introduced, which determine which subset of the meta-data is distributed, and how it is distributed. This is described in Section 4.3.2.

How to react to incoming refracted data? Reconfiguration behaviour is frequently triggered by changing application context. To support this behaviour, nodes are equipped with *reactive policies* specifying how to store, forward and react to refracted meta-data. This is described in Section 4.3.3.

4.3.1 The refractive meta-model of RxCom

RxCom's basic meta-model is based on the LooCI meta-model, which is shown more formally in Figure 4.3. The meta-model defines the concepts of node, component and binding, and determines what can be inspected and modified using reflection. RxCom then extends this meta-model with refractive concepts.

$$
\begin{aligned}
\textsf{Node} &::= \mathit{address} \times \textsf{Comp}^* \times \underset{\sim}{\textsf{Binding}}^* \\
\textsf{Comp} &::= \mathit{c\text{-}id} \times \mathit{c\text{-}type} \times \textsf{Property}^* \times \\
&\quad\;\; \textsf{Interface}^* \times \underset{\sim}{\textsf{Status}} \\
\textsf{Property} &::= \mathit{p\text{-}name} \times \underset{\sim}{\mathit{p\text{-}val}} \times \mathit{p\text{-}type} \\
\textsf{Interface} &::= \underline{\texttt{required}} \times \mathit{e\text{-}type} \mid \underline{\texttt{provided}} \times \mathit{e\text{-}type} \\
\textsf{Status} &::= \underline{\texttt{active}} \mid \underline{\texttt{inactive}} \\
\textsf{Binding} &::= \textsf{Local} \mid \textsf{RemoteIn} \mid \textsf{RemoteOut} \\
\textsf{Local} &::= \mathit{e\text{-}type} \times \mathit{c\text{-}id} \times \mathit{c\text{-}id} \\
\textsf{RemoteOut} &::= \mathit{e\text{-}type} \times \mathit{c\text{-}id} \times \mathit{address} \\
\textsf{RemoteIn} &::= \mathit{e\text{-}type} \times \mathit{address} \times \mathit{c\text{-}id} \times \mathit{c\text{-}id}
\end{aligned}
$$

Figure 4.3: Formal specification of the basic RxCom meta-model.

```
1 // get components of type "Motion Detector" from N1
2 n1Compnts = N1.getComponents("Motion Detector")
3 // update components to sample every 120s
4 forall (n ∈ n1Compnts) {
5   N1.setProperty(n,sampleRate,120s)
6 }
```

Listing 4.2: Querying and modifying the basic meta-model with reflective operations.

While the LooCI meta-model is used as a foundation, the refractive extensions themselves are generic and can be added to any component model.

The following notation is used in the meta-model specifications: keywords written in a Sans font are terms defined by the grammar; *italic* keywords denote terms defined outside of the grammar, such as strings and numbers; underlined keywords denote constant symbols; the star operator $\cdot^* \cdot$ denotes a set of a given attribute and the times operator $\cdot \times \cdot$ creates tuples of elements. All elements of the meta-model can be introspected. Wave-underlined keywords denote parts of the meta-model that can also be reconfigured using reflection.

As shown in Figure 4.3, a node consists of an address, a set of *components*, and a set of *bindings*. Each component has a local instance identifier *c-id*, an identifying type *c-type*, a set of properties, a set of required and provided interfaces, and a status indicating whether it is active or not. Each property associates a name *p-name* to a value *p-val* of a given type *p-type*. A binding connects one or more provided interfaces to required interfaces, and has a type *e-type* corresponding to the type of the events sent and received by both interfaces. This binding is said to be *local* when it connects two components running on the same node, and *remote* if it connects a components hosted on different nodes. Outgoing bindings specify the local component and the address of the remote node, while incoming bindings specify the originating node address, source component and the destination component.

The meta-model of a node can be queried by reflective operations, represented using a simplified Java-like notation. The **bold** keywords in the code listings denote operations over the meta-model. For example, given a node N, the statement N.**getProperties**(C) retrieves the set of properties of the component C deployed on N, and N.**setProperty**(C, property=value) modifies the value of a property of a component C deployed on node N. Listing 4.2 shows more advanced reflective operations, where a node is queried for a specific type of components, after which the sample rate of those components is adjusted.

4.3.2 Specifying refractive policies

A refractive policy determines what subset of the reflective meta-model is refracted (i.e. transmitted with application data) and when. Refractive policies are specified by RefPolicy in Figure 4.4, which extends the basic meta-model of RxCom (Figure 4.3). A refractive policy consists of:

- SComp – specifies which component instance's meta-model to refract
- SelectElem – identifies a collection of elements from the component's meta-model, such as the state, properties or bindings.
- Frequency – specifies when the selected meta-data elements should be refracted. The options are: `always`, which appends the selected refracted data to every outgoing component message, `on-change`, which appends the selected refracted data only when it differs to the previously sent refracted data, and `never`, which causes the meta-data of the selected element not to be refracted.
- Target – specifies which mechanism should be used for transmitting the meta-data. The options are: `non-critical`, which augments outgoing events with meta-data, or *address*, which dispatches meta-data directly to the refractive pool located at the node with the given address.

Refractive policies are set on a per node basis and do not change the underlying component meta-model, rather they specify a systematic and customisable way

$$
\begin{aligned}
\mathsf{Node} &::= \mathit{address} \times \mathsf{Comp}^* \times \mathsf{Binding}^* \times \mathsf{RefPolicy}^* \\
\mathsf{RefPolicy} &::= \mathsf{SComp} \times \mathsf{SelectElem}^* \times \mathsf{Frequency} \times \mathsf{Target} \\
\mathsf{SComp} &::= \underline{\texttt{allComps}} \mid \textit{c-id} \mid \textit{c-type} \\
\mathsf{SelectElem} &::= \mathsf{SProps} \mid \mathsf{SBindings} \mid \mathsf{SStatus} \\
\mathsf{SProps} &::= \underline{\texttt{allProps}} \mid \textit{p-name} \mid \textit{p-type} \\
\mathsf{SStatus} &::= \underline{\texttt{status}} \\
\mathsf{SBindings} &::= \underline{\texttt{allBindings}} \mid \underline{\texttt{localBindings}} \\
&\mid \underline{\texttt{inBindings}} \mid \underline{\texttt{outBindings}} \\
\mathsf{Frequency} &::= \underline{\texttt{on-change}} \mid \underline{\texttt{always}} \mid \underline{\texttt{never}} \\
\mathsf{Target} &::= \underline{\texttt{non-critical}} \mid \mathit{address}
\end{aligned}
$$

Figure 4.4: Extension of RxCom's meta-model with *refractive* policies.

```
1 // refract the properties of every component on N1
2 N1.refract(allComps,allProps,on-change,non-critical)
3 // continuously refract status of C1 on N3
4 N3.refract(C1,status,always,non-critical)
5 // critical refraction of C1 on N2 to node N4
6 N2.refract(C1,status,on-change,N4)
```

Listing 4.3: Updating a refractive policy on the running example.

to selectively distribute that model. The API for configuring refractive policies is exemplified in Listing 4.3, which shows how it is applied to the running example of Figure 4.2.

It should be noted that multiple policies can refer to the same element of the component meta-model with different frequency values. For example, there can be a general policy for `allComps` and a more specific one for a given component ID. In this case, the most specific policy takes precedence.

4.3.3 Specifying reactive policies

A node in RxCom can react in four different ways upon receiving refracted meta-data, it can: i) *discard* it; ii) *forward* it through a specific interface; iii) *store* it to the local meta-data registry; or iv) *trigger reconfigurations* based on the received meta-data update. The extension to the RxCom meta-model to accommodate these changes is presented in Figure 4.5, and exemplified in Listing 4.4.

Reactive policies can either be *component marks* or *conditional reconfigurations*.

$$
\begin{aligned}
\mathsf{Node} &::= \mathit{address} \times \mathsf{Comp}^* \times \mathsf{Binding}^* \times \\
&\qquad \mathsf{RefPolicy}^* \times \mathsf{RctPolicy}^* \\
\mathsf{RctPolicy} &::= \mathsf{SComp} \times \mathsf{Mark} \\
&\quad\ \mid \mathsf{Reconf} \\
\mathsf{Mark} &::= \underline{\texttt{forward}} \mid \underline{\texttt{store}} \mid \underline{\texttt{discard}} \\
\mathsf{Reconf} &::= \mathit{guard} \times \mathit{reconfiguration}
\end{aligned}
$$

Figure 4.5: Extension of RxCom's meta-model with *reactive* policies.

The former mark local components as being `forward` or `store`, and the latter associate a triggering condition to reconfiguration instructions. Upon receiving a remote message over the interface of a component C, RxCom searches for refracted data aggregated to it. If refracted data is found, its mark is checked. If C is marked as `discard`, it the data is thrown away, otherwise, it is processed as follows:

1. If C is marked as `forward` then the refracted data is queued to be aggregated to the next outgoing message from each provided interface of C and of all local components that are connected to C.
2. If C is marked as `store` then the refracted data is added to a *refraction table*, which is indexed by the component IDs and network addresses that provided the refracted data. If the data overrides a previous entry, then the old value is temporarily saved until the next step completes. All nodes which store data become *refractive pools*.
3. Each reconfiguration R with a guard G that evaluates to true is executed. G is a boolean expression over the local meta-model and the refraction table, both before and after the store operations. R describes modifications to the meta-model of local or remote components using the standard reflection API.

The example in Listing 4.4 showcases how reactive policies apply to the composition of the running example depicted in Figure 4.2. First, the Motion

```
1  // configure intermediate node N3 to unconditionally forward all
2  // upstream meta-data
3  N3.forward(allComps)
4  // configure N4 to act as a refractive pool for upstream components
5  // of type "Motion Detector"
6  N4.store("Motion Detector")
7  // add reactive policies to N4, actively monitoring and reconfiguring
8  // the "sampleRate" property of the "Motion Detector" components
9  N4.addReconfiguration("sync.pol")
10
11 // contents of the "sync.pol" file
12 if ( N1.C1(sampleRate) != N1.C1(sampleRate)' ) {
13   N2.C1(sampleRate) = N1.C1(sampleRate);
14 }
```

Listing 4.4: Specification of reactive policies on the running example.

```
1  // get components of type "Motion Detector" from all refracted nodes
2  // refracted into node N4
3  rxCompnts = N4.getRefComponents("Motion Detector")
4
5  // update component properties
6  forall (c ∈ rxCompnts) {
7     c.getNode().setProperty(c,sampleRate,120s)
8  }
9
10 // add policy to all nodes known by N4
11 forall (n ∈ N4.getRefNodes()) {
12   n.refract(allComps,allProps,on-change)
13 }
```

Listing 4.5: Example queries on the refractive pool of the running example.

Aggregator component on N_3 is set to forward any refractive meta-data downstream. Next, the Motion Reporter component on N_4 is set to store all upstream refractive data coming from Motion Detector components. A reconfiguration is then added to N_4, which sets the rate parameter of the Motion Detector component on N_1 to the Motion Detector component on N_2 whenever it is updated. The reconfiguration and its associated guard are written in a domain specific language, which includes traditional arithmetic and logical operators. This language supports reading and the modification of meta-data, and uses the prime (') as a suffix to represent the value prior to the store operations.

A refractive pool can be queried by a remote manager for the meta-model of both the node itself as well as for the refracted meta-data from all upstream nodes. Hence the API of nodes that are refractive pools is extended to satisfy queries over a group of nodes, rather than over a single node. Examples of such queries for the running example are listed in Listing 4.5.

4.4 Refraction in practice

The principles of refraction in RxCom are realised by extending LooCI [47] with support for refractive policies and reactive policies. Chapter 2 gives an in-depth description of the basic LooCI architecture upon which RxCom is built. The rest of this section describes the extensions made to LooCI, visualized in Figure 4.6 by red boxes.

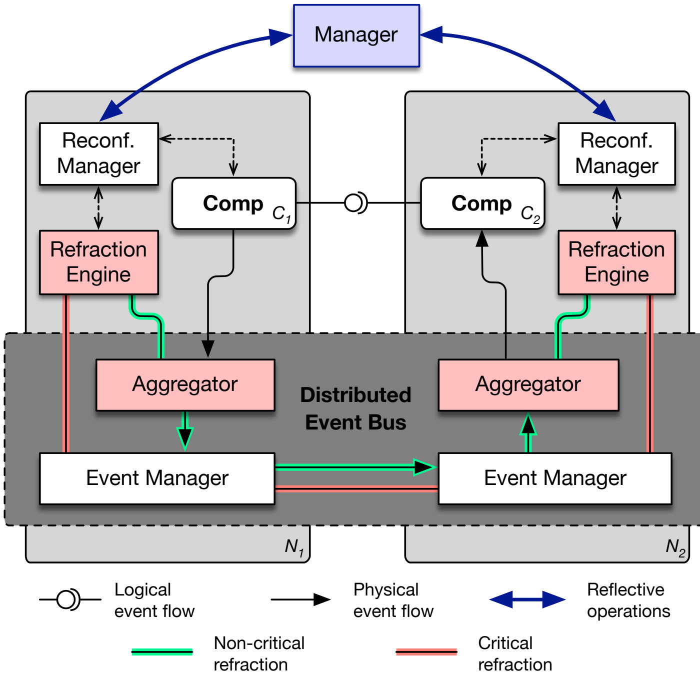


Figure 4.6: The LooCI architecture extended with RxCom modules (red boxes).

Components A component definition in RxCom preserves the same information as a component in the original LooCI meta-model. However, the refraction engine stores a mapping between refractive or reactive policies and their associated components. The former maps components to a list of tuples $(\{e\}, f, t)$ with the elements $\{e\}$ being refracted, f the frequency of refraction and t the target. The latter maps components to the flag `forward` or `store`. References to refracted meta-data and to the reconfigurations that are triggered by refracted data are maintained in separate tables. Reactive policies are evaluated and executed whenever matching meta-data is received via refraction.

Architecture The architecture illustrated in Figure 4.6 includes two refractive extensions: the Refraction Engine and Aggregator modules. These additional modules are isolated from existing functionality and use hooks available in LooCI to modify behaviour.

Refraction Engine The Refraction Engine module maintains the refractive policy table and the reactive reconfiguration policy table as described above, which specify how data should be refracted and used in reconfiguration respectively. The Refraction Engine module receives new policies and policy modifications from manager nodes; receives updates to the refractive meta-model from the Reconfiguration module; executes local reconfiguration instructions when triggered by refracted data and exchanges meta-data either through non-critical or critical refraction. Non-critical refractive meta-data is relayed to the Aggregator module, while critical refractive meta-data is sent directly as a single event over the distributed event bus.

Aggregator The Aggregator operates on the level of the distributed event bus, intercepting incoming and outgoing application events, and is responsible for aggregating and deaggregating the refracted data to and from the main application traffic. It uses a queue of outgoing refracted data for each output interfaces, to cache the data to be aggregated in the next outgoing message.

Manager The standard LooCI network manager is extended with refraction-related calls. This extended API allows, for example, the querying of the refraction tables (stored refracted data in the node), to update or add refraction and reactive policies, as illustrated in Listings 4.4 and 4.5.

In addition to adding refraction-related calls, the manager was extended to parse and recognize the reactive policies shown in Listing 4.4. These policies are first parsed locally by the manager, and then serialized for transmission over the network. The target node will deserialize the policy and enforce it when relevant meta-data is received via a refractive stream.

4.5 Evaluation

The evaluation starts with an analysis of the performance overhead associated with evaluating policies and aggregating meta-data. Next, a configuration repair scenario is presented that showcases the benefits of using refraction to inspect and reconfigure IoT applications. This scenario is based on a real-world IoT application. In the original approach, application configuration is periodically introspected and, when faults are discovered, repaired using remote reflective operations. The case study compares the original reflection based approach to a refraction based approach. These results are presented in Section 4.5.2.

All the results presented in this section are benchmarked on an extended version of the Java/OSGi port of LooCI. All tests were conducted on a standard computer, with an Intel Core i5-2400 CPU and 8 GB of RAM. The OSGi version of LooCI is targeted at more powerful back-end devices where refractive pools will reside. For the performance and overhead analysis, this is appropriate as nodes with refractive pools will be the most policy-heavy and have to deal with massive deaggregation of meta-data. The rest of the evaluation is agnostic to which platform is used for prototyping.

4.5.1 Performance and overhead analysis

Refraction introduces some performance overhead when sending and receiving application data due to i) the aggregation of refractive meta-data with *outgoing* application traffic and ii) the deaggregation of refractive meta-data from *incoming* application traffic and the evaluation of reactive reconfiguration policies pertaining to the new meta-data. Table 4.1 shows minimum, maximum and average performance timings for all case study policies described in the following two sections. The mechanisms that underlie refraction perform well in terms of both the time required to aggregate reflective meta-data and to deaggregate it and evaluate policies.

Operation	Min.	Max.	Avg.
Data aggregation	0.29 ms	0.51 ms	0.36 ms
Deaggregation & Policy eval.	0.35 ms	0.48 ms	0.42 ms

Table 4.1: Performance overhead of aggregation and policy evaluation.

4.5.2 Configuration repair case-study

This case-study looks at a common problem with IoT systems [63]: repairing faulty system configurations that arise due to faults, damage or power-loss. For example, configuration elements can be lost due to memory corruption after a node reboots. A classical solution to this is a monitoring service running on a reliable back-end. The monitor periodically queries the nodes for their configuration and checks if it matches with a *desired state*. If not, reconfiguration is carried out.

Refraction offers a more efficient alternative. First refractive policies are used to specify which configuration parameters from the meta-model should be refracted. Next, a refractive pool is created on the reliable back-end. Finally, reactive

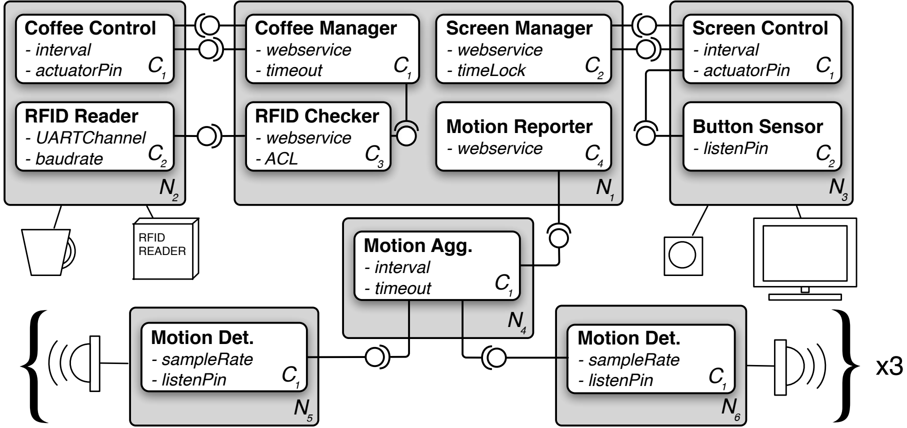


Figure 4.7: Evaluation scenario: configuration repair in a real-world smart lab.

policies are installed on the back-end to check the current configuration against the desired configuration, and where necessary carry out repair operations.

Scenario Both approaches are benchmarked by implementing a common scenario, depicted in Figure 4.7, which is a subset of a real world smart lab deployment in our research facility. More specifically, 3 services are offered by this composition: i) a *Motion detection* service using data from 6 embedded nodes equipped with motion sensors spread around the lab, ii) a *Screen control* service, which allows users to either remotely or locally turn on or off screens used for presentations, and iii) a *Coffee control* service, which authorizes access to a coffee machine through RFID authorization.

N_1 is a reliable always-on back-end, and runs components with more complex functionality. Furthermore, this node is connected to a database for logging purposes, and all back-end components have configurable webservice interfaces to allow integration with a web platform. All other nodes in the network are embedded nodes with volatile configuration parameters.

While it is possible with both approaches to monitor a subset of the configuration parameters, this scenario considers a complete monitoring approach. The configuration of 15 components spread over 10 nodes is monitored, amounting to 71 configuration parameters that have to be consistent for this deployment to correctly operate.

Development effort A first point of comparison is the amount of development effort spent implementing a purely reflective solution versus using one that leverages refraction.

In the case of a reflective monitoring solution, introspection commands are scripted in the back-end to query all the parameters of the component composition spanning the network. When a specific parameter deviates from the desired value, a reconfiguration command is sent to rectify the problem. An example of reflective code used for monitoring is shown in Listing 4.6. In this example, the parameters of one of the Motion Detector components are monitored.

Listing 4.7 on the other hand shows the reactive policies that can be used when using refraction to monitor the same configuration parameters. It can be seen that the reactive policies closely match reflective operations from the point of view of the developer. The primary difference is that the execution of reflective code is statically scheduled, while reflective policies pertaining to a parameter are automatically evaluated when an updated parameter value is available.

The general development effort is quantified in the form of LoC (Lines of Code) for both approaches. When implementing configuration repair for Figure 4.7, **142 lines** of reactive policies are required versus **146 lines** of reflective code.

```
1  while(True) {
2     // Component has to be active
3     if(N5.getStatus(C1) == deactivated)
4        N5.activateComponent(C1)
5     // Guarantee a sampleRate >= 60
6     if(N5.getProperty(C1, sampleRate) < 60)
7        N5.setProperty(C1, sampleRate, 60)
8     // Motion detector is connected to pin 2 on C1
9     if(N5.getProperty(C1, listenPin) != 2)
10       N5.setProperty(C1, listenPin, 2)
11    // Wire motion events to aggregator on N4
12    if(N5.hasWireTo(C1, N4, motionEvent))
13       N5.wireTo(C1, N4, motionEvent)
14    ...
15    // Monitoring rate: hourly
16    sleep(1h);
17 }
```

Listing 4.6: Reflective code for monitoring the Motion Detector's parameters on N_5.

```
1   if(N5.C1.Status == deactivated)
2      N5.Status = activated
3
4   if(N5.C1(sampleRate) < 60)
5      N5.C1(sampleRate) = 60
6
7   if(N5.C1(listenPin) != 2)
8      N5.C1(listenPin) = 2
9
10  if(! hasWireTo(N5.C1,N4,motionEvent)
11     wireTo(N5.C1,N4,motionEvent)
12
13  ...
```

Listing 4.7: Refractive code for monitoring the Motion Detector's parameters on N_5.

In conclusion, reflection imposes no development overhead when implementing a configuration repair system. On the contrary, the absence of scheduling code is expected to lead to development savings in more complex reconfiguration scenarios.

Average latency A second point of comparison is the average latency of configuration repair. This metric indicates how long it takes before a fault is repaired. Figure 4.8 breaks this down for each component in the compositions and every monitoring method.

As can be seen from Figure 4.8, repair latency based on non-critical refraction varies for each component. This is caused by delays imposed by application data. RFID Reader and Button Sensor are very ill-suited for non-critical refraction, because their application data is sent unpredictably. Only when somebody pushes the button or swipes an RFID card is application data generated. In the data shown in Figure 4.8, it is assumed that on average every 30 minutes an event is sent. However, in reality application events are unpredictable and there are no hard guarantees in terms of average repair latency.

Both critical refraction and classic monitoring at two different rates give the same average repair latency for every single parameter in the network. When comparing all approaches, both critical and non-critical refraction clearly give significant latency advantages when compared to classic monitoring with reflective operations. When dealing with stochastic application data, critical refraction performs best.

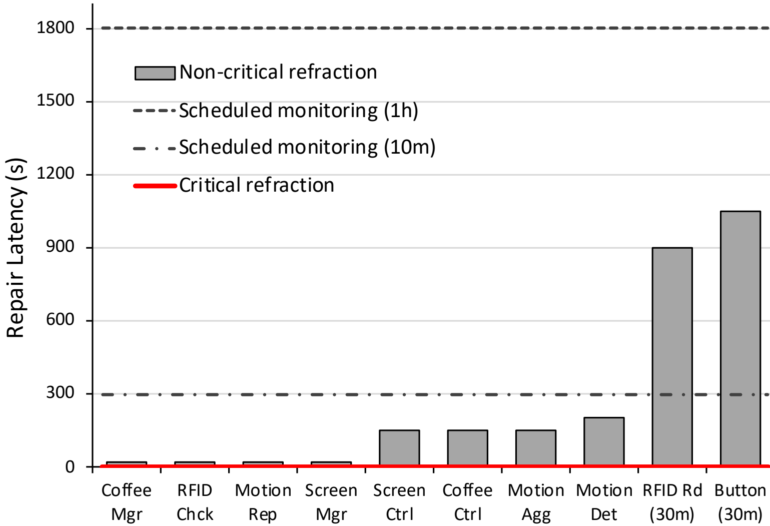


Figure 4.8: Average latency of configuration repair for the evaluation scenario.

Network overhead Lastly, the amount of explicit network messages per hour required for repairing one *remote* configuration fault in the same time span is compared. Table 4.2 shows the results. A classic monitoring approach almost always causes more explicit network traffic than either critical or non-critical refraction when using reasonable checking intervals. Comparing nominal scheduled monitoring checking configuration once per hour with non-critical refraction shows a *reduction of 85.7%* in network traffic. This number is still quite conservative, in reality scheduled monitoring can incur a higher overhead due to configuration of a single node exceeding packet size. Furthermore, explicit messaging overhead of monitoring will increase linearly with the scale of the network, while critical and non-critical refraction will scale gracefully.

Method	**Overhead**	**Relative reduction**		
		Mon. 10m	Mon. 1h	Crit. refr.
Scheduled monitoring (10m)	37 msgs/h	-	-	-
Scheduled monitoring (1h)	7 msgs/h	81.1%	-	-
Critical refraction	2 msgs/h	94.6%	72.2%	-
Non-critical refraction	1 msg/h	97.3%	85.7%	50%

Table 4.2: Network-wide messages per hour for 1 repair in the evaluation scenario.

4.6 Summary

This chapter presented *Refraction*, a principled means to lower the cost and complexity of reflection in IoT deployments, and introduced RxCom, a component model that realises this concept. Refraction provides a concise and low-overhead mechanism to manage reflective meta-data across distributed IoT applications. Refraction provides the developer with: i) *refractive policies* to control meta-data distribution, ii) *reactive policies* to automatically react to updates of meta-data from remote neighbours and iii) runtime support for efficient meta-data distribution through existing data flows and reactive reconfiguration enactment.

Evaluation of RxCom shows that policy evaluation in refractive pools has low overhead. Refraction was further evaluated through a configuration repair case-study, where scheduled configuration monitoring was compared with a refractive solution developed using RxCom. In comparison to monitoring purely through reflection, the refractive solution developed using RxCom and drastically reduces network traffic in all cases, and shows faster repair latencies for IoT applications with steady data flows. For applications with intermittent traffic, a variant called *critical* refraction that does not rely on existing network traffic shows near-instant repair latencies with minimal extra messaging.

Refraction is part of the second contribution of this dissertation: reducing the cost and complexity associated with reflection. Refraction minimizes the overhead of introspection by efficiently distributing it to central locations. While network messaging related to reconfiguration is not directly reduced, refractive policies significantly lower complexity and management effort. Looking ahead, the next chapter introduces a methodology that lowers the cost of both introspection and reconfiguration without relying on existing application traffic, thereby making it more useful in scenarios with low network activity.

Chapter 5

Tomography: efficient regional reflection for the IoT

This chapter describes the concept of *Tomography.* As discussed in Chapter 2, componentization of distributed IoT deployments offers numerous advantages such as code-reuse and runtime reconfiguration, however, the introspection and reconfiguration of distributed applications is cumbersome and inefficient. The previous chapter introduced refraction, which lowers the cost of inspection by augmenting application data with meta-data. Tomography further improves on this by recognizing that components are often queried in groups, and reduces overhead by reimagining the visitor design pattern from object-oriented programming for distributed component based compositions. Tomography is applied on a real-world application and the performance of this approach is evaluated when discovering, introspecting and reconfiguring. In comparison to classic management operations, tomography is shown to reduce both the number of explicit queries and the volume of network messages. This significantly reduces management effort and energy consumption.

Within this chapter, Section 5.1 introduces the problem, and outlines how tomography tackles it. Section 5.2 introduces a running example which illustrates the shortcomings of reflection when querying groups of components. Section 5.3 introduces the principles of tomography. Section 5.4 evaluates this framework in a real-world case-study scenario. Section 5.5 concludes this chapter.

5.1 Introduction

IoT applications are known to be hard to build and maintain. Typical IoT applications are run on a large scale infrastructure of extremely resource constrained embedded systems, demand high software flexibility and dynamism, and are often deployed in hard to reach locations. Because of this, IoT systems demand comprehensive support for remote management and reconfiguration. As discussed in detail in Chapter 2, a promising solution for this is reflective component-based middleware [47, 18, 96]. Components are small units of functionality with clearly defined interfaces and parameters, which can be independently deployed and managed remotely. Applications are built by binding components together in a composition, minimizing developer effort and maximizing reuse. These mechanisms allow for software evolution [44] and adaptation [39] after deployment through reconfiguration.

Reflective component models introduce a per component meta-model, which is causally connected to the implementation of the component. The meta-model exposes elements of the component —such as interfaces and parameters that influence component behaviour— that can either be *introspected* (querying of meta-model) or *reconfigured* (modifying the meta-model). The introspection and reconfiguration of the meta-model is an important tool to monitor and enact change in a component composition. However, the distributed nature of the meta-model implies a large message passing overhead and high management complexity when introspecting or reconfiguring a component composition, which can span many nodes. In the preceding chapter, refraction was introduced as a means to reduce the costs of introspection. While this approach showed good results, two shortcomings can be identified: i) refraction only works optimally when application data is flowing, low data rates can cause stale meta-data, and ii) reactive policies simplify reconfiguration but do not reduce runtime costs. This chapter further improves on this and aims to have an efficient solution even in those cases.

This chapter proposes the concept of tomography[1], a new approach to efficiently introspect and reconfigure component compositions. Tomography is inspired by the visitor design pattern from object-oriented programming. The visitor pattern is a structured way of performing an operation on an object hierarchy by having a visiting object traverse the hierarchy [35]. Tomography applies this concept to distributed component compositions, exploiting existing communication flows for efficiency. Tomography is based on the notion of *probes* that can be injected into a component composition, flowing along the same path (i.e. a set of bindings) as the application data. While traversing the component composition,

[1]This contribution is aptly named after the tomography imaging technique, which builds up a full image from sliced sections.

the probe can either query or modify the meta-model of individual components as it visits them. The added notion of a *region* of components in a composition allows for more fine-grained targeting of queries.

The benefits of tomography are evaluated in the same real-world IoT case-study used for *refraction*: a smart lab composed of 12 sensor nodes. The evaluation shows that tomography greatly reduces the number of commands that developers must issue and also the number of messages that are transmitted when compared to classic reflection.

5.2 Reflection on regions of compositions

Reflective component models have been discussed in depth in Chapter 2, and are intensively used throughout this dissertation. Tomography extends reflective component models further by allowing cost-effective reflection on a region of a component composition. This section introduces a running example, used as a guideline when explaining the intricacies of tomography. This example and the concepts introduced in this chapter are built on top of LooCI [47], a reflective middleware for the IoT, although this approach can be applied to other reflective component-based systems as well.

Figure 5.1 shows a component composition used in a real-world IoT smart lab deployment. This example application authorises users to operate a coffee

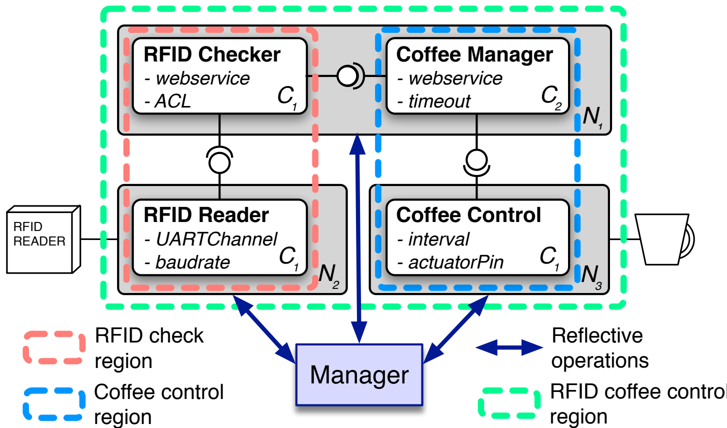


Figure 5.1: Running example illustrating reflection on regions of component compositions.

machine through an RFID tag. In the distributed component composition shown, node N_2 is equipped with RFID hardware, and is running an RFID Reader software component. This component transmits swiped IDs to a remote RFID Checker component residing on resource rich back-end node N_1, which authorises or denies access based on an access control list (ACL). Additionally, access attempts are logged and viewable on a web platform. The coffee machine is physically connected to node N_3, where the Coffee Control component controls its state. The Coffee Control component is connected to the back-end Coffee Manager component, which exposes control over it on a web page. Lastly, RFID Checker and Coffee Manager are bound in the back-end, so an authorised RFID swipe enables the coffee machine. Deployed components have a node-local identifier, denoted in Figure 5.1 with C_i.

Each component has a local meta-model that is causally connected to the underlying implementation. Reflection allows the per-component meta-model of components to be remotely *introspected* and *reconfigured* by an external Manager entity. These interactions are visualised in Figure 5.1 by the thicker blue arrows. The same reflective operation often has to be performed over a group of connected components in an application. In the IoT application shown in Figure 5.1, for example, three *regions* are distinguished that group distributed components that are often queried together. Listing 5.1 highlights reflective operations made to groups of components in the smart lab deployment: collecting the properties of all components of a region, and deactivating the components of another region.

This example exposes the two main shortcomings of reflection on regions of component compositions. Firstly, remote reflection requires that many messages are sent over the network in order to introspect or reconfigure sets of components. In the context of IoT applications, these networks are typically extremely low power and every packet sent imposes a large energy overhead [44]. Secondly, the scope of a typical reflective operation is limited to a single component's

```
1 // introspection: properties of RFID coffee control region
2 N2.getProperties(C1)
3 N1.getProperties(C1)
4 N1.getProperties(C2)
5 N3.getProperties(C1)
6 // reconfiguration: deactivate RFID check region
7 N2.deactivateComponent(C1)
8 N1.deactivateComponent(C1)
```

Listing 5.1: Reflective operations on the regions of the running example.

meta-model. Due to this, the manager has too many responsibilities: keeping track of every component in each region, and querying every component of a region individually for every region-wide query. This becomes impractical with large and dynamic regions, and does not scale well.

5.3 Tomography

Tomography is used to inspect or modify a *region* of a component-based application, i.e. a set of connected components. This is achieved by a generalisation of the visitor pattern for object-oriented programming. A *probe* is broadcasted to a set of *start components*, which search for the desired region by traversing the components using the dataflow order. It is also possible to traverse the components in the opposite direction, called *upstream tomography*, but the remainder of this chapter will use the direction of the dataflow.

This section begins by defining regions of IoT applications and explaining how they are specified by a manager. Next, the queries that can be performed to inspect and modify connectors are described, along with how they are propagated throughout the network.

5.3.1 Regions of component compositions

Formally, a *region* is a set of connected components. It is specified by: i) a set of *start* components, and ii) a (possibly empty) set of *end* components. Start components mark the beginning of the region, which includes all components traversed by following the dataflow direction until either an end component or a component without outgoing interfaces.

The example in Figure 5.1 contains 3 different regions, represented by dashed rectangles. The *RFID Check* region, for example, consists of the two components on the left side, and it is specified by indicating that RFID Reader is a start point and RFID Checker is an end point of this region. For both of the other two regions in the figure it is enough to specify only the start point.

5.3.2 Specifying regions

The external Manager is responsible for specifying regions. It starts by identifying the boundaries of a region (start and end components) and assigning a unique ID to that region. It then sends a reconfiguration request to the nodes with the

selected boundary components to mark them as start and/or end components of the region with the assigned ID. Finally, the Manager stores the region ID and the nodes where the start components are deployed. The region ID and start nodes are sufficient for the Manager to inspect and reconfigure the full region.

Marking boundaries of regions instead of marking components belonging to regions has two main advantages: i) *compactness*, as less data is needed to specify a region; and ii) *flexibility*, as it becomes easier to adapt regions to newly added components or removed components without large changes to the marking data.

There are no restrictions regarding the amount of regions that can be active within a network, the amount of components they contain or whether or not they overlap. Which components should ideally be grouped together in a region depends on the maintainer of the distributed application, any group of components that is frequently queried together will benefit from being contained in a region. Ideal candidates are groups of components forming a single distributed application, or a coherent subgroup thereof providing supporting functionality that should be configured together.

5.3.3 Using tomography

The specification of regions allows the precise definition of scope for inspection or modification of components. Listing 5.2 shows how to use tomography to perform the same operations on the running example as previously shown in Listing 5.1. The reduction in management complexity is clear: tomography requires a single interaction per region, while classic reflection needs multiple interactions per region.

Under the hood, tomography only sends commands to the nodes with start components, in this case node N_2. This process is illustrated in more detail in Figure 5.2, which shows querying the *RFID coffee control* region in the running example. The manager wraps the desired query in a *probe* that is sent to the

```
1 // introspection: get properties of RFID coffee control region
2 rfidCoffeeControl.getComponentProperties()
3 // reconfiguration: deactivate RFID check region
4 rfidCheck.deactivateComponents()
```

Listing 5.2: Using tomography on the regions of the running example.

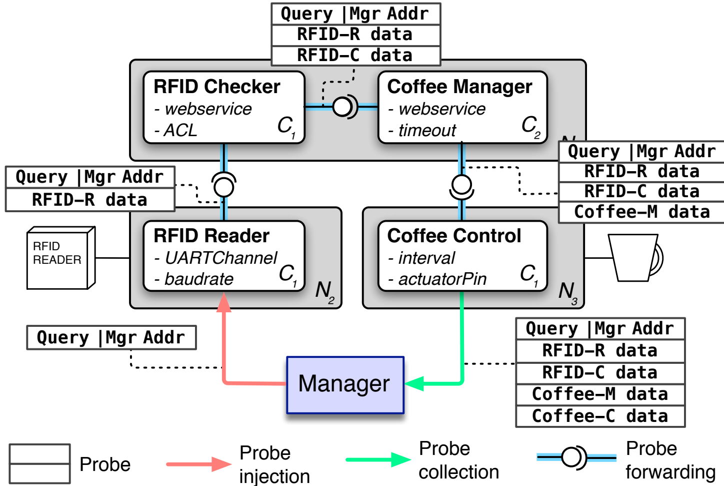


Figure 5.2: Querying the RFID coffee control region of the running example using tomography.

start component RFID Reader. This probe initially contains only the query and the return address of the manager, and is forwarded by each component in the region until it reaches a dead-end, which is either a component explicitly marked as end component, or a component without provided interfaces. At this point the probe is sent back to the manager using the return address. While traversing the composition, each component executes the query on its local meta-model, and the results are appended to the probe. A more general case with multiple start points and splits during traversal exists and is explained in detail in the following subsections.

5.3.4 Probe propagation

This subsection provides more technical details on how the probes are propagated in advanced component compositions. The probes have to be *split* when traversing a component with multiple provided interfaces and when multiple start points exist, and the probes must stop the traversal when reaching a component already traversed by a probe with the same query (*merge* of probes).

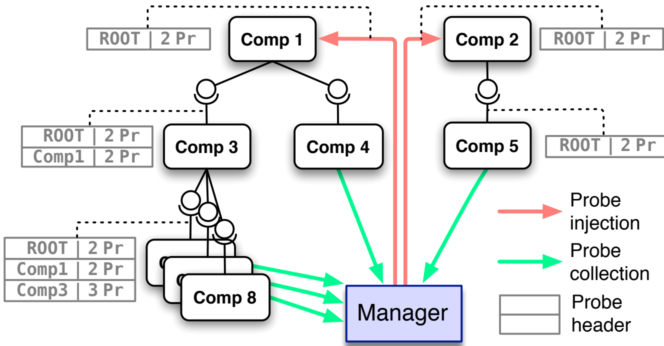


Figure 5.3: Splitting probes - extending headers of probes to notify the manager when to stop waiting for replies.

Splitting probes When a probe is split into n probes, its header is extended with a new pair containing: i) the identifier of the component that created the split (or `Root` if it was the manager), and ii) the number n of splits. A concrete example is visualized in Figure 5.3, where probes get injected into 2 root components, one of which gets split twice again as it traverses the composition. Upon collection of the probes, the extra header describing where and how many times the probe was split allows the manager to conclude if all probes have been collected or if there is any split probe missing.

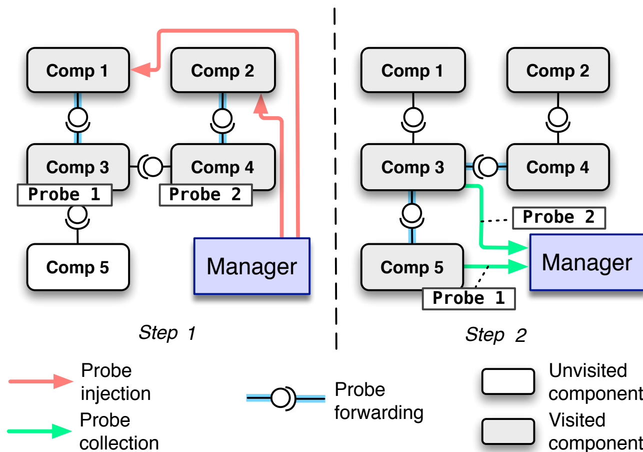


Figure 5.4: Merging probes - marking components as visited.

Merging probes Each probe also includes an unique transaction ID. During the traversal of a composition, each component is marked as being visited by that ID. Hence, if a visited component receives a probe with the same ID, it will simply send it back to the manager without executing its query. Figure 5.4 illustrates this process. In Step 1 the manager sends split probes to the 2 entry points Comp 1 and Comp 2, who execute its query, are marked as visited, and forward it to Comp 3 and Comp 4. In Step 2 the probes are propagated further, and component Comp 3 receives the same probe again from Comp 4, consequently returning the probe to the manager without any action.

5.4 Evaluation

Tomography is evaluated by comparing classic reflective operations with tomography when inspecting and reconfiguring a region of components. The cost of querying a region is measured by counting the number of messages sent over the low-power network (Section 5.4.1), the cost of setting up a region is measured by the number of messages sent to create a region (Section 5.4.2), and the cost of managing a region is based on how easy it is for the manager to perform queries and to maintain the required knowledge about regions (Section 5.4.3). These measurements are applied to a real-world smart lab deployment in Section 5.4.4.

5.4.1 Querying a region

In general, the number of messages required to query a region defined by its start and end points is smaller than when the manager contains a list of all components. For example, the query illustrated in Figure 5.2 uses 4 messages between nodes: a message from the manager to node N_2, 2 messages between the 3 nodes, and a message from N_3 to the manager. A purely reflective approach where the manager queries all 3 nodes independently would use 8 messages: 4 to perform the query and 4 to collect the result.

This can be defined more precisely by analysing 3 regions with different topologies in Figure 5.5: a chain of n components, a split of n components, and a merge of n components. Without tomography, querying all components in all these scenarios requires around $n * 2$ messages ($(n + 1) * 2$ for the split and merge cases). Using tomography, the chain scenario reduces this number to $n + 1$ messages, the split scenario uses a similar number of messages ($(n * 2) + 1$), and the merge scenario increases this number to $n * 3$ messages. Summarising, the number of messages is reduced by half with chaining, is not affected with splitting, and is increased by n with merging.

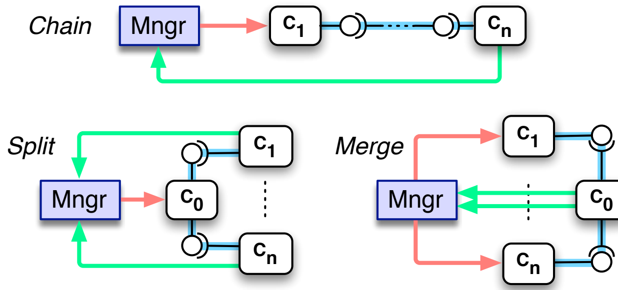


Figure 5.5: The effect of component topology on tomography message overhead.

Chaining is more commonly found in IoT applications than splitting or merging. Indeed, all the examples presented so far exchange a smaller number of messages with tomography than without it. A larger deployment, described later in Section 5.4.4, has a region that benefits from tomography only when using upstream tomography, i.e., when traversing components in the opposite direction of dataflow. Furthermore, simple optimisations can avoid the increased number of messages in the merge case. For example, one could allow nodes to wait for probes that could be merged, sending the combined probe to the rest of the chain instead of replying to the manager. This would require extra annotations to nodes and probes, increasing the complexity of modifying bindings within regions without breaking the traversal process of regions.

5.4.2 Setting up a region

Setting up a region involves updating the nodes with information that describes the region. For tomography this means marking start and end components of a region by sending request over the low-power network to their deployment nodes to mark them as such. This is a fixed cost that has to be paid before a region can be queried. Without tomography, no components have to be marked and the setup consists of storing a list of all the components of the region and the nodes where they reside locally on the manager. In this case, there is no overhead on the network.

With tomography, the number of messages required to setup a region highly depend on the region itself. More specifically, it depends on its number of boundary nodes. In the *best case* it is enough to mark a single component as a start component to define a whole region; this is the case for 2 out of the 3 regions in the example in Figure 5.1, and for the chain and split scenarios in Figure 5.5. In the *worse case*, all components have to be tagged as start and/or

end components, producing as many messages as there are components in the region.

5.4.3 Management overhead

The biggest advantage of tomography is that, after setting up the regions, complexity from the point of view of the Manager is greatly reduced. This claim is supported by 2 performance indicators: i) the complexity to query regions, and ii) the amount of data stored by the manager.

Querying regions As illustrated in Listing 5.1 (example without tomography) and Listing 5.2 (example with tomography), explicitly querying a region with tomography requires less instructions than querying each component individually. The exception for this scenario is when the manager only needs to query part of a region—e.g., all RFID readers in the coffee application—, in which cases it is more performant to query the desired components individually.

Recalling regions In order to query a region the manager must know how to reach it. Using tomography, the manager needs to store information about every node with a start component for each region. This means N regions times n_s node addresses with start components (in average) per region. Without tomography, the manager needs to store information about every node and component in each region. This means N regions times n_c pairs of components and node addresses (in average) per region. The size of this management information is always strictly smaller with tomography, since it does not store component information (only nodes), and only nodes with start components.

5.4.4 Smart lab case-study

This case-study looks at how tomography performs in comparison to classic reflection in a real world case-study, using the 3 previously discussed metrics as performance indicators. Both approaches are benchmarked by introspecting and reconfiguring the component composition shown in Figure 5.6, which is a subset of a *smart lab* deployment in our research facility. This component composition was used before in Chapter 4 to evaluate refraction, and offers 3 services: i) a *Motion detection* service using data from 6 embedded nodes equipped with motion sensors around the lab, ii) a *Screen control* service, which allows either remote or local control of screens used for presentations, and iii) a *Coffee control* service, which was used as a running example throughout

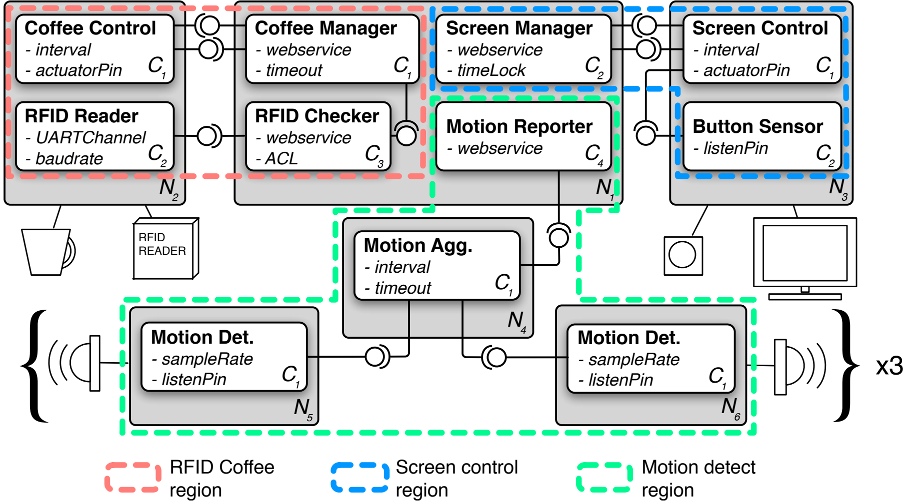


Figure 5.6: Evaluation scenario: smart lab component composition with regions.

this chapter and authorises access to a coffee machine through RFID. The component composition has 3 regions grouping the components each of the 3 applications are composed of, denoted by dashed lines in Figure 5.6.

Message overhead The number of messages used to introspect and reconfigure regions in the 3 defined regions and the whole composition are presented in Table 5.1. In most cases tomography requires less messages to be sent over the low-power network when compared to classic reflection. The exception is the *Motion detect* region, which is less efficient because of the 6-way merge, following the discussion in Section 5.4.1. One way to avoid this problem is to traverse the region upstream, which is listed as *Motion Detect* ***up*** in the table, turning the merge into a split. This effectively makes the tomography approach more efficient again. The automatic detection of the most efficient traversal direction is a possible optimisation.

Setup A next point of comparison is the region set-up cost outlined in Section 5.4.2. For tomography, only the start points need to be tagged at setup time. Assuming upstream tomography for *Motion detect*, setting up all regions in this scenario would cost **3 messages**. In case *Motion detect* is traversed downstream, the set-up cost is **8 messages** due to the multiple start points of that region.

Region	Tomography		Reflection		Gain
	Intr.	Recf.	Intr.	Recf.	
RFID Coffee	4	3	8	4	42%
Motion Detect	19	13	16	8	-33%
Motion Detect **up**	14	8	16	8	9%
Screen Control	3	2	6	3	44%
All components	20	12	30	15	29%

Table 5.1: Message overhead when querying the regions of the smart lab.

Management overhead Lastly, the management overhead is evaluated. As discussed in Section 5.4.3, tomography greatly simplifies the specification of queries over groups of components. This is also the case in this case-study. Where **15 queries** are required when using standard per-component reflection to query all components, tomography only requires **1 query**.

Summarising, tomography imposes a minimal set-up cost and outperforms classic reflective operations both in terms of message passing overhead and management overhead.

5.5 Summary

This chapter introduced *Tomography*, an approach to lower the overhead of reflection for component-based IoT applications. Tomography provides a way to specify regions of connected components and to introspect and reconfigure these using a distributed variant of the visitor design pattern. Tomography is based on the notion of *probes*, which when injected into a region of components flow along the bindings connecting them, collecting meta-data and enacting reconfiguration in individual components as they are visited.

Tomography was evaluated using a real-world smart lab scenario, and has shown promising results. When querying representative component compositions and regions, tomography outperforms classic reflection both in terms of network messaging and management overhead, while imposing only a minimal set-up cost prior to use. As a result, tomography significantly reduces management effort and energy consumption of reflection.

Together with refraction, tomography embodies the second contribution of this dissertation, which aims to reduce the overhead and complexity of introspection and reconfiguration for component-based middleware. The contributions discussed up until this point all focus on enabling dynamism through effective remote management. As the software on IoT devices grows more open and dynamic, the need for securing them increases. The next chapter looks at a cost-effective way to secure the IoT.

Chapter 6

Securing dynamic IoT systems

This chapter presents the final contribution of this dissertation: the *Security MicroVisor*, a software-based security architecture for constrained IoT devices. SµV provides memory isolation through selective virtualization of machine instructions and assembly-level code verification. This memory isolation is used to safeguard the integrity of critical operations and secret key material while concurrently running insecure user applications, providing a Trusted Computing Base. SµV is used to implement two key security features: *remote attestation* and *secure deployment*.

Within this chapter, Section 6.1 introduces the problem SµV tries to solve and provides a general overview of the solution. Section 6.2 describes the design rationale and key mechanisms of SµV. An overview of remote attestation and secure deployment protocols is provided in Section 6.3 and Section 6.4 respectively. Implementation details are discussed in Section 6.5. Section 6.6 reports on the evaluation of SµV. Section 6.7 looks at applications beyond what is discussed in this chapter, and finally, Section 6.8 concludes.

6.1 Introduction

Security is a vital element to the success of the Internet of Things. IoT is becoming more predominant in everyday life, being applied at large scale in the industry, at home and in the public domain. Millions of sensors and actuators

connect critical industrial processes and our everyday lives to the internet. At the same time, as discussed in previous chapters, the software on IoT devices is evolving to become more complex and dynamic. While the added benefits are clear, allowing dynamic reconfiguration and deployment of software can be a security hazard. The omnipresence and quantity of IoT devices in the field in combination with an increase in software dynamism makes IoT platforms an interesting target for malware and hackers.

Malware is a critical and growing threat to the IoT. The StuxNet [54] worm was the first high-profile example. StuxNet damaged an Iranian uranium enrichment facility by spoofing rotation sensor data from an enriching centrifuge, causing a motor actuator to increase its speed until the centrifuges it controlled were destroyed. More recently, the Mirai malware used IoT devices to create a botnet that mounted a massive scale Denial of Service (DoS) attack, which peaked at over 1 Terabit per second (Tbps) [103].

Despite the clear danger, the vast majority of deployed IoT products provide little or no protection against malware and even basic features such as memory protection, are typically not available on resource constrained IoT devices. Ronen et al. recently showcased these weaknesses through the creation of a rapidly spreading worm for the Philips Hue smart light bulbs [84].

There are a range of well-known security techniques that could be used to address the problem of IoT malware, such as secure software deployment [97] and remote attestation [31, 13], where the state of a device can be remotely attested at all times and any tampering can be detected. However, the application of these techniques is complicated by the limited security features of contemporary IoT microcontrollers, and typically require costly hardware add-ons or a complete platform redesign. The lack of memory protection is particularly problematic, as it allows malware full access to all device resources.

This chapter addresses the problem of IoT malware by introducing the concept of a *Security MicroVisor*, which uses selective software virtualization and assembly-level code verification to isolate a software-based Trusted Computing Base (TCB) from untrusted application software. SµV works with all standard microcontrollers that support global interrupt disabling, are single threaded and have sufficient flash to support the preinstalled SµV module. These features are offered by all microcontrollers used in contemporary IoT products.

The software-based TCB created by SµV provides the isolation required for secure software deployment and remote attestation. This chapter showcases how these secure operations leverage the memory isolation that SµV provides to run on unmodified resource constrained IoT devices.

6.2 Design of SμV, the Security MicroVisor

This section provides an overview of the software mechanisms that SμV uses to provide memory isolation, remote attestation and secure deployment.

The design of SμV shares similarities with the Software-based Fault Isolation (SFI) approach proposed by Wahbe et al. [105]. SFI prevents faults in untrusted software modules from corrupting other software on processors with a single shared address space and no memory protection. For each software module, SFI reserves a logically separate portion of the application's address space. The isolation of module address spaces is maintained at runtime by rewriting unsafe instructions to verify target addresses. SμV also uses selective software virtualization and assembly-level code verification to ensure full isolation of the trusted software module (SμV) from untrusted application software.

Attacker model The adversary is assumed to have full access to the network, but cannot physically tamper with the IoT device. The attacker can communicate with the IoT device over the network or prevent legitimate communication from occurring. In other words, SμV only guarantees that secure operations like remote attestation occur *securely*, it cannot prevent an attacker from blocking the attestation process or otherwise rendering the IoT device unavailable. Furthermore, the trusted SμV is assumed to be bug and exploit free and is pre-deployed on the device by a trusted party.

6.2.1 The platform requirements of SμV

The vast majority of IoT devices are based on off-the-shelf Micro Controller Units (MCUs), that are optimized for low cost and low power operation. Hardware security features and Memory Protection Units (MPUs) are uncommon on these devices and millions of IoT devices are already deployed without these features. This essentially means that a piece of software running on such a device can execute arbitrary instructions and read or modify both data and instruction memory.

SμV is a pure software solution which allows additional security features to be added to conventional microcontrollers. SμV will be used to provide MPU-like memory protection and support for remote attestation and secure deployment. These basic MCUs are found in the majority of low-end IoT devices on the market, and have the following characteristics:

1. **No memory protection**: The MCU is not required to provide any form of memory protection. Nor is it required that the MCU provides ROM memory. The MCU must, at a minimum provide sufficient flash memory to store the SμV MicroVisor (under 4 KB for typical MCU architectures).

2. **Single thread of execution**: The MCU should only support a single thread of execution. This is to guarantee atomic execution of critical code without preemption by other threads. This is typical for conventional microcontrollers.

3. **Interrupts**: The MCU must support the disabling of global interrupts to ensure the atomic execution of code without preemption by interrupt handlers. This feature is offered by all major families of MCUs.

6.2.2 Architecture of SμV

SμV reserves part of the memory for a Trusted Computing Base (TCB) called the MicroVisor. This software is installed prior to the deployment of the IoT device using a physical programming device (e.g. SPI or JTAG). The MicroVisor code is considered immutable and resides in *virtual* ROM memory, which is enforced by the MicroVisor itself. The remainder of the device memory, from now on referred to as *Application Memory*, is available to untrusted applications. Application memory is further subdivided into *Instruction Memory*, which applications can execute but not read or write to and *Data Memory*. This is visualised in Figure 6.1.

The trusted MicroVisor code is subject to no restrictions. Untrusted applications on the other hand are strongly restricted in the following ways:

1. **Control transfer**: branch and jump operations can only address the application instruction memory or the select entry points of the MicroVisor instruction memory which expose virtual operations. This allows for controlled interaction with the MicroVisor.

2. **Data memory access**: read and write operations can only address application data memory or Memory Mapped IO (MMIO) locations.

3. **Instruction memory access**: read and write operations can not address the application instruction memory or the MicroVisor memory.

4. **Deployment** of new applications can only occur through the MicroVisor. This property is enforced by preventing applications by the previous restriction that disallows an application to write in its own instruction

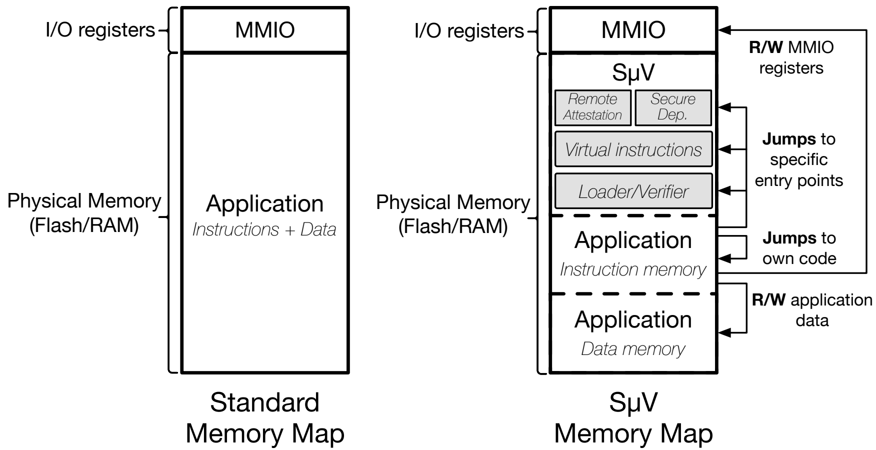


Figure 6.1: Standard (left) and SµV (right) memory map. In the standard case, memory is monolithic and operations are unrestricted. SµV splits application memory into instruction and data memory and restricts sensitive operations.

memory, only the MicroVisor is allowed to do so. As a result, all new applications pass through the MicroVisor during loading.

Restrictions on application code are enforced at the instruction level through two basic mechanisms: i) incoming applications are *verified* by the MicroVisor at load-time to ensure that they adhere to the rules listed above, and ii) certain inherently unsafe instructions which are nonetheless essential for normal operation are replaced by safe *virtualized instructions*.

Verifier and loader

As described above, application deployment can only occur through the MicroVisor, which subjects the application code to assembly-level verification at load time on the embedded IoT device. Only *safe* instructions are allowed, i.e. instructions that do not violate the above memory restrictions. Two types of illegal instructions can be distinguished: i) instructions that statically jump to or access restricted memory, and ii) instructions that jump to or access *any* memory dynamically and cannot be checked statically.

Most control transfer instructions, such as program-counter relative branches and calls, have their target address encoded in the instruction and can be checked

```
1 app_function:
2     ...
3     // Load pointer register r30 with address 0xF512
4     load r30, 0xF512
5     // Calls the function r30 is pointing to
6     icall
7     ...
8     ret
```

Listing 6.1: Unsafe indirect call through pointer register. The icall instruction is inherently unsafe and will be rejected by SµV during verification.

statically by the verifier at load time. Store operations to static variables also use an immediate addressing mode and can be checked by the verifier. Any instruction of this type that has an illegal memory address as static argument is detected and this results in the application being rejected by the verifier, canceling its deployment.

Applications that contain instructions which cannot be statically checked are rejected outright. Instructions using indirect addressing belong to this category, such as jumps and stores that use a pointer register to hold their target address. These are common when using pointer logic or arrays in C. An example of such an illegal instruction is shown in Listing 6.1, where a function is called through a pointer register. As supporting these operations is essential to normal operation of any processor, they must be replaced prior to deployment with calls to secure *virtualized* instructions provided in the MicroVisor that perform runtime checking of their arguments. SµV offers toolchain support that transparently replaces these operations for the developer.

Secure virtual instructions

The MicroVisor offers replacements for all unsafe dynamic instructions, which can be accessed via a call to a subroutine in the MicroVisor. These virtual instructions check their arguments and will perform the matching operation, only where it does not break memory access or control transfer rules. Following the execution of the virtual instruction, the MicroVisor returns control to the application. Any operation which attempts to access an illegal memory addresses is trapped and causes the microcontroller to reset. Using this approach, the security features of the MCU are enhanced, without sacrificing functionality.

```
1 app_function:
2     ...
3     // Load pointer register r30 with address 0xF512
4     load r30, 0xF512
5     // Call suv_safe_icall residing in SuV, checking
6     // the contents of r30 before jumping
7     call suv_safe_icall
8     ...
9     ret
```

Listing 6.2: Safe indirect call with at runtime checking of the dynamic argument by a subroutine in SμV.

```
1  suv_safe_icall:
2    // Clear global interrupt flag
3    cli
4    // Check validity of target address in r30
5    comp r30, 0xF000
6    branch_gt icall_failure
7  icall_success:
8    // Success, re-set global interrupt flag and jump to target
9    sei
10   ijmp
11 icall_failure
12   // In case of failure, soft reset the MCU
13   jmp 0x0000
```

Listing 6.3: Example virtualized indirect call subroutine in SμV. In this case targets above 0xF000 are refused.

Listing 6.2 shows how the unsafe indirect call instruction from the running example is replaced by a safe virtualized alternative in the form of a subroutine call to the SμV which is known to be safe and is therefore not rejected by the verifier. Listing 6.3 shows an example implementation for the secure virtualized indirect call operation residing in the SμV. The virtualized alternative will first disable global interrupts to ensure atomic verification of the target address. This is essential, as the adversary could schedule an interrupt in order to interfere with the outcome of the check. The check in this example deliberately trivial: any address under `0xF000` is considered a valid target. Next, if the target address is valid, global interrupts are re-enabled and a jump to the target is carried out.

6.2.3 SμV toolchain modifications

SμV provides a modified toolchain which allows the application developer to write software for the SμV architecture transparently and with the same ease-of-use as the underlying MCU architecture.

In a standard toolchain, application code passes through multiple tools: first the compiler produces human readable assembly files. These are processed by the assembler resulting in binary object files. Lastly, the linker combines all object files together with relevant libraries in a single binary image that can be deployed on the microcontroller.

The changes required to this pipeline for SμV are minimal, and are visualized in Figure 6.2. Firstly, between the compiler and assembler stages, a post-processor is added which substitutes all unsafe dynamic instructions with calls to their secure virtualized equivalents. Since this is performed when the application is in text ASM form rather than binary, simple regular expressions will suffice and no custom tools are required. Secondly, in the linker stage the addresses of the functions residing in the SμV are injected in the form of a symbol table. The SμV is preinstalled on the microcontroller, and the application must be linked against these functions at their given addresses.

It should be noted that all libraries must be recompiled with the previously mentioned substitutions carried out. Failing to do so will create an image where unsafe instructions appearing in library functions. These functions will be rejected by the load-time SμV verifier.

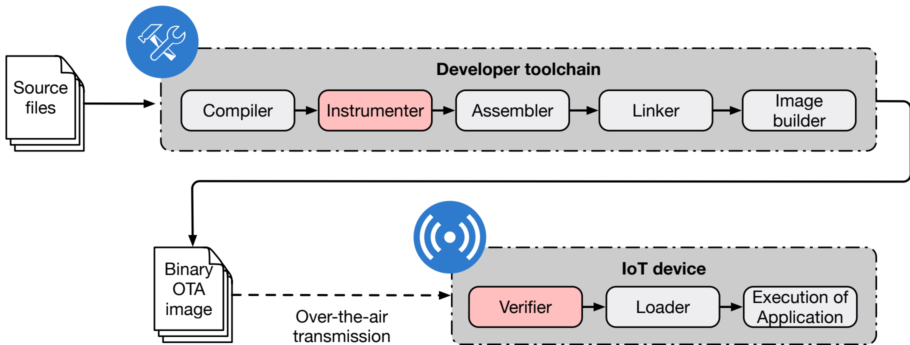


Figure 6.2: The modified toolchain for SμV contains two extra components: an Instrumenter in the developer toolchain, and a Verifier running on the IoT device.

Additionally, the security properties of SµV are maintained even when an adversary uses their own tool-chain or writes hand-crafted assembly because the load-time verification of applications occurs in the MicroVisor on the embedded device itself.

6.3 Remote attestation

Remote attestation is described as a protocol that allows a Challenger to check the internal state of an untrusted Prover remotely over the network. The purpose of this protocol is to allow an uncompromised Prover to create a token that proves to the Challenger that it is indeed uncompromised and in an expected internal state. Conversely, if the Prover is compromised, the token should reflect this. A more formal definition is given by Francillon et al. [34]:

Definition of remote attestation A protocol $\mathcal{P}$ is comprised of the following components:

- Setup(1^κ): A probabilistic algorithm that, given a security parameter 1^κ, outputs a long-term key k. This key is shared between both parties, and is preinstalled on the Prover during commissioning.
- Attest(k, n, s): A deterministic algorithm used by the Prover that, given a pre-shared key k, a nonce n (provided by the Challenger) and internal state s, outputs an attestation token α.
- Verify(k, n, s, α): A deterministic algorithm used by the Challenger that, given a pre-shared key k, a nonce n, an internal state s, and an attestation token α, outputs 1 iff α reflects state s in the Prover (i.e. Attest(k, n, s) $=$ α), and outputs 0 otherwise.

These components are used in the protocol $\mathcal{P}$ between Challenger and Prover as follows: i) both Prover and Challenger possess a pre-shared long-term key k generated by Setup(1^κ) prior to deployment, ii) Challenger requests proof of the state of Prover and generates a nonce n, iii) Prover runs Attest(k, n, s) with s its current state, returning the resulting attestation token α to Challenger, iv) Challenger runs Verify(k, n, s, α), with s the expected state. The output of Verify proves state s of Prover.

Applying SµV to support secure remote attestation Francillon et al. [34] further analyze protocol $\mathcal{P}$ and its security characteristics, and define a list of

minimal properties that are required to support remote attestation, for which they argue that specialized hardware is necessary. To prove that SµV can provide equivalent support in software, each property is listed below with an explanation on how MicroVisor accomplishes it.

1. **Invocation from Start**: The Attest routine should only be invoked from its first instruction. This is accomplished by placing it in the SµV ROM and allowing it to be called from the application when attestation is required. As with all other routines residing in the SµV, only a call to the entry point is allowed, forcing Attest to be run from the very first instruction.

2. **Exclusive access to secret k**: The secret k should only be accessible by the trusted remote attestation code. This is achieved by placing it in the SµV ROM alongside the attestation code. The untrusted application cannot read from this memory area.

3. **Uninterruptibility**: Even on a single threaded platform, the untrusted application can regain control after invoking Attest using interrupts (e.g. a timer expiring). This can cause unintended side effects such as the leaking of k and false positives. All major microcontroller families allow global interrupts to be temporarily turned off. This functionality is used to ensure atomic execution of Attest.

4. **Immutability**: The Attest code cannot be modified by untrusted code before invocation. This is guaranteed by placing the code in the SµV virtual ROM and executing it in-place. Untrusted application code is not allowed to modify this memory area.

5. **No leaks**: Under no circumstance should invoking Attest leak the secret k or any by-products except for the final return value α. This is guaranteed by above properties and additionally implementing the Attest routine in a way that erases these sensitive values from memory before returning.

From a practical point of view, Attest and Verify depend on computing a Message Authentication Code (MAC) of the state to be attested. The contents of flash, RAM memory, registers and any other volatile or non volatile memory can be considered state. When the Prover receives a request from the Challenger to attest a region of memory containing state s with nonce n, Attest is called to compute the MAC α of $(s||n)$ using pre-shared key k. The nonce n should be used only once and is essential to avoid replay attacks. The computed token α is sent back to the Challenger, where Verify computes the MAC of $(s||n)$ once again, this time with s the expected state of the segment of memory. If the

computed MAC matches token α, the Prover has the expected state. If the MAC differs, the Prover's memory is compromised and necessary measures should be taken such as performing secure erasure.

6.4 Secure deployment

Due to changing application requirements and emerging security threats, the remote updating of deployed IoT devices is a necessity. While remote updates offer flexibility, they also increase attack surface as attackers can misuse this mechanism to install unsanctioned software remotely.

Secure deployment refers to the process of setting up new software on a device, without compromising the security of the device. Texas Instruments [97] identified four security threats that can affect the software deployment process: i) firmware alteration, ii) firmware reverse engineering, iii) loading unauthorized firmware, and iv) loading authorized firmware onto unauthorized devices. Accordingly, three security attributes are specified that must be fulfilled in order to secure the deployment process and overcome these threats: i) data confidentiality, ii) authenticity, and iii) integrity.

The solution presented in this work, defines an initial version of secure deployment that does not provide protection against firmware reverse engineering and consequently does not fulfill the confidentiality requirement. Since SµV targets resource constrained IoT devices, the overhead of encrypting the deployment process is high. As such, confidentiality should be added as an extra feature depending on the sensitivity of the application domain (e.g. military), and the primary focus is to avoid tampering through the software update mechanism. Confidential software deployment can be one direction of future work. This initial version of secure deployment allows an Updater to verify that a software image received over the network is uncompromised and issued by an authorized Issuer. More formally, secure deployment is defined as follows:

Definition of secure deployment A protocol $\mathcal{Q}$ is comprised of the following components:

- Setup(1^κ): A probabilistic algorithm that, given a security parameter 1^κ, outputs a long-term key k. This key is shared between both parties, and is preinstalled on the Updater during commissioning.
- Sign(k, i): A deterministic algorithm used by the Issuer that, given a pre-shared key k and an image i, outputs a token β.

- Verify(k, i, β): A deterministic algorithm used by the Updater that, given a pre-shared key k, an image i, and a token β, outputs 1 iff β reflects image i in the Issuer (i.e. Sign$(k, i) = \beta$), and outputs 0 otherwise.

These components are used in the protocol $\mathcal{Q}$ between Issuer and Updater in this manner: i) both Issuer and Updater possess a pre-shared long-term key k generated by Setup(1^κ) prior to deployment, ii) Issuer issues an updated image i, and generates token β using Sign(k,i), iii) Updater receives image i and token β, and runs Verify(k, i, β). The output of Verify determines if Updater will install the software update or not.

Applying SμV to support secure deployment The minimal set of properties required to run Verify on the Updater uncompromised and without leaking the secret k, coincides with the properties discussed previously in section 6.3, namely: *invocation from start*, *exclusive access to secret k*, *uninterruptibility*, *immutability* and *no leaks*. As argued above, SμV can guarantee these properties without any specialized hardware support.

Sign and Verify rely once again on calculating a MAC of the entire update image and its meta-data (i). The Issuer uses Sign to compute the MAC (β) of the image using a pre-shared key (k). The complete image along with β is transmitted to the Updater, where Verify computes the MAC once more using k. If this newly computed MAC does not match β, the image or its meta-data has been tampered with in transit (integrity), or the Issuer did not posses the correct key k (authenticity).

In addition to verifying the above security attributes, the instructions in the target binary image should not violate SμV rules discussed in section 6.2. This is ensured by the SμV verifier.

6.5 Implementation

A prototype of SμV has been implemented for the MicroPnP IoT platform [111], which offers an IEEE 802.15.4e [107] radio and an 8-bit AVR ATmega 1284p [7] microcontroller running at 10 MHz, with 16 KB of SRAM and 128 KB of flash. The AVR ATmega family of microcontrollers have a long track record of being used in IoT platforms, including the Zigduino [59], the AVR Raven [6] and Berkeley Mica Mote [42]. The previous section discussed the design and general operation of SμV. Figure 6.3 shows an overview of how SμV is implemented on the AVR architecture, which necessitated several adjustments as outlined below.

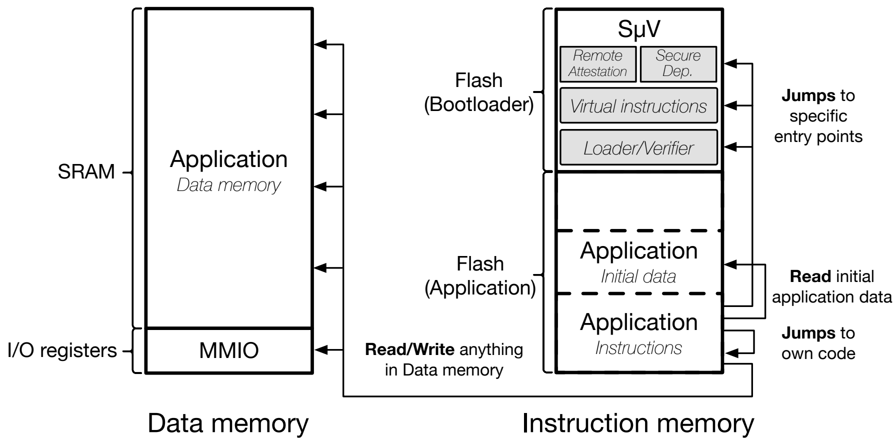


Figure 6.3: SμV memory map for modified Harvard architectures. Data memory is left unprotected, as code can not be executed from there. The SμV is placed in the bootloader segment at the end of the instruction memory, otherwise the Loader/Verifier can not write to instruction memory when loading code.

6.5.1 Modified Harvard architecture

Section 6.2 assumed microcontrollers with the generic *von Neumann* architecture, where flash, RAM and any MMIO peripherals are mapped on to a single address space. The AVR family of microcontrollers use a *modified Harvard* architecture, where instruction and data memory are physically separate. In the case of the AVR, this means that the flash memory holding the instructions and the volatile RAM containing the data have an isolated address space. In a strict Harvard architecture, the instruction memory cannot be read from or written to, and instructions cannot be executed from data memory. However, the AVR uses a *modified* Harvard architecture, which relaxes this restriction by offering special instructions to read and modify instruction memory. This allows self programming and storage of data constants in flash.

Implications for SμV Due to the modified Harvard architecture of the AVR, the approach can be simplified to only protect instruction memory, while data memory operations remain unmodified and unrestricted. In order for this to work, SμV data is placed alongside SμV code in instruction memory, which is permitted by the modified Harvard architecture. The application is not allowed to read the SμV code or data from the instruction memory, and is only permitted to jump to itself or select SμV entry points. These restrictions on the

instruction memory, in addition to the fact that data memory is not executable are sufficient to offer the same security guarantees previously mentioned. The techniques used to restrict these instructions are identical to those used on a *von Neumann* architecture. Leaving the data memory unprotected has the additional advantage that operations dealing with data memory do not incur any runtime overhead due to SµV. Figure 6.3 shows the general memory map of both instruction and data memory, together with the class of operations that are allowed by application code. If code contains disallowed operations, it is rejected by either load-time verification or runtime virtualized operations.

6.5.2 Bootloader

Instruction memory in the AVR is further subdivided into an application and a bootloader section. Code residing in the bootloader section has special privileges over application code. The application can only read from instruction memory, while the bootloader code can read and write instruction memory. Any self-programming code has to be located in the bootloader. The size of the bootloader section is configurable before deployment of the IoT device using a physical programmer, but can not be modified at runtime.

Implications for SµV The *Loader/Verifier* component of SµV requires write privileges to instruction memory. Therefore the natural place of the SµV Trusted Computing Base is in the bootloader section of the instruction memory. The application is loaded in the application section, and as a result has no write privileges. This is convenient, as this does not need to be enforced through virtualized instructions. The read behaviour of the application still needs to be monitored, however, applications should not be able to read SµV instructions or data from the bootloader section. Figure 6.3 shows the placement of the SµV core in flash memory.

6.5.3 Initialization of data memory

At boot time, the volatile data memory is empty. In order to comply with the C standard, variables with an initializer should have their value assigned before any code execution. In order to do this, the instruction memory will hold all initial data memory values in addition to the application instructions. Immediately before the application executes, bootstrapping code will copy the initial values from instruction memory to data memory.

Implications for SµV Special care should be taken that no jumps from the application code to the initial data stored in instruction memory are made. The data stored in instruction memory can include illegal instructions that could be misused by an adversary to attack SµV. As the compiler always places the initial data in instruction memory straight after the application's instructions, only the address of the last valid instruction should be transmitted as extra meta-data at load time. Any jump after that address is either invalid or a bootloader entry point. This is also visualized in Figure 6.3.

6.5.4 Two word instructions

The AVR has a variable length instruction set. A standard AVR instruction is 2 bytes (1 word) long. As a result, the program counter and all jumps can only point to even bytes in the flash. Some instructions however require 2 words, with the 2nd word being a target address in either data or instruction memory.

Implications for SµV There is a possibility that the 2nd word of an two word instruction unintentionally forms an unsafe normal length instruction. While these unsafe 2nd words appear inside the application's instructions, they should not be jumped to. This is accomplished by maintaining a list of unsafe 2nd words, which is enforced by both the load-time verification for static branches and jumps, and by the run-time virtualized instructions for dynamic jumps. While it is possible for SµV to generate a list locally on the node when the application is updated, an extra step is added to the tool-chain to pre-generate this list at compile-time. The list is added to the meta-data shipped within the application image, and is checked by the SµV for validity at load-time. Applications with an incomplete list of unsafe 2nd words are rejected. Pre-generating the list reduces the overhead of loading and verifying a new application and moves it from the device to the machine generating the application image.

6.5.5 Remote attestation and secure deployment

As discussed previously, remote attestation and secure deployment require the computation of a Message Authentication Code (MAC) over either state or an update image. In this implementation, a highly optimized HMAC-SHA1 implementation in AVR assembly is used to optimize performance and reduce space required in the SµV ROM. HMAC-SHA1 returns a 160 bit keyed hash, and the pre-shared cryptographic key is also 160 bits long.

The selection of HMAC-SHA1 deserves extra clarification after the recent publication of SHAttered, a first real-world SHA1 collision resulting from joint work of Google and CWI Amsterdam [38]. Theoretical attacks on SHA1 have been known since 2005, but SHAttered is the first practical attack on the hashing algorithm. While SHA1 is no longer collision free, HMAC is significantly less sensitive to collisions of the underlying hash algorithm. Until today, HMAC-SHA1 is still secure and not breakable. Another prime example of HMAC's added resistance to collisions is HMAC-MD5. MD5 has known flaws since 1996, but HMAC-MD5 is still collision free today. Taking this in consideration, this work has no strong dependency on HMAC-SHA1 specifically and can easily switch to other hashing algorithms like HMAC-SHA265 in the future, albeit with a slight decrease in performance.

HMAC relies on a pre-shared key. While a solution with public-key crypto using digital signatures can give better security guarantees and facilitate key management, overhead in computation time and binary code size has proven to make this approach unfeasible for the low-power devices SµV targets.

On the AVR platform, state can be stored in flash, SRAM, CPU registers and EEPROM. While remote attestation can be used on any of these memory types, this implementation specifically focuses on attesting the flash memory. The reasons for this are twofold: i) flash is the only memory from where code can be executed on a Harvard architecture, and as a result poses the biggest security risk when compromised, and ii) the flash memory on the ATmega 1284p is the largest and slowest memory, and thus provides a good benchmark for worst-case performance.

6.6 Evaluation

SµV is evaluated by implementing reference applications and measuring the overhead imposed by the Security MicroVisor as well as the remote attestation and secure deployment techniques that are implemented on top of it. More specifically, 3 key performance indicators are used: i) development overhead, ii) application deployment overhead, and iii) runtime overhead: execution time, battery life, and memory footprint. For every performance metric, the overhead imposed by SµV itself is first considered, before analyzing overhead created by secure deployment or remote attestation.

Four reference applications were selected to benchmark performance: i) a cryptographic application which encrypts and decrypts a random 8 byte cleartext with a 128-bit key, using a software implementation of the lightweight SPECK block cipher [9], ii) the same crypto application, but implemented in a modular

fashion by introducing indirect calls through pointers, iii) sampling temperature readings from a SHT25 Sensirion sensor over the I2C bus, and lastly iv) writing and reading a block of 256 bytes to the built-in EEPROM. These reference applications provide a good balance between the more computationally intensive and the more IO intensive tasks that are typical for an IoT device. The pointer version of cryptographic application is included to provide the worst case overhead for SµV.

6.6.1 Development overhead

From the application developer's point of view, no development overhead will be perceived after the installation of the SµV toolchain. The modified toolchain will transparently replace unsafe instructions with secure virtual alternatives, link to the pre-installed Security MicroVisor and generate a binary image with the correct meta-data. Any security features embedded in the SµV (i.e. remote attestation and secure deployment) will be available transparently.

The time required to port SµV to a different architecture can also be considered development overhead. As of now, SµV is implemented and evaluated solely on AVR microcontrollers. The implementation of SµV took about 120 man-hours. This includes the software running on the microcontroller itself as well as all additions to the toolchain. The time to port to a different architecture is likely to depend upon its inherent characteristics. For example, the variable length of the AVR instruction set causes more overhead than a fixed length instruction set due to the extra bookkeeping mechanisms required. Furthermore, architectures similar to AVR can borrow parts of this initial version of SµV, reducing porting overhead.

6.6.2 Deployment overhead

The deployment of an application to a wireless IoT device occurs in two phases. First, the binary image of the application is wirelessly transferred as a stream of packets. Secondly, the device will go offline to load, and in the case of SµV, verify the image. During wireless transmission of the application image, the size of the binary image is of critical importance for energy consumption and network overhead. During application loading, the time required to load and verify the image determines the total down-time of the wireless device. Secure deployment incurs an additional overhead on top of the basic deployment overhead of SµV.

Over-the-air binary image size Previous research has shown that for battery powered, low-energy wireless devices, remote software updates have a significant impact on battery life [44]. This is not surprising, as sending and receiving data over the radio is typically the most energy consuming activity for any wireless device. The size of the application image is linearly correlated to the total energy spent during the software update, as it determines the time spent actively receiving data over the radio.

For this benchmark, the size of the image for a microcontroller without SµV is compared with the size of an image of the same application for a microcontroller with SµV. Table 6.1 shows the results for each of the reference applications previously introduced. The size overhead for the SµV-enabled images is relatively small and averages at 1.61%. This overhead is caused by two different effects: i) on the AVR, unsafe single word instructions are replaced with a 2 word long instruction that calls a safe virtualized version residing in the MicroVisor, and ii) the SµV-enabled image carries a small amount of extra meta-data, such as a list of unsafe 2nd words and the address of the last valid application instruction. Secure deployment adds 20 bytes additional meta-data to the image in the form of an HMAC-SHA1 digest, and raises the average overhead to 3.58%. The overhead specific to each application is listed in Table 6.1 as well.

Application	Without SµV	With SµV	SµV + Secure Deployment
Crypto	1414 B	1428 B (+**0.99%**)	1448 B (+**2.40%**)
Crypto ptr	1438 B	1458 B (+**1.39%**)	1478 B (+**2.78%**)
Sense temp	1012 B	1034 B (+**2.17%**)	1054 B (+**4.15%**)
Storage R/W	640 B	652 B (+**1.88%**)	672 B (+**5.00%**)
Avg overhead		**1.61%**	**3.58%**

Table 6.1: Overhead of SµV on over-the-air binary image sizes.

Loading and verification time During the transmission of the binary image over the wireless network, the device will remain online and operational, executing the existing application. Once the application is transmitted, the device will go offline for the duration of the verification and installation of the new application. It is important that this time is minimized in order to maximize device up-time. For a microcontroller without SµV, verification is a simple routine that checks if all parts were transmitted, and in case of unreliable network communication all chunks will be check-summed to rule out a corrupted image. On a SµV-enabled microcontroller, verification additionally includes a static check of all instructions of the application, and a validity check of the

meta-data transmitted with the image (i.e. the address of last valid instruction and a list of unsafe 2nd words).

Secure deployment adds additional overhead caused by the calculation of a HMAC-SHA1 keyed hash of the entire image and its meta-data, which is compared to the digest included with the transmitted application image. This further increases the load time of the image.

Table 6.2 shows local verification and load times for all reference applications, both with and without the Security MicroVisor. Additionally, the load overhead of secure deployment with SµV is shown. The relative overhead introduced by the extra verification of SµV for the test applications averages at 4.16%, which is minimal. When adding secure deployment, the relative overhead rises to 80.42% due to the computationally intensive calculation of the HMAC-SHA1.

Application	**Without SµV**	**With SµV**	**SµV + Secure Deployment**
Crypto	128.2 ms	133.8 ms (+**4.3%**)	231.2 ms (+**80.3%**)
Crypto ptr	128.6 ms	134.3 ms (+**4.4%**)	231.7 ms (+**80.2%**)
Sense temp	91.8 ms	96.0 ms (+**4.6%**)	171.0 ms (+**86.2%**)
Storage R/W	73.5 ms	75.9 ms (+**3.3%**)	128.6 ms (+**75.0%**)
Avg overhead		**4.16%**	**80.42%**

Table 6.2: Overhead of SµV on node-local load times.

While the local verification and load time overhead of secure deployment is high, end-to-end deployment overhead including the transmission of the image over a low energy duty cycled network is significantly lower due to the time required to transmit the application image over the wireless network. To quantify the effect of network throughput on remote software updates, a simulation was run on the IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) [107] network used by MicroPnP. In TSCH-based networks, time is divided into time slots long enough for a packet to be transmitted. These time slots avoid packet collisions and are assigned to links between devices. Bandwidth is configurable by assigning more or less time slots to a link.

Figure 6.4 shows the total end-to-end deployment time and the overhead imposed by secure deployment through a standard[1] TSCH network with various bandwidth assignments. Overhead is typically smaller than 2.5%, except for the *Crypto pointer* application, where overhead is closer to 8% due to an extra packet transmission.

[1] 10 ms slot length, 90 bytes MTU, 101 slots/slotframe

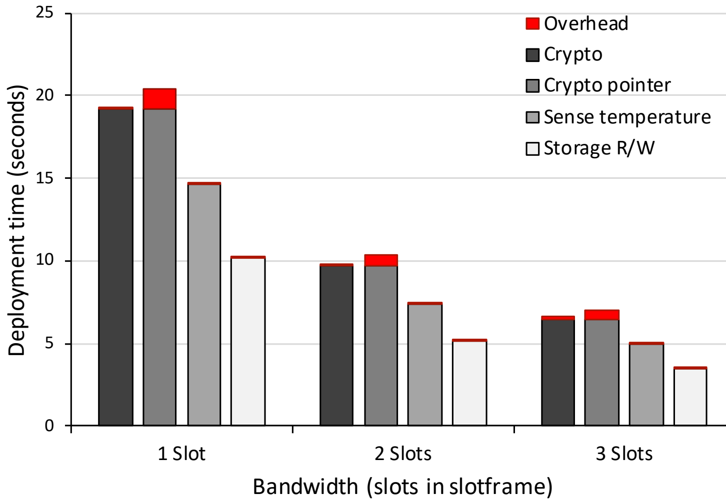


Figure 6.4: End-to-end secure deployment overhead for different bandwidth allocations in a TSCH network.

6.6.3 Runtime overhead

Finally, the runtime overhead in terms of execution time, RAM overhead and flash is evaluated.

Execution time Execution time is directly correlated with the battery life of low power microcontrollers. Typically, after the application is done with its tasks, the microcontroller is put into a sleep mode to minimize current draw. The execution time of the application determines how much time is spent in the high current active state.

Once again the execution times of all reference applications with and without SµV are compared, with the results shown in Table 6.3. On average, the relative execution time overhead amounts to 2.67%. However, there is a difference between the more computationally intensive tasks (i.e. Crypto), and the more IO intensive tasks (i.e. temperature sensing and storage read/write). The IO intensive tasks spend a relatively large amount of time busy waiting for operations to complete, while the computationally intensive tasks are continually executing instructions. Due to this, the relative overhead for computational tasks is higher than for IO intensive tasks.

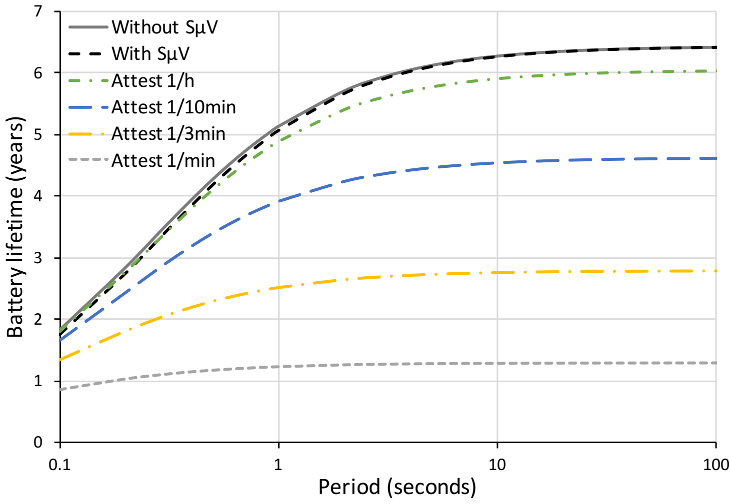
Application	Without SµV	With SµV	Relative overhead
Crypto	3.9460 ms	4.1695 ms	5.66%
Crypto ptr	3.9469 ms	4.1706 ms	5.67%
Sense temp	65.9048 ms	65.9364 ms	0.05%
Storage Write	859.6278 ms	859.6897 ms	0.01%
Storage Read	0.4382 ms	0.4468 ms	1.96%
Avg overhead			**2.67%**

Table 6.3: Overhead of SµV on the execution times of the sample applications.

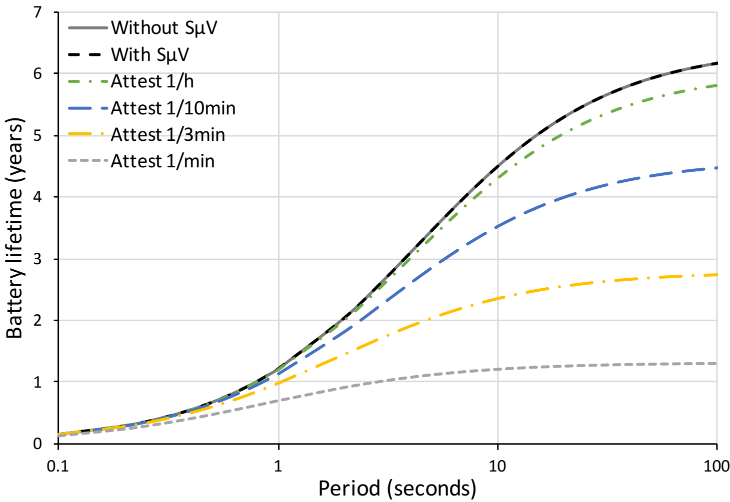
As previously argued, the execution time of tasks is directly correlated with energy consumption. Longer execution times cause the microcontroller to remain in a higher power state for a longer amount of time, draining the batteries faster. The MicroPnP platform on which the benchmarks are conducted consumes 3.54 mA when executing a task and 54.5 µA when idle. Every MicroPnP device is powered by a standard 3000 mAh battery pack. Based upon these values, an estimation of the device battery lifetime is plotted for each task relative to the rate at which it is scheduled. Note that the network is a constant in this benchmark as no data is transmitted and only the energy consumed by the node-local code execution is considered.

Figure 6.5 shows the impact of SµV for the worst-case CPU intensive *Crypto pointer* application, and the more IO intensive *Sense temperature* application. The baseline battery lifetime if the application is sleeping constantly is 6.5 years. In general, the *Sense temperature* application has worse battery life and produces a more S-shaped curve. This is due to the busy waiting that is associated with IO intensive tasks, keeping the microcontroller in an active state for a longer time. When comparing identical applications with and without SµV, a marginal overhead (<1%) is measured for the CPU intensive *Crypto pointer* application at high scheduling rates, which disappears when the application is only scheduled once every 10 seconds. For the IO intensive *Sense temperature* application, SµV overhead is imperceivable on the graph whether the application is scheduled every 100ms or every 100s. In most real world IoT applications, long IO intensive tasks will dominate over brief CPU intensive tasks, making any measurable reduction of battery life caused by SµV unlikely.

Remote attestation causes additional active CPU time, and as a result an increase in energy consumption. Remote attestation execution times depend on the amount of flash memory attested. Attesting the entire 128 KB of flash memory takes 7.6 seconds, attesting just the untrusted application (62 KB) takes 3.7 seconds. The impact on battery life largely depends on the frequency



(a) Crypto pointer application



(b) Sense temperature application

Figure 6.5: Graph showing estimated battery life for a computationally intensive application (Crypto) and an IO intensive application (Sense temperature), relative to the period at which they perform their tasks (X-axis). Different curves represent battery life without and with SµV, and of increasing rates of remote attestation.

of remote attestation. Figure 6.5 shows the battery lifetime of our reference applications when remotely attested at rates ranging from once every minute to once every hour. For both applications, an attestation rate of once every hour incurs a worst-case reduction in battery lifetime of 6.2%. The effect of a higher attestation rate is less prominent for applications with a higher sampling rate, where the energy consumption of the primary task overshadows attestation energy consumption.

RAM memory SµV incurs no static RAM overhead as all constants are stored in flash. The stack is used for short term data storage when calling any subroutine in the SµV (i.e. any virtual instruction, remote attestation or secure deployment). In order for these subroutines to properly function, the application can not use all available stack space. For correct basic operation of the microcontroller, a minimum amount of 13 bytes of free stack space is required at all time for the virtual instructions. Remote attestation and secure deployment inherently need more stack space to temporary store full pages of flash and maintain intermediate states of the cryptographic functions, and require 511 bytes and 807 bytes respectively. This is a compromise where memory usage is traded for speed. Loading pages in smaller chunks is possible, but more time consuming.

Flash memory When evaluating deployment overhead, the extra size of the application to be installed in flash is measured. Additionally, less total space will be available in flash due to space reserved for the preinstalled SµV.

The reduction of available space is shown in Table 6.4. The total space occupied by the SµV can be broken down in its components. The *SµV core* contains everything required to ensure memory isolation: the verifier, the loader, secure virtualized operations and any required data constants, and amounts to 1070 bytes or a marginal 0.82% of the total flash memory available. The *Remote attestation* and *Secure deployment* anti-malware modules contained in the SµV consume an additional 242 bytes and 462 bytes, respectively. Lastly, the crypto library containing the *HMAC-SHA1* algorithm used by both remote attestation and secure deployment uses 1296 bytes of flash. Total flash used amounts to 3070 bytes or 2.34% of the total 128 KB flash available on the platform.

As explained previously, on the AVR architecture the SµV has to be installed in a special bootloader section of flash memory. The bootloader's size is configurable. The Security MicroVisor comfortably fits in the 3rd smallest size of 4096 bytes, occupying 74.95% of bootloader space.

Component	**Size**	**% of flash (128 KB)**
SµV core	1070 B	0.82%
HMAC-SHA1	1296 B	0.99%
Remote attestation	242 B	0.18%
Secure deployment	462 B	0.35%
Total	**3070 B**	**2.34%**

Table 6.4: Flash memory utilization

6.7 Discussion

The techniques embodied by SµV can be used to implement a wide range of security features on conventional microcontrollers. This section first discusses the role of SµV in the development of an IoT platform and then highlight high priority security features that can be realized using the SµV approach.

6.7.1 Promoting the adoption of MicroVisors

The security techniques contributed by this dissertation demand simple and reliable toolchain support across a range of processor architectures to facilitate the use of SµV by third parties. Applying SµV should be as simple for the developer as modifying the compile target and flashing a Trusted Computing Base (TCB) to their IoT device.

Looking beyond the current implementation, SµV is built in a way that promotes third-party researchers developing new MicroVisors which offer novel security properties that go beyond the features provided by SµV today. To foster this activity, the SµV core implementation is subdivided into a generic and reusable set of libraries and tools for building new forms of MicroVisor (i.e. a MicroVisor development kit).

Systematic support for the *formal verification* of MicroVisor properties provides vital assurance of security properties. In this regard, formal verification tools such as VeriFast [50] hold great potential. A current working point is applying this approach to verify the security properties of the SµV TCB, and also including this in a MicroVisor development kit to ensure that third-party extensions are also secure.

6.7.2 SμV and component-based middleware

This chapter introduced the Security MicroVisor as a generic technique for enhancing the security features offered by conventional microcontrollers. The MicroVisor technique was showcased by demonstrating that it is capable of providing the properties that are required to support remote attestation and secure deployment. However, the MicroVisor can also be used to support a wide range of security features.

Consider for example, protected module architectures such as SANCUS [72], which provide support for modular and extensible remote attestation of individual software modules. Each SANCUS module securely maintains its local state, while securely interacting with trusted software modules. In its current form, SANCUS achieves these features using a minimal hardware extension to conventional microcontroller architectures. It is also possible however, to use the MicroVisor to implement SANCUS in pure software. The memory and performance overhead of implementing SANCUS using a Security MicroVisor should be reasonable, though higher than remote attestation due to: i) the increased complexity of checking multiple regions of memory with different access rights and ii) the necessity of storing per-module key material.

As discussed in Chapter 2, modularization of software with component-based middleware is indispensable for remote management of large IoT deployments during their full lifecycle. During the lifetime of a device, software components can be added, removed, reconfigured and bound together. The primary drawback of adding this kind of flexibility to constrained IoT devices without any memory protection, paging or privileged/protected execution modes is that they become more susceptible to both malware and buggy components. Any component can access and modify the memory other components or the underlying middleware and operating system.

By combining a pure-software version of the SANCUS protected module architecture with component-based middleware, the inherent risks of memory corruption and security that go along with a component-based middleware are reduced. Every component will be sand-boxed from other components and critical system memory. The realisation of a pure-software secure component-based middleware requires the techniques laid out in this chapter.

6.8 Summary

This chapter introduced the final contribution of this thesis: SµV, the *Security MicroVisor*. As IoT systems become more open and dynamic, securing them becomes a primary concern. SµV tackles the problem of a lack of hardware memory isolation that is common to most resource constrained IoT devices. SµV isolates a trusted software module from untrusted application software, and uses it to contribute the first pure software approach to remote attestation and secure software deployment. The SµV approach rests on three pillars: i) selective software virtualization of the microcontroller architecture, ii) deployment-time verification of incoming code at the assembly level and iii) toolchain modifications which allow developers to transparently compile their software for the virtual SµV architecture.

SµV is compatible with all standard IoT microcontrollers that are single threaded and support the disabling of interrupts and, crucially, as SµV does not require additional hardware security features, the approach can be applied to improve the security of millions of IoT devices that are already in the field in a cost-effective way.

The overhead of SµV is reasonable. Evaluation on the ATmega 1284p shows a modest increase in the size of deployable code at 3.58%. The execution time of application code also increases minimally at 2.67%, and has little effect (<1%) on battery life. Code verification overhead during software updates is likewise feasible for embedded IoT devices, adding an average local overhead of just 4.16%. Secure deployment adds additional verification overhead, but end-to-end impact on over-the-air software load times including network transmissions is acceptable at 8%. SµV-based remote attestation was shown to be feasible when scheduled hourly, with a maximum reduction in battery life of 6.2%.

SµV has useful applications beyond remote attestation and secure deployment. Combining the techniques outlined in this chapter can extend a component-based middleware with component-level isolation, heightening security and lowering the risk of down-time due to software faults in badly written components.

Chapter 7

Conclusion

This last chapter concludes the dissertation. Section 7.1, first reviews the contributions in light of the problem statement. Section 7.2 reflects on how concepts from this dissertation interact with past and present work within the research group, and how parts have made their way into an industrial product. Lastly, Section 7.3 situates the presented work in the big picture and looks ahead into the future of the Internet of Things.

7.1 Contributions in review

The research presented in this thesis focuses on bringing dynamism and security to large-scale IoT infrastructures. As the IoT achieves widespread adoption, the technology is continuously being applied in larger and more dynamic scenarios. To increase return on investment, deployments such as smart cities and Industry 4.0 factories are often required to last up to 10 years on a single battery and run multiple applications with changing requirements throughout their lifetime. Another effect of the increasing presence of IoT technology in our daily lives and within crucial industrial processes, is the growing importance of security.

Contemporary programming abstractions and reconfiguration approaches offer some support for increasing levels of dynamism, but are impractical to use on a large scale due to management complexity and network overhead. Solutions aiming to secure IoT devices require in-depth hardware modifications, which is expensive to implement on the millions of devices currently in the field. The contributions presented in this dissertation improve on the state-of-the-art

by: i) detecting and avoiding inconsistent configuration of distributed IoT applications, ii) reducing the runtime cost and complexity of inspecting and modifying decentralized configuration, and iii) creating a software-only security architecture for representative IoT devices. Special care was taken to maximize the impact of these solutions, while at the same time taking into account the resource constraints of IoT devices.

Chapter 2 presented an in-depth study of state-of-the-art solutions that promote dynamism and security in the IoT. More specifically, programming abstractions, runtime reconfiguration approaches and security through trusted computing were surveyed. LooCI was presented as a case-study of a reflective component model, demonstrating both the merits and shortcomings of component-based software development and reconfiguration. The chapter concluded with a gap analysis of the presented solutions. This analysis revealed that, while the reported solutions are an important step towards a dynamic and secure IoT, they do not provide cost effective and appropriate support for managing distributed applications and securing existing IoT deployments.

Chapter 3 focused on managing misconfiguration in distributed IoT applications. The problem of incompatible decentralized configuration and the mechanisms behind it were first studied in a real-world experiment. To counteract this issue, *Safe reparametrization* was proposed as an extension to reflective component models. Safe reparametrization offers a descriptive language to component developers for expressing implicit configuration dependencies, and uses a network protocol to resolve and enforce these dependencies over a component composition, thereby reducing complexity and avoiding downtime or malfunctions. The runtime overhead of safe reparametrization was shown to be acceptable, while management effort is significantly reduced when compared to exhaustive methods of ensuring safe and correct reparametrization.

Chapter 4 introduced *Refraction*, a solution that significantly lowers the cost and complexity of introspecting and reconfiguring reflective component models in two key ways. Firstly, existing streams of application data are selectively augmented with meta-data, traveling to centralized refractive pools. Refractive pools maintain a partial copy of the state of distributed applications running in the network and allow for low-cost introspection. Reconfiguration is further simplified through reactive policies, which when deployed to a refractive pool automatically trigger reconfiguration based on incoming meta-data. A research prototype, *RxCom*, was implemented and shows that policy evaluation creates no significant performance hit. Further evaluation through a case-study indicates a great reduction of network and development overhead when comparing reflection with refraction.

Chapter 5 presented *Tomography*, which takes the idea of cost-effective reflection

further by recognizing that oftentimes components are queried or reconfigured in groups. Tomography improves upon regular reflection by reimagining the visitor design pattern from object-oriented programming: a probe packet is fired into a distributed component composition, collecting meta-data and enacting change as it visits components. Tomography was validated using a real-world scenario, showing significantly reduced network and management overhead when maintaining applications built from distributed components.

Chapter 6 focused on securing dynamic IoT systems and introduced the *Security MicroVisor* (SµV), a software-only trusted computing base for resource constrained IoT devices, the first one of its kind. SµV partially virtualizes the microcontroller's instructions, effectively providing memory isolation. Subsequently, this memory isolation is used to safeguard the integrity of critical operations and secret key material while concurrently running insecure user applications. While securing resource constrained devices in such a way does cause runtime overhead, evaluation has shown that for typical usage patterns of IoT applications it is minimal.

Together, these contributions provide a middleware that is fit for purpose in a dynamic and safe Internet of Things. While these contributions were not evaluated concurrently, it is important to consider the effect they will have when used in tandem. The complexity and network overhead of remote management will be significantly reduced by *Safe reparametrization*, *Refraction* and *Tomography*. These solutions do require a setup cost involving additional network traffic, but as shown in their individual evaluations, the benefits far outweigh the costs and, in general less network traffic will be generated. This effect will be further amplified as distributed applications grow in scale. Other costs associated with system resources like processing speed (SµV), flash, RAM and the size of over-the-air deployable software components are acceptable when added up, and have a minimal impact relative to the total resources available on representative hardware platforms.

7.2 Future research and industrialisation

This section looks at how the concepts discussed in this dissertation relate to past and present work, either in the context of research or as as an integral part of an industrial product.

Research The ideas developed within the first two contributions of this dissertation closely integrate and build on past research conducted in my research group, DistriNet. More specifically, the Networked Embedded Systems

(NES) taskforce has a long track record of research promoting software dynamism and flexibility for IoT applications. While the presented concepts are applicable to any existing component model, the LooCI component-based middleware, as initially presented in Chapter 2, was used for prototype implementations and evaluation throughout this thesis. LooCI was initially developed by colleagues Nelson Matthys [64], Wouter Horré [43] and Klaas Thoelen [98], but was quickly adopted by the entire taskforce. In a later phase, I became the lead developer and maintainer of the embedded variant of LooCI, and contributed back to the middleware I previously used as a foundation for my research. Any extensions made to LooCI within the context of this thesis were realized as modular add-ons, which future researchers can use or extend if they are valuable for their work.

The last contribution, the Security MicroVisor, was built from the ground up, but has given rise to an interesting new line of research within our group. SµV provides an architecture upon which secure functionality can be built, not only for the IoT, but also for automotive applications, as colleague Mahmoud Ammar [3] is currently doing in his research. Furthermore, with the help from prof. Bart Jacobs and his team, efforts to formally verify core functionality within SµV are showing promising results, and provide a way to formally prove the security properties that SµV provides.

Industry During the final years of my PhD trajectory, I had the opportunity to collaborate with my colleagues on the industrialisation of MicroPnP [111, 27, 65], which incorporates many ideas discussed in this dissertation to create a truly flexible industrial IoT platform. MicroPnP takes the idea of software dynamism and extrapolates it to include hardware in the form of pluggable peripherals. When a peripheral is added or removed, driver components are dynamically loaded, configured and the peripheral is made accessible using standard protocols such as CoAP [91].

MicroPnP was submitted to the IPSO (IP for Smart Objects) challenge in 2015. The IPSO Challenge is a global competition that focuses on innovation and entrepreneurship, inviting submissions from both academia and industry. After pitching the MicroPnP research prototype to a panel of industry leaders, our team was awarded third place. The idea of MicroPnP formed the technological foundation of VersaSense, a DistriNet spin-off founded in 2016 focusing on solutions for the Industrial IoT. VersaSense has since proven its expertise in many fields, with industrial applications such as production line monitoring, smart farming and building monitoring. This success story highlights the importance of synergy between academia and industry in this relatively young research field.

7.3 Outlook

This dissertation presented three contributions towards promoting dynamism and security in the IoT. Naturally, these contributions do not fully resolve all challenges the IoT is facing. This section looks to the road ahead and what is required to achieve widespread adoption of the technology, and how the contributions made in this thesis fit in.

Integrated configuration management The benefits of configuration management have long been recognized in mainstream computing. Industry standard tools like Chef [16] and Puppet [77] are ubiquitous for managing large-scale enterprise and cloud infrastructures, using high-level declarative languages to specify configuration. In the IoT space such solutions are currently non-existent, while at the same time more and more configuration is exposed locally to cope with dynamism. In order to support future large-scale deployments such as smart cities, integration with configuration frameworks such as Puppet and Chef is imperative. Although in their current form the way these tools inspect and configure systems is too heavy weight, contributions from this dissertation like *Refraction*, *Tomography* and *Safe reparametrization* are first steps to lower overhead and to provide a holistic configuration management framework for the IoT.

Security as a primary design objective As the IoT is moving out of the lab and into the real world, security becomes a main attention point. While most devices out in the field have some form of security, it is often severely lacking. This is illustrated by the many exploits published for IoT devices [54, 103, 84]. A general change of mentality is required when designing IoT hardware and software to make security a primary design objective, and to harden these devices for attackers. The *Security MicroVisor*, the last contribution presented in this dissertation, aids in this by providing memory protection and isolation for existing IoT devices with inherently insecure processors. While a permanent solution would involve designing systems with secure hardware from the ground up, SµV provides a reliable solution for legacy devices. Physical tamper resistance and countermeasures against hardware side-channel attacks are other research tracks that should be explored to provide full stack security in the IoT.

Interoperability and standardization Efforts have been made in recent years to standardize the IoT in several ways. Examples include radio technologies, like Zigbee [113] and LoRa [60] standardized by their respective alliances, and

application protocols like CoAP (IETF) [91] and MQTT-SN (ISO) [95]. The ultimate goal is interoperability between different hardware and software, but in practice this is often not the case due to multiple competing standards or manufacturer specific variations. A prime example of the proliferation of standards is the large amount of Zigbee-like radio protocols, such as Thread [99], SmartMeshIP [106] and recently Bluetooth Mesh [37], all providing similar features within the same frequency band. When looking at the upper layers of the software stack, application protocol standards remain limited to harmonizing data ingestion and control from a back-end perspective. As a result, currently available IoT products often only operate within their own ecosystem. Looking into the future, it is expected that market demand will drive the IoT to be more flexible and open. It is therefore essential that there is an evolution towards unified standards with a renewed focus on in-network interaction, security and uniform management interfaces.

Fully realizing these points will go a long way towards fulfilling the vision of the Internet of Things, closing the gap between the virtual and physical world. The contributions made in this dissertation are enabling technologies for achieving this goal in a safe, manageable way.

Bibliography

[1] Aberer, K., Hauswirth, M., and Salehi, A. A Middleware for Fast and Flexible Sensor Network Deployment. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)* (2006), VLDB Endowment, pp. 1199–1202.

[2] Akkermans, S., Daniels, W., Ramachandran, G. S., Crispo, B., and Hughes, D. CerberOS: A Resource-Secure OS for Sharing IoT Devices. In *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks (EWSN)* (USA, 2017), Junction Publishing, pp. 96–107.

[3] Ammar, M., Daniels, W., Crispo, B., and Hughes, D. SPEED: Secure Provable Erasure for Class-1 IoT Devices. In *ACM Conference on Data and Application Security and Privacy (CODASPY)* (mar 2018).

[4] Amsterdam. Amsterdam Smart City, `http://www.amsterdamsmartcity.com` (Accessed: 2018-04-25).

[5] Arbaugh, W., Farber, D., and Smith, J. A Secure and Reliable Bootstrap Architecture. In *Proceedings of the IEEE Symposium on Security and Privacy* (1997), IEEE Comput. Soc. Press, pp. 65–71.

[6] Atmel. AVR Raven, `http://www.atmel.com/Images/doc8117.pdf` (Accessed: 2018-03-15), 2007.

[7] Atmel. AVR ATmega 1284p 8-bit microcontroller, `http://www.atmel.com/images/doc8059.pdf` (Accessed: 2018-03-15), 2009.

[8] Balani, R., Han, C.-C., Rengaswamy, R. K., Tsigkogiannis, I., and Srivastava, M. Multi-level Software Reconfiguration for Sensor Networks. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software (EMSOFT)* (New York, New York, USA, 2006), ACM Press, p. 112.

[9] Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., and Wingers, L. The SIMON and SPECK Lightweight Block Ciphers. In *Proceedings of the 52nd Annual Design Automation Conference* (New York, New York, USA, 2015), ACM Press, pp. 1–6.

[10] Bischoff, U., and Kortuem, G. RuleCaster: A Macroprogramming System for Sensor Networks. In *OOPSLA Workshop on Building Software for Sensor Networks* (2006), pp. 1–5.

[11] Blair, G. S., Coulson, G., Robin, P., and Papathomas, M. An Architecture for Next Generation Middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing* (London, UK, UK, 1998), Springer-Verlag, pp. 191–206.

[12] Bormann, C., Ersue, M., and Keranen, A. Terminology for Constrained-Node Networks. RFC 7228, Internet Engineering Task Force, may 2014.

[13] Brasser, F., El Mahjoub, B., Sadeghi, A.-R., Wachsmann, C., and Koeberl, P. TyTAN: Tiny Trust Anchor for Tiny Devices. In *Proceedings of the 52nd Annual Design Automation Conference* (New York, New York, USA, jun 2015), ACM Press, p. 6.

[14] Brouwers, N., Langendoen, K., and Corke, P. Darjeeling, a Feature-Rich VM for the Resource Poor. In *Proceedings of the 7th International Conference on Embedded Networked Sensor Systems (SenSys)* (New York, New York, USA, 2009), ACM Press, p. 169.

[15] Castelluccia, C., Francillon, A., Perito, D., and Soriente, C. On the Difficulty of Software-based Attestation of Embedded Devices. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)* (New York, New York, USA, 2009), ACM Press, pp. 400–409.

[16] Chef. Chef Webpage, `http://www.chef.io` (Accessed: 2018-03-23).

[17] Costa, P., Coulson, G., Gold, R., Lad, M., Mascolo, C., Mottola, L., Picco, G. P., Sivaharan, T., Weerasinghe, N., and Zachariadis, S. The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario. In *IEEE International Conference on Pervasive Computing and Communications (PerCom)* (2007), IEEE, pp. 69–78.

[18] Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., and Sivaharan, T. A Generic Component Model for

Building Systems Software. *ACM Transactions on Computer Systems 26*, 1 (feb 2008), 1–42.

[19] Crowley, C., Bachiller, R., Daniels, W., Joosen, W., and Hughes, D. Participation in Context: An Exploratory Study of Querying in Participatory Applications. In *Academy of Management Proceedings* (aug 2014), Academy of Management, p. 16983.

[20] Crowley, C., Daniels, W., Bachiller, R., Joosen, W., and Hughes, D. Increasing User Participation: An Exploratory Study of Querying on the Facebook and Twitter Platforms. *First Monday 19*, 8 (jul 2014).

[21] Daniels, W., del Cid Garcia, P. J., Hughes, D., Michiels, S., Blondia, C., and Joosen, W. Composition-safe Re-parametrization in Distributed Component-based WSN Applications. In *IEEE 12th International Symposium on Network Computing and Applications (NCA)* (aug 2013), IEEE, pp. 153–156.

[22] Daniels, W., del Cid Garcia, P. J., Joosen, W., and Hughes, D. Safe Reparametrization of Component-Based WSNs. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services (MobiQuitous)* (2014), Springer International Publishing, pp. 524–536.

[23] Daniels, W., Hughes, D., Ammar, M., Crispo, B., Matthys, N., and Joosen, W. SμV - the Security MicroVisor: a Virtualisation-based Security Middleware for the Internet of Things. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track* (New York, New York, USA, 2017), ACM Press, pp. 36–42.

[24] Daniels, W., Proença, J., Clarke, D., Joosen, W., and Hughes, D. Refraction: Low-Cost Management of Reflective Meta-Data in Pervasive Component-Based Applications. In *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)* (New York, New York, USA, 2015), ACM Press, pp. 27–36.

[25] Daniels, W., Proença, J., Matthys, N., Joosen, W., and Hughes, D. Tomography: Lowering Management Overhead for Distributed Component-Based Applications. In *Proceedings of the 2nd Workshop on Middleware for Context-Aware Applications in the IoT (M4IoT)* (New York, New York, USA, 2015), ACM Press, pp. 13–18.

[26] Daniels, W., Vanbrabant, B., Hughes, D., and Joosen, W. Automated Allocation and Configuration of Dual Stack IP Networks. In

IFIP/IEEE International Symposium on Integrated Network Management (IM) (may 2013), pp. 1148–1153.

[27] Daniels, W., Yang, F., Matthys, N., Joosen, W., and Hughes, D. Demo: Enabling Plug-and-Play for the Internet of Things. In *Proceedings of the Posters and Demos Session of the 16th International Middleware Conference* (New York, New York, USA, 2015), ACM Press, pp. 1–2.

[28] Dunkels, A., Finne, N., Eriksson, J., and Voigt, T. Run-time Dynamic Linking for Reprogramming Wireless Sensor Networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys)* (New York, New York, USA, 2006), ACM Press, p. 15.

[29] Dunkels, A., Gold, R., Marti, S. A., Pears, A., and Uddenfeldt, M. Janus. In *Proceedings of the 1st ACM workshop on Dynamic Interconnection of Networks (DIN)* (New York, New York, USA, 2005), ACM Press, p. 48.

[30] Dunkels, A., Gronvall, B., and Voigt, T. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *IEEE International Conference on Local Computer Networks (LCN)* (2004), IEEE (Comput. Soc.), pp. 455–462.

[31] Eldefrawy, K., Tsudik, G., Francillon, A., and Perito, D. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *19th Annual Network and Distributed System Security Symposium (NDSS)* (2012), The Internet Society.

[32] Ericsson. Ericsson Mobility Report November 2017, `https://www.ericsson.com/assets/local/mobility-report/documents/2017/ericsson-mobility-report-november-2017.pdf` (Accessed: 2018-03-15), 2017.

[33] Filipponi, L., Vitaletti, A., Landi, G., Memeo, V., Laura, G., and Pucci, P. Smart City: An Event Driven Architecture for Monitoring Public Spaces with Heterogeneous Sensors. In *Fourth International Conference on Sensor Technologies and Applications* (jul 2010), IEEE, pp. 281–286.

[34] Francillon, A., Nguyen, Q., Rasmussen, K. B., and Tsudik, G. A Minimalist Approach to Remote Attestation. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)* (Leuven, Belgium, 2014), European Design and Automation Association, p. 6.

[35] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, 1 ed. Addison-Wesley Professional, nov 1994.

[36] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. The nesC language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, New York, USA, 2003), ACM Press, p. 1.

[37] Gomez, C., Darroudi, S. M., and Savolainen, T. IPv6 Mesh over Bluetooth Low Energy using IPSP. RFC draft-ietf-6lo-blemesh-02, Internet Engineering Task Force, sep 2017.

[38] Google. Announcing the First SHA1 Collision, `https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html` (Accessed: 2018-03-23), 2017.

[39] Grace, P., Hughes, D., Porter, B., Blair, G. S., Coulson, G., and Taiani, F. Experiences with Open Overlays. *ACM SIGOPS Operating Systems Review 42*, 4 (apr 2008), 123.

[40] Gummadi, R., Gnawali, O., and Govindan, R. Macro-programming Wireless Sensor Networks Using Kairos. *Proceedings of the 1st IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS)* (2005), 126–140.

[41] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. System Architecture Directions for Networked Sensors. *ACM SIGARCH Computer Architecture News 28*, 5 (dec 2000), 93–104.

[42] Hill, J. L., and Culler, D. E. Mica: A Wireless Platform for Deeply Embedded Networks. *IEEE Micro 22*, 6 (nov 2002), 12–24.

[43] Horré, W. *Management Solutions for Distributed Software Applications in Multi-Purpose Sensor Networks*. PhD thesis, KU Leuven, 2011.

[44] Hughes, D., Cañete, E., Daniels, W., Ramachandran, G. S., Meneghello, J., Matthys, N., Maerien, J., Michiels, S., Huygens, C., Joosen, W., Wijnants, M., Lamotte, W., Hulsmans, E., Lannoo, B., and Moerman, I. Energy Aware Software Evolution for Wireless Sensor Networks. In *IEEE 14th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)* (jun 2013), IEEE, pp. 1–9.

[45] Hughes, D., Crowley, C., Daniels, W., Bachiller, R., and Joosen, W. User-Rank: Generic Query Optimization for Participatory

Social Applications. In *47th Hawaii International Conference on System Sciences* (jan 2014), IEEE, pp. 1874–1883.

[46] Hughes, D., Greenwood, P., Blair, G., Coulson, G., Grace, P., Pappenberger, F., Smith, P., and Beven, K. An Experiment with Reflective Middleware to Support Grid-based Flood Monitoring. *Concurrency and Computation: Practice and Experience 20*, 11 (aug 2008), 1303–1316.

[47] 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 *IEEE 11th International Symposium on Network Computing and Applications (NCA)* (aug 2012), IEEE, pp. 236–243.

[48] Hui, J. W., and Culler, D. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys)* (New York, New York, USA, 2004), ACM Press, p. 81.

[49] Huygens, C., Hughes, D., Lagaisse, B., and Joosen, W. Streamlining Development for Networked Embedded Systems Using Multiple Paradigms. *IEEE Software 27*, 5 (sep 2010), 45–52.

[50] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Proceedings of the 3rd International Conference on NASA Formal Methods* (Berlin, Heidelberg, 2011), Springer-Verlag, pp. 41–55.

[51] Jongboom, J., and Stokking, J. Enabling Firmware Updates over LPWANs. Tech. rep., ARM & The Things Industries, 2018.

[52] Koeberl, P., Schulz, S., Sadeghi, A.-R., and Varadharajan, V. TrustLite: A Security Architecture for Tiny Embedded Devices. In *Proceedings of the 9th European Conference on Computer Systems* (New York, NY, USA, 2014), ACM, p. 14.

[53] Krishnamurthy, L., Adler, R., Buonadonna, P., Chhabra, J., Flanigan, M., Kushalnagar, N., Nachman, L., and Yarvis, M. Design and Deployment of Industrial Sensor Networks. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys)* (New York, New York, USA, 2005), ACM Press, p. 64.

[54] Langner, R. Stuxnet: Dissecting a Cyberwarfare Weapon. *IEEE Security & Privacy Magazine 9*, 3 (may 2011), 49–51.

[55] Lau, S.-y., Chang, T.-h., Hu, S.-y., Huang, H.-j., Shyu, L.-d., Chiu, C.-m., and Huang, P. Sensor Networks for Everyday Use: The BL-Live Experience. In *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC)* (2006), vol. 1, IEEE, pp. 336–343.

[56] Li, S., Lin, Y., Son, S. H., Stankovic, J. A., and Wei, Y. Event Detection Services Using Data Service Middleware in Distributed Sensor Networks. *Telecommunication Systems 26*, 2-4 (jun 2004), 351–368.

[57] Li, Y., McCune, J. M., and Perrig, A. SBAP: Software-Based Attestation for Peripherals. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing* (Berlin, Heidelberg, 2010), Springer-Verlag, pp. 16–29.

[58] Li, Y., McCune, J. M., and Perrig, A. VIPER: Verifying the Integrity of Peripherals' Firmware. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)* (New York, New York, USA, 2011), ACM Press, pp. 3–16.

[59] Logos. Zigduino, `http://www.logos-electro.com/zigduino` (Accessed: 2018-03-23), 2013.

[60] LoRa Alliance. LoRa Webpage, `https://lora-alliance.org` (Accessed: 2018-04-25).

[61] Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. TinyDB: an Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems 30*, 1 (mar 2005), 122–173.

[62] 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* (feb 2015), 141–169.

[63] 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 (M4IOT)* (New York, New York, USA, 2014), ACM Press, pp. 7–12.

[64] Matthys, N. *Software Technologies for Dynamic Sensor Networks*. PhD thesis, KU Leuven, 2014.

[65] Matthys, N., Yang, F., Daniels, W., Joosen, W., and Hughes, D. Demonstration of MicroPnP: The Zero-Configuration Wireless Sensing and Actuation Platform. In *13th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)* (jun 2016), IEEE, pp. 1–2.

[66] Matthys, N., Yang, F., Daniels, W., Michiels, S., Joosen, W., Hughes, D., and Watteyne, T. μPnP-Mesh: The Plug-and-Play Mesh Network for the Internet of Things. In *IEEE 2nd World Forum on Internet of Things (WF-IoT)* (dec 2015), IEEE, pp. 311–315.

[67] Morales Ruiz, A., Daniels, W., Hughes, D., and Grothoff, C. Cryogenic: Enabling Power-Aware Applications on Linux. In *Proceedings of the 2nd International Conference on ICT for Sustainability (ICT4S)* (Paris, France, 2014), Atlantis Press.

[68] Mottola, L., and Picco, G. P. Middleware for Wireless Sensor Networks: an Outlook. *Journal of Internet Services and Applications 3*, 1 (nov 2011), 31–39.

[69] Mottola, L., and Picco, G. P. Programming Wireless Sensor Networks. *ACM Computing Surveys 43*, 3 (apr 2011), 1–51.

[70] Nest Labs. Nest Labs Introduces World's First Learning Thermostat, `https://nest.com/press/nest-labs-introduces-worlds-first-learning-thermostat` (Accessed: 2018-03-15), 2011.

[71] Nest Labs. New Nest Cam IQ Indoor Security Camera Knows What to Look for in the Home, Tells You What's Important, `https://nest.com/press/new-nest-cam-iq-indoor-security-camera-knows-what-to-look-for-in-the-home-tells-you-whats-important` (Accessed: 2018-03-15), 2017.

[72] 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 *Proceedings of the 22nd USENIX Conference on Security* (Berkeley, CA, USA, 2013), USENIX Association, pp. 479–494.

[73] OSGi Alliance. The Dynamic Module System for Java, `https://www.osgi.org` (Accessed: 2018-04-28).

[74] Parno, B., McCune, J. M., and Perrig, A. Bootstrapping Trust in Commodity Computers. In *Proceedings of the IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), IEEE, pp. 414–429.

[75] Perito, D., and Tsudik, G. Secure Code Update for Embedded Devices via Proofs of Secure Erasure. In *European Symposium on Research in Computer Security (ESORICS)* (2010), pp. 643–662.

[76] Philips. Introducing Philips Hue: the World's Smartest LED Bulb, Marking a New Era in Home Lighting, `http://www.newsroom.lighting.philips.com/news/2012/20121029-Introducing-Philips-hue` (Accessed: 2018-03-15), 2012.

[77] Puppet. Puppet Webpage, `http://www.puppet.com` (Accessed: 2018-03-23).

[78] Ramachandran, G. S., Daniels, W., Matthys, N., Huygens, C., Michiels, S., Joosen, W., Meneghello, J., Lee, K., Cañete, E., Díaz Rodríguez, M., and Hughes, D. Measuring and Modeling the Energy Cost of Reconfiguration in Sensor Networks. *IEEE Sensors Journal 15*, 6 (jun 2015), 3381–3389.

[79] Ramachandran, G. S., Daniels, W., Proença, J., Michiels, S., Joosen, W., Hughes, D., and Porter, B. Hitch Hiker: A Remote Binding Model with Priority Based Data Aggregation for Wireless Sensor Networks. In *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)* (New York, New York, USA, 2015), ACM Press, pp. 43–48.

[80] Ramachandran, G. S., Matthys, N., Daniels, W., Joosen, W., and Hughes, D. Building Dynamic and Dependable Component-Based Internet-of-Things Applications with Dawn. In *Proceedings of the 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)* (apr 2016), IEEE, pp. 97–106.

[81] Ramachandran, G. S., Proença, J., Daniels, W., Pickavet, M., Staessens, D., Huygens, C., Joosen, W., and Hughes, D. Hitch Hiker 2.0: a Binding Model with Flexible Data Aggregation for the Internet-of-Things. *Journal of Internet Services and Applications 7*, 1 (dec 2016), 4.

[82] Rellermeyer, J. S., and Alonso, G. Concierge: A Service Platform for Resource-Constrained Devices. *ACM SIGOPS Operating Systems Review 41*, 3 (jun 2007), 245.

[83] Rezgui, A., and Eltoweissy, M. Service-oriented Sensor–Actuator Networks: Promises, Challenges, and the Road Ahead. *Computer Communications 30*, 13 (sep 2007), 2627–2648.

[84] Ronen, E., and Shamir, A. Extended Functionality Attacks on IoT Devices: The Case of Smart Lights. In *IEEE European Symposium on Security and Privacy (EuroS&P)* (mar 2016), IEEE, pp. 3–12.

[85] Rooney, S., Bauer, D., and Scotton, P. Edge Server Software Architecture for Sensor Applications. In *Proceedings of the 2005 Symposium on Applications and the Internet (SAINT)* (Trento, Italy, 2005), IEEE, pp. 64–71.

[86] Sanchez, L., Muñoz, L., Galache, J. A., Sotres, P., Santana, J. R., Gutierrez, V., Ramdhany, R., Gluhak, A., Krco, S., Theodoridis, E., and Pfisterer, D. SmartSantander: IoT Experimentation over a Smart City Testbed. *Computer Networks 61* (mar 2014), 217–238.

[87] Seshadri, A., Luk, M., and Perrig, A. SAKE: Software Attestation for Key Establishment in Sensor Networks. In *Distributed Computing in Sensor Systems* (Berlin, Heidelberg, 2008), Springer Berlin Heidelberg, pp. 372–385.

[88] Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., and Khosla, P. Pioneer: : Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. *ACM SIGOPS Operating Systems Review 39*, 5 (oct 2005).

[89] Seshadri, A., Perrig, A., van Doorn, L., and Khosla, P. SWATT: Software-based Attestation for Embedded Devices. In *Proceedings of the IEEE Symposium on Security and Privacy* (may 2004), IEEE, pp. 272–282.

[90] Shankar, U., Chew, M., and Tygar, J. D. Side Effects Are Not Sufficient to Authenticate Software. In *Proceedings of the 13th USENIX Conference on Security* (Berkeley, CA, USA, 2004), vol. 13, USENIX Association, pp. 89–101.

[91] Shelby, Z., Hartke, K., and Bormann, C. The Constrained Application Protocol (CoAP). RFC 7252, Internet Engineering Task Force, jun 2014.

[92] Shneidman, J., Pietzuch, P., Ledlie, J., Roussopoulos, M., Seltzer, M., and Welsh, M. Hourglass: An Infrastructure for Connecting Sensor Networks and Applications. Tech. rep., Harvard, 2004.

[93] Simon, D., Cifuentes, C., Cleal, D., Daniels, J., and White, D. Java™ on the Bare Metal of Wireless Sensor Devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE)* (New York, New York, USA, 2006), ACM Press, pp. 78–88.

[94] Smith, B. C. *Procedural Reflection in Programming Languages.* PhD thesis, Massachusetts Institute of Technology, 1982.

[95] Stanford-Clark, A., and Truong, H. L. MQTT For Sensor Networks (MQTT-SN) - Protocol Specification 1.2, http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf (Accessed: 2018-04-25), 2013.

[96] Taherkordi, A., Loiret, F., Abdolrazaghi, A., Rouvoy, R., Le-Trung, Q., and Eliassen, F. Programming Sensor Networks using REMORA Component Model. In *Proceedings of the 6th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS)* (2010), pp. 45–62.

[97] Texas Instruments. Secure In-Field Firmware Updates for MSP MCUs, http://www.ti.com/lit/an/slaa682/slaa682.pdf (Accessed: 2018-04-28), 2015.

[98] Thoelen, K. *An Application Platform for Multi-purpose Sensor Systems.* PhD thesis, KU Leuven, 2016.

[99] Thread Group. Thread Webpage, https://www.threadgroup.org (Accessed: 2018-04-25).

[100] Tolle, G., and Culler, D. Design of an Application-Cooperative Management System for Wireless Sensor Networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN)* (Istanbul, Turkey, 2005), IEEE, pp. 121–132.

[101] Trusted Computing Group. TPM Main Specification Level 2 Version 1.2, http://www.trustedcomputinggroup.org/tpm-main-specification (Accessed: 2018-03-15), 2011.

[102] United States Department of Defense. Trusted Computer System Evaluation Criteria (Orange Book). Tech. rep., 1985.

[103] US Computer Emergency Readiness Team. Heightened DDoS Threat Posed by Mirai and Other Botnets - alert TA16-288A, http://www.us-cert.gov/ncas/alerts/TA16-288A (Accessed: 2018-03-23), 2016.

[104] Vinoski, S. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine 35*, 2 (1997), 46–55.

[105] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient Software-based Fault Isolation. *ACM SIGOPS Operating Systems Review 27*, 5 (dec 1993), 203–216.

[106] Watteyne, T., Doherty, L., Simon, J., and Pister, K. Technical Overview of SmartMesh IP. In *Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing* (jul 2013), IEEE, pp. 547–551.

[107] Watteyne, T., Palattella, M. R., and Grieco, L. A. Using IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) in the Internet of Things (IoT): Problem Statement. RFC 7554, Internet Engineering Task Force, may 2015.

[108] Welsh, M., and Mainland, G. Programming Sensor Networks Using Abstract Regions. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation (NSDI)* (Berkeley, CA, USA, 2004), USENIX Association, pp. 3–17.

[109] Werner-Allen, G., Lorincz, K., Ruiz, M., Marcillo, O., Johnson, J., Lees, J., and Welsh, M. Deploying a Wireless Sensor Network on an Active Volcano. *IEEE Internet Computing 10*, 2 (mar 2006), 18–25.

[110] Whitehouse, K., Sharp, C., Brewer, E., and Culler, D. Hood. In *Proceedings of the 2nd International Conference on Mobile systems, Applications, and Services (MobiSYS)* (New York, New York, USA, 2004), ACM Press, p. 99.

[111] Yang, F., Matthys, N., Bachiller, R., Michiels, S., Joosen, W., and Hughes, D. μPnP: Plug and Play Peripherals for the Internet of Things. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys)* (New York, New York, USA, 2015), ACM Press, pp. 1–14.

[112] Yao, Y., and Gehrke, J. The Cougar Approach to In-Network Query Processing in Sensor Networks. *ACM SIGMOD Record 31*, 3 (sep 2002), 9.

[113] Zigbee Alliance. Zigbee Webpage, `http://www.zigbee.org` (Accessed: 2018-04-25).

List of publications

Articles in international peer reviewed journals

Ramachandran, G. S., Proença, J., Daniels, W., Pickavet, M., Staessens, D., Huygens, C., Joosen, W., and Hughes, D. Hitch Hiker 2.0: a Binding Model with Flexible Data Aggregation for the Internet-of-Things. *Journal of Internet Services and Applications 7*, 1 (dec 2016), 4

Ramachandran, G. S., Daniels, W., Matthys, N., Huygens, C., Michiels, S., Joosen, W., Meneghello, J., Lee, K., Cañete, E., Díaz Rodríguez, M., and Hughes, D. Measuring and Modeling the Energy Cost of Reconfiguration in Sensor Networks. *IEEE Sensors Journal 15*, 6 (jun 2015), 3381–3389

Crowley, C., Daniels, W., Bachiller, R., Joosen, W., and Hughes, D. Increasing User Participation: An Exploratory Study of Querying on the Facebook and Twitter Platforms. *First Monday 19*, 8 (jul 2014)

Papers at international conferences and symposia, published in full in proceedings

Ammar, M., Daniels, W., Crispo, B., and Hughes, D. SPEED: Secure Provable Erasure for Class-1 IoT Devices. In *ACM Conference on Data and Application Security and Privacy (CODASPY)* (mar 2018)

Akkermans, S., Daniels, W., Ramachandran, G. S., Crispo, B., and Hughes, D. CerberOS: A Resource-Secure OS for Sharing IoT Devices. In *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks (EWSN)* (USA, 2017), Junction Publishing, pp. 96–107

Daniels, W., Hughes, D., Ammar, M., Crispo, B., Matthys, N., and Joosen, W. SμV - the Security MicroVisor: a Virtualisation-based Security Middleware for the Internet of Things. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track* (New York, New York, USA, 2017), ACM Press, pp. 36–42

Matthys, N., Yang, F., Daniels, W., Joosen, W., and Hughes, D. Demonstration of MicroPnP: The Zero-Configuration Wireless Sensing and Actuation Platform. In *13th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)* (jun 2016), IEEE, pp. 1–2

Ramachandran, G. S., Matthys, N., Daniels, W., Joosen, W., and Hughes, D. Building Dynamic and Dependable Component-Based Internet-of-Things Applications with Dawn. In *Proceedings of the 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)* (apr 2016), IEEE, pp. 97–106

Daniels, W., Proença, J., Clarke, D., Joosen, W., and Hughes, D. Refraction: Low-Cost Management of Reflective Meta-Data in Pervasive Component-Based Applications. In *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)* (New York, New York, USA, 2015), ACM Press, pp. 27–36

Daniels, W., Proença, J., Matthys, N., Joosen, W., and Hughes, D. Tomography: Lowering Management Overhead for Distributed Component-Based Applications. In *Proceedings of the 2nd Workshop on Middleware for Context-Aware Applications in the IoT (M4IoT)* (New York, New York, USA, 2015), ACM Press, pp. 13–18

Daniels, W., Yang, F., Matthys, N., Joosen, W., and Hughes, D. Demo: Enabling Plug-and-Play for the Internet of Things. In *Proceedings of the Posters and Demos Session of the 16th International Middleware Conference* (New York, New York, USA, 2015), ACM Press, pp. 1–2

Matthys, N., Yang, F., Daniels, W., Michiels, S., Joosen, W., Hughes, D., and Watteyne, T. μPnP-Mesh: The Plug-and-Play Mesh Network for the Internet of Things. In *IEEE 2nd World Forum on Internet of Things (WF-IoT)* (dec 2015), IEEE, pp. 311–315

Ramachandran, G. S., Daniels, W., Proença, J., Michiels, S., Joosen, W., Hughes, D., and Porter, B. Hitch Hiker: A Remote Binding Model with Priority Based Data Aggregation for Wireless Sensor Networks. In *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)* (New York, New York, USA, 2015), ACM Press, pp. 43–48

Crowley, C., Bachiller, R., Daniels, W., Joosen, W., and Hughes, D. Participation in Context: An Exploratory Study of Querying in Participatory Applications. In *Academy of Management Proceedings* (aug 2014), Academy of Management, p. 16983

Daniels, W., del Cid Garcia, P. J., Joosen, W., and Hughes, D. Safe Reparametrization of Component-Based WSNs. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services (MobiQuitous)* (2014), Springer International Publishing, pp. 524–536

Hughes, D., Crowley, C., Daniels, W., Bachiller, R., and Joosen, W. User-Rank: Generic Query Optimization for Participatory Social Applications. In *47th Hawaii International Conference on System Sciences* (jan 2014), IEEE, pp. 1874–1883

Morales Ruiz, A., Daniels, W., Hughes, D., and Grothoff, C. Cryogenic: Enabling Power-Aware Applications on Linux. In *Proceedings of the 2nd International Conference on ICT for Sustainability (ICT4S)* (Paris, France, 2014), Atlantis Press

Daniels, W., del Cid Garcia, P. J., Hughes, D., Michiels, S., Blondia, C., and Joosen, W. Composition-safe Re-parametrization in Distributed Component-based WSN Applications. In *IEEE 12th International Symposium on Network Computing and Applications (NCA)* (aug 2013), IEEE, pp. 153–156

Daniels, W., Vanbrabant, B., Hughes, D., and Joosen, W. Automated Allocation and Configuration of Dual Stack IP Networks. In *IFIP/IEEE International Symposium on Integrated Network Management (IM)* (may 2013), pp. 1148–1153

Hughes, D., Cañete, E., Daniels, W., Ramachandran, G. S., Meneghello, J., Matthys, N., Maerien, J., Michiels, S., Huygens, C., Joosen, W., Wijnants, M., Lamotte, W., Hulsmans, E., Lannoo, B., and Moerman, I. Energy Aware Software Evolution for Wireless Sensor Networks. In *IEEE 14th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)* (jun 2013), IEEE, pp. 1–9

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 *Proceedings of the 22nd USENIX Conference on Security* (Berkeley, CA, USA, 2013), USENIX Association, pp. 479–494

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMEC-DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Leuven
wilfried.daniels@cs.kuleuven.be
http://distrinet.cs.kuleuven.be