KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Sancus: A Low-Cost Security Architecture for Distributed IoT Applications on a Shared Infrastructure

Job Noorman

Supervisors:
Prof. dr. ir. F. Piessens
Prof. dr. B. Jacobs

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

April 2017

Sancus: A Low-Cost Security Architecture for Distributed IoT Applications on a Shared Infrastructure

Job NOORMAN

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

April 2017

Preface

This thesis describes the results of over four years of hard work, enjoyment, and frustration. Although I can now look back at this time as one of the most enjoyable and educational periods of my life, more than once, the frustration gained the upper hand. I would like to start this text by thanking the people without whom I might not have come this far.

First of all, I would like to thank my supervisor, Frank Piessens, because I could not have wished for a better person to guide me through my PhD. Not only were the discussions we had interesting, instructive, and even fun, he *always* seemed to be available for them. Frank, I will genuinely miss working with you.

Many thanks also go out to the members of my jury: Omer Van der Biest, Bart Jacobs, Ingrid Verbauwhede, Christophe Huygens, Deepak Garg, and Bjorn De Sutter. The discussions we had, feedback you gave, and tough questions you asked during the preliminary defense greatly improved the quality of this text.

Research is obviously not something you do on your own. I would like to thank all the people that contributed to Sancus over the years through the exchange of ideas, although doing so by name would unfortunately be an impossible task. I *do* want to explicitly thank Jan Tobias Mühlberg and Pieter Maene for their active contributions to Sancus and Jo Van Bulck for continuing the Sancus project after me.

Luckily, I did not only work during my PhD and I would like to thank all my colleagues, office mates, and friends at the computer science department for the good moments at the coffee machine, at lunch, during social events,... I especially want to thank two of my closest friends – Gijs Vanspauwen and Joachim Jansen – for all the fun times we had. Having you guys as colleagues during the past years made the hard times a bit easier to overcome.

Last but not least, I want to thank my family – my parents, my brothers, and all my in-laws – for always being supportive and listening to my problems. But most importantly, I want to thank my wife, Jessica, for *always* being there when I most needed it. At times, living with me must have been a challenge but

you did it with more love than I could have wished for. I am looking forward to facing the next challenge with you!

Thank you all!

– Job Noorman

This work has been partially supported by a doctoral grant of the Research Foundation - Flanders (FWO), by Intel Lab's University Research Office, and by the Research Fund KU Leuven.

Abstract

With the rising popularity of the Internet of Things (IoT), the use of small, low-power embedded devices is rapidly increasing. Unfortunately, these kind of devices often lack the security features we are grown used to in the domain of desktop and server computing. However, in a context where multiple mutually distrusting stakeholders are able to share an IoT infrastructure to process sensitive data, the lack of, for example, basic software isolation is becoming increasingly irresponsible. Finding secure yet inexpensive ways to protect those low-end devices is therefore becoming more and more critical.

The first part of this thesis proposes Sancus, an inexpensive security architecture for resource-constrained IoT devices. We start with accurately defining our context; the kind of systems we want to protect and the attacker model we will use. Then, we introduce Sancus' design in enough detail for interested parties to be able to create alternative implementations. Next, our own implementation, based on the TI MSP430 architecture, is described and evaluated in terms of hardware cost and software overhead. We conclude this part by giving an overview of related work and a comparison of Sancus with the most relevant alternative architectures.

In the second part, we discuss some applications of the Sancus architecture. The first application shows how to use a small number of protected Sancus modules to attest the state of a large unprotected software base. This can be used when adapting the whole software base to make use of Sancus' features is for some reason infeasible. We then show, in our second application, how Sancus can be used to provide security guarantees for distributed applications that use I/O devices. We provide a deployment and attestation technique that gives high assurance that if a distributed application produces an output, there must have been a sequence of physical input events that, when processed by the application as specified in its source code, produces the observed output event.

We conclude this thesis with a discussion of some of the design decisions of Sancus and ways to improve the architecture. We show how to improve the secure communication primitive, how to employ public-key cryptography, and how to overcome some of the inflexibilities in Sancus' design.

Samenvatting

Met de opkomende populariteit van het Internet der Dingen (*the Internet of Things*), stijgt het gebruik van kleine, laag vermogen ingebedde apparaten exponentieel. Helaas missen dit soort apparaten vaak de beveiligingsfuncties die we gewend zijn bij desktop- en serversystemen. Echter, in een context waar verschillende onderling wantrouwende partijen een infrastructuur delen om gevoelige data te verwerken, wordt het gebrek aan, bijvoorbeeld, software isolatie in toenemende mate onverantwoord. Het wordt daarom steeds kritieker om veilige doch goedkope manieren te vinden om zulke low-end apparaten te beveiligen.

In het eerste deel van deze thesis stellen we Sancus voor, een goedkope beveiligingsarchitectuur voor apparaten die slechts beperkte middelen ter beschikking hebben. We definiëren eerst exact wat onze context is; het type systemen dat we willen beveiligen en het aanvalsmodel dat we zullen gebruiken. Dan presenteren we het ontwerp van Sancus in genoeg detail zodat geïnteresseerden alternatieve implementaties kunnen maken. Vervolgens wordt onze implementatie, die gebaseerd is op de TI MSP430 architectuur, beschreven en geëvalueerd in functie van de hardwareprijs en softwareoverhead. We besluiten dit deel met een overzicht van gerelateerd werk en een vergelijking van Sancus met de meest relevante alternatieve architecturen.

Het tweede deel bediscussieert een aantal toepassingen van de Sancus architectuur. Een eerste toepassing toont hoe een klein aantal beschermde Sancus modules gebruikt kan worden om de staat van een groot onbeveiligd softwaresysteem te attesteren. Dit is nuttig wanneer het om bepaalde redenen onmogelijk is om het volledige softwaresysteem gebruik te laten maken van de beveiligingsfuncties van Sancus. In een tweede toepassing laten we daarna zien hoe Sancus beveiligingsgaranties kan bieden aan gedistribueerde applicaties die gebruik maken van I/O apparaten. We ontwikkelen een installatie- en attestatietechniek die een grote mate van zekerheid geeft dat wanneer een applicatie een output produceert, er een sequentie van fysische inputs moet geweest zijn die, wanneer deze verwerkt wordt door de applicatie zoals gespecificeerd in de broncode, de geobserveerde output voortbrengt.

We besluiten deze thesis met een bespreking van bepaalde ontwerpbeslissingen van Sancus en manieren om de architectuur te verbeteren. We laten zien hoe veilige communicatie met Sancus vereenvoudigd kan worden, hoe asymmetrische cryptografie gebruikt kan worden, en hoe bepaalde gevallen van starheid van het ontwerp van Sancus overwonnen kunnen worden.

Contents

Abstract iii

Samenvatting v

Contents vii

List of Figures xi

List of Tables xiii

Acronyms xv

1 Introduction 1

I Core Architecture 5

2 Problem Statement 7
- 2.1 System Model 7
- 2.2 Attacker Model 9
- 2.3 Security Properties 9

3 Design of Sancus 11
- 3.1 Cryptographic Primitives 11
- 3.2 Overview 12
  - 3.2.1 Nodes 12
  - 3.2.2 Software Providers 12
  - 3.2.3 Software Modules 13
  - 3.2.4 Memory Protection on the Nodes 15
  - 3.2.5 Remote Attestation and Secure Communication 15
  - 3.2.6 Secure Linking 15
- 3.3 Key Management 16

3.4 Memory Access Control 17
3.5 Remote Attestation and Secure Communication 19
3.6 Confidential Loading 20
3.7 Secure Linking and Local Communication 21
3.8 An End-to-End Example 22

4 Implementation 25
4.1 The Processor 25
4.1.1 Isolation 25
4.1.2 Cryptography 27
4.1.3 Core Modifications 28
4.2 The Compiler 29
4.2.1 Entry Points 29
4.2.2 Stack Switching 30
4.2.3 Exiting Modules 31
4.2.4 Secure Linking 31
4.3 Deployment 31

5 Evaluation 33
5.1 Performance 33
5.1.1 Critical Path 33
5.1.2 Microbenchmarks 35
5.1.3 Macrobenchmark 36
5.2 Area 37
5.3 Security 37

6 Related Work 41
6.1 Isolation in High-End Devices 41
6.2 Isolation in Low-End Devices 42
6.3 Capability-Based Addressing 43
6.4 Secure Compilation 44

II Applications 47

7 Authentic Execution of Distributed Event-Driven Applications 49
7.1 Running Example, Infrastructure & Objectives 50
7.1.1 The Shared Infrastructure 51
7.1.2 Modules & Applications 53
7.1.3 Security Objective 53
7.2 Design of Authentic Execution 54
7.2.1 Underlying Architecture 54
7.2.2 Mapping source modules to Protected Modules (PMs) . 55

7.2.3 Untrusted Software on the Nodes 57
7.2.4 Physical Input and Output Channels 59
7.2.5 Deployment 59
7.2.6 Security Argument 60
7.3 Implementation 61
7.3.1 Secure I/O on Sancus 61
7.3.2 Compiler and Untrusted Runtime 64
7.4 Evaluation 65
7.5 Discussion 66
7.5.1 Integrity versus Confidentiality 66
7.5.2 Metaprogramming 68
7.5.3 Hardware Attacks and Side-Channels 68
7.5.4 Towards a Formal Security Proof 69

8 Trust Assessment Modules for the Internet of Things 71
8.1 Trust Assessment Modules 71
8.1.1 Module Deployment 72
8.1.2 Trust Indicators 72
8.1.3 Fault Recovery 75
8.2 Evaluation 75
8.2.1 Scenario & Implementation 75
8.2.2 Overheads 76
8.2.3 Security Evaluation 79
8.3 Related Work 81
8.3.1 Trust Management in Wireless Sensor Networks 81
8.3.2 Trust Assessment on Desktop & Server Systems 82
8.4 Conclusion 83

III Conclusion 85

9 Discussion on some Design Decisions and Future Work 87
9.1 Software Provider Keys 87
9.1.1 Problems with the Current Design 88
9.1.2 Multi-Vendor Protected Modules 88
9.1.3 Module Identity and its Main Vendor 89
9.1.4 Remote Attestation Procedure 90
9.1.5 Secure Linking Revisited 90
9.2 Public-Key Cryptography for Sancus 91
9.2.1 Public-Key Protocol for Sancus 91
9.2.2 Advantages of a Public-Key Protocol 92
9.3 Rigid Access Control Rules 93
9.3.1 A Many-to-Many Mapping Between Code and Data . . 94

9.3.2 Sharing Data Using Capabilities 95
9.4 Hardware Limit on the Number of Modules 95
9.4.1 Moving Metadata to Memory 96
9.4.2 Tagging Memory with Ownership Information 96
9.4.3 Reverse Sandboxing Using Capabilities 98

10 Conclusion 101

Bibliography 103

List of Figures

2.1 System model 8

3.1 Sancus node 14
3.2 Example setup 23

4.1 Memory Access Logic circuit 26
4.2 Hardware blocks of the Sancus core 30

5.1 Maximum CPU frequency 34
5.2 Execution time of the `encrypt` instruction 35
5.3 Execution time of the macro benchmark 36
5.4 FPGA area of Sancus 38
5.5 ASIC area of Sancus 38

7.1 Example of a distributed event-driven application 51
7.2 Source code of the running example 52
7.3 Pseudocode of the `SetKey` entry point 56
7.4 Pseudocode of the `HandleInput` entry point 57
7.5 Pseudocode of the generated output wrapper 57
7.6 Memory layout of a compiled module 58
7.7 C implementation of an example module 64
7.8 Benchmark timings of the example application 67

8.1 Deployment process of a trust assessment module 73

List of Tables

3.1 Keys used in Sancus 14
3.2 Memory access control rules enforced by Sancus 18

7.1 Code size of the evaluation scenario 65

8.1 Trust assessment module API 76
8.2 Size and execution time of the trust assessment components . . 77

Acronyms

API	Application Programming Interface
CA	Certificate Authority
ECC	Elliptic Curve Cryptography
ELF	Executable and Linkable Format
HAN	Home Area Network
IoT	Internet of Things
IP	Infrastructure Provider
LOC	Lines Of Code
MAC	Message Authentication Code
MAL	Memory Access Logic
MMIO	Memory-Mapped I/O
MMU	Memory Management Unit
OS	Operating System
PCBAC	Program Counter Based Access Control
PKI	Public Key Infrastructure
PM	Protected Module
PMA	Protected Module Architecture
ROP	Return-Oriented Programming
SGX	Software Guard Extensions

SP	Software Provider
TCB	Trusted Computing Base
TPM	Trusted Platform Module
VM	Virtual Machine
WAN	Wide Area Network
WSN	Wireless Sensor Network

Chapter 1

Introduction

Computing devices and software are omnipresent in our society, and society increasingly relies on the correct and secure functioning of these devices and software. Two important trends can be observed. First, network connectivity of devices keeps increasing. More and more (and smaller and smaller) devices get connected to the Internet or local ad-hoc networks. Many consumer products contain embedded technology to have Internet connectivity. This Internet of Things (IoT) is estimated to grow to an astonishing number of 26 billion units by 2020 [38]. Second, more and more devices support extensibility of the software they run – often even by third parties different from the device manufacturer or device owner. The IoT becomes an *infrastructure* on which many stakeholders can install and run software applications. These two factors are important because they enable a vast array of interesting applications, ranging from over-the-air updates on smart cards, over updateable implanted medical devices, to programmable sensor networks or smart home applications. However, these two factors also have a significant impact on security threats. The combination of connectivity and software extensibility leads to malware threats. Researchers have already shown how to perform code injection attacks against embedded devices to build self-propagating worms [39, 35]. Viega and Thompson [88] describe several recent incidents and summarize the state of embedded device security as “a mess”.

An important research question in this context is what infrastructural support should be provided to make it easier for device manufacturers to construct secure networked and extensible devices. Of course, this is by no means a new question and we have a rich body of experience to build on from securing classical computing devices like servers and desktops. One of the challenges addressed in this thesis is learning from this experience to create a security architecture for devices that do not have the hardware features – nor the resources to implement them – commonly found in these classical devices.

A first important and widely deployed technique used on classical devices is hardware support for virtual memory and processor privilege levels. An operating systems can use these features to run processes in isolation from each other, guard interaction between processes, and guard itself from malicious processes. It should be noted, however, that virtual memory was not designed for security but rather for usability; i.e., to make it easier to implement multiprocessing. This can be seen, for example, in the poor granularity virtual memory systems usually provide for isolation. A second technique is the use of a memory-safe Virtual Machine (VM) where modules are deployed in memory-safe bytecode and the VM implementation guards interactions between them.

While these techniques are valuable and definitely help to strengthen the security of computing devices, they require the presence of a sizable Trusted Computing Base (TCB); respectively, the Operating System (OS) or the VM implementation. Unfortunately, history has shown that it is notoriously difficult to protect such large software layers from being exploited by software attacks such as buffer overflows and code injection attacks [90]. Moreover, in the context of an infrastructure that may be shared between mutually distrusting parties, it becomes increasingly important to be able to attest the correct state of the software that has been deployed on the infrastructure. However, the larger the size of the TCB, the smaller the trustworthiness of the attestation results provided by it, making it difficult to securely implement attestation using the classic solutions.

As a consequence, researchers have started to investigate alternatives to reduce the size of the TCB. One important line of research is developing Protected Module Architectures (PMAs) [66, 65, 83]. These are security architectures running independently from classical OSs and can execute security sensitive code in an isolated area of the system. Since the isolation does not rely on the OS, the security guarantees that can be offered are significantly increased. That PMAs are a promising avenue for security research can be seen from the introduction by Intel of Software Guard Extensions (SGX), a hardware-based PMA, in their 6th Generation Intel Core processor platform. Alas, the current generation of PMAs rely on expensive hardware features, like virtualization, making them not directly applicable to resource-constrained devices.

To return to the embedded world, no effective low-cost solutions are known for low-end, resource-constrained devices. Many embedded platforms lack standard security features present in high-end processors, such as the ones discussed in the previous paragraphs. Depending on the overall system security goals, as well as the context in which the system must operate, there may be more optimal solutions than just porting the general-purpose security features from high-end processors.

Over the past few years, researchers have been exploring alternative security architectures for low-end networked devices. For instance, Eldefrawy et al. [29]

propose SMART, a simple and efficient hardware-software primitive to establish a dynamic root of trust in an embedded processor, and Strackx, Piessens, and Preneel [84] propose a simple Program Counter Based Access Control (PCBAC) system to isolate software components. For a more complete overview of this line of work, the reader is referred to Chapter 6.

The key contribution of this thesis is the design, implementation and evaluation of one such security architecture: Sancus. Sancus was first proposed in 2013 at the USENIX Security conference [72] as a security architecture that supports secure third-party software extensibility for a network of low-end processors with a hardware-only TCB. Over the past three years, significant experience has been gained with applications of Sancus, including for instance the development of a trust assessment infrastructure that uses Sancus to protect the trust measurement code [69], and the design of a smart meter secured by Sancus [68]. Also, researchers have been investigating several extensions of Sancus, for instance to support more flexible resource sharing [87] or to support confidential loading of code [42]. Informed by these additional research results, this thesis describes an improved design and implementation, supporting additional security guarantees, such as confidential deployment and a more efficient cryptographic core. More specifically, I make the following contributions:

- I propose Sancus[1], a security architecture for resource-constrained, extensible networked embedded systems, that can provide strong isolation guarantees, remote attestation, secure communication, secure linking, and confidential software deployment with a minimal (hardware) TCB.
- I implement the hardware required for Sancus as an extension of a mainstream microprocessor architecture, the TI MSP430.
- I implement a C compiler that targets Sancus-enabled devices. Building software modules for Sancus can be done using simple annotations on standard C code.
- I implement a Contiki-based (untrusted) software stack to automate the deployment process of Sancus modules.
- I evaluate Sancus in terms of security, hardware cost, and software overhead.
- I report on our experience with implementing trust assessment modules – used to measure the state of unprotected code – on Sancus.
- I design and implement a deployment strategy for securing distributed applications on Sancus and demonstrate how to securely perform I/O operations from these applications.

[1]Sancus was the ancient Roman god of trust, honesty and oaths.

- I discuss a number of interesting design decisions and potential alternatives to, for example, the way Sancus implements its memory access logic or uses cryptographic primitives.

To guarantee the reproducibility and verifiability of my results, all my research materials, including the hardware design of the processor, the C compiler, and the deployment stack are publicly available [71].

Since research is a collaborative effort, I cannot take credit for each and every aspect of Sancus described in this thesis. In particular, I would like to thank Jan Tobias Mühlberg for his work on the trust assessment modules (Chapter 8) and Pieter Maene for implementing confidential loading (Section 3.6) and measuring the ASIC area of Sancus (Section 5.2).

Besides the results described in this thesis, I have made the following contributions:

- I designed and implemented a novel compiler technique to protect against return-address smashing attacks [73]. This technique works by duplicating functions in such a way that every function always returns to the same location, allowing the compiler to replace return instructions by direct jumps.

- I assisted the design and implementation of a secure resource sharing technique for Sancus [87]. Since Sancus' access control rules are rigid, it is difficult to share resources between modules. This work explores the use of secure resource sharing modules to implement sharing policies between other modules.

- I helped exploring how Sancus should be adapted in order to be able to provide availability and real-time guarantees [86].

The remainder of this thesis is structured as follows. In Part I, I first clarify the problem I address by defining the system model, attacker model, and the security properties I aim for (Chapter 2). Then, I detail the design of Sancus (Chapter 3) and some interesting implementation aspects (Chapter 4). Chapter 5 reports on my evaluation of Sancus and I conclude this part by giving an overview of related work (Chapter 6). Part II describes two interesting application of the Sancus architecture. First, I extend Sancus' security guarantees to distributed applications (Chapter 7). Then, I show how Sancus can be used to perform trust measurements in order to provide security guarantees about unprotected code (Chapter 8). I end this thesis in Part III by first discussing some of Sancus' design decisions (Chapter 9) and finally concluding (Chapter 10).

Part I

Core Architecture

Chapter 2

Problem Statement

Before introducing the design of the Sancus architecture (Chapter 3), this chapter introduces the context in which this design is supposed to work. The goal of this chapter is to precisely state the security properties Sancus aims to provide (Section 2.3). To be able to do this, we will start with describing the type of system that we target (Section 2.1) and the attacks we want to prevent (Section 2.2).

2.1 System Model

We consider a setting where a single infrastructure provider, *IP*, owns and administers a (potentially large) set of microprocessor-based systems that we refer to as *nodes* N_i. A variety of third-party *software providers* SP_j are interested in using the infrastructure provided by *IP*. They do so by deploying *software modules* $SM_{j,k}$ on the nodes administered by *IP*. Figure 2.1 provides an overview.

This abstract setting is an adequate model for many ICT systems today, and the nodes in such systems can range from high-performance servers (for instance in a cloud system), over smart cards (for instance in GlobalPlatform-based systems [41]) to tiny microprocessors (for instance in sensor networks). In this thesis, we focus on the low-end of this spectrum, where nodes contain only a small embedded processor that does not support a Memory Management Unit (MMU), protection rings, hypervisors, or other security mechanisms typically found on high-end processors.

Any system that supports extensibility (through installation of software modules) by several software providers must implement measures to make sure that the different modules cannot interfere with each other in undesired ways, either because of bugs in the software or because of malice. For high- to mid-end

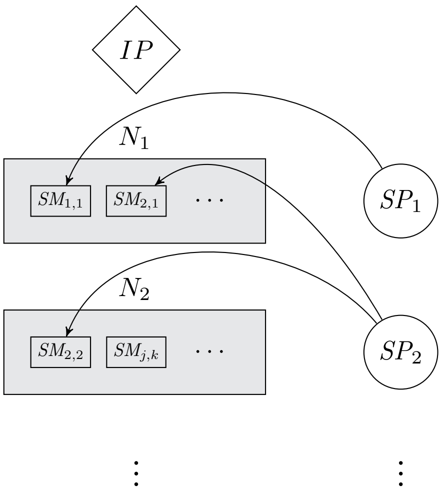

Figure 2.1: Overview of our system model. *IP* provides a number of nodes N_i on which software providers SP_j can deploy software modules $SM_{j,k}$.

systems, this problem is relatively well-understood, and good solutions exist. Two important classes of solutions are (1) the use of virtual memory, where each software module gets its own virtual address space, and where an OS or hypervisor implements and guards communication channels between them (for instance shared memory sections or inter-process communication channels), and (2) the use of a memory-safe VM, for instance a Java VM, where software modules are deployed in memory-safe bytecode and a security architecture in the VM guards the interactions between them.

For low-end systems with cheap microprocessors, providing adequate security measures for the setting sketched above is still an open problem, and an active area of research [32]. One straightforward solution is to transplant the higher-end solutions to these low-end systems: one can extend the processor with virtual memory, or implement a Java VM. This will be an appropriate solution in some contexts, but there are two important disadvantages. First, the cost (in terms of required resources such as chip surface, power or performance) is non-negligible. And second, these solutions all require the presence of a sizable trusted software layer (either the OS or hypervisor, or the VM implementation).

The problem we address in this thesis is the design, implementation and evaluation of a novel security architecture for low-end systems that is inexpensive and does not rely on any trusted software layer. The TCB on the networked device is *only* the hardware. More precisely, a software provider needs to trust only the hardware of the infrastructure and his own modules; he does not need to trust any infrastructural or third-party software on the nodes.

2.2 Attacker Model

We consider attackers with two powerful capabilities. First, we assume attackers can manipulate *all* the software on the nodes. In particular, attackers can act as a software provider and can deploy malicious modules to nodes. Attackers can also tamper with the operating system (for instance because they can exploit a buffer overflow vulnerability in the operating system code), or even install a completely new operating system.

Second, we assume attackers can control the communication network that is used by software providers and nodes to communicate with each other. Attackers can sniff the network, can modify traffic, or can mount man-in-the-middle attacks. Note that the security of the communication channel between *IP* and software providers is out of scope.

With respect to the cryptographic capabilities of the attacker, we follow the Dolev-Yao attacker model [25]: attackers cannot break cryptographic primitives, but they can perform protocol-level attacks.

Finally, attacks against the hardware of individual nodes are out of scope. We assume the attacker does not have physical access to the hardware, cannot place probes on the memory bus, cannot disconnect components, and so forth. While physical attacks are important, the addition of hardware-level protections is an orthogonal problem that is an active area of research in itself [49, 50, 14, 6]. The addition of hardware-level protection will be useful for many practical applications (in particular for sensor networks) but does not have any direct impact on our proposed architecture or on the results of this thesis.

Although our attacker model excludes hardware attacks, our security properties do limit the consequences of such an event (Section 2.3, *hardware breach confinement*).

2.3 Security Properties

For the system and attacker model described above, we want our security architecture to enforce the following security properties:

- *Software module isolation.* Software modules on a node run *isolated* in the sense that no software outside the module can access its runtime state and code. The only way for other software on the node to interact with a module is by calling one of its designated entry points.

- *Remote attestation.* A software provider can verify the state of a software module they deployed. That is, they can verify the integrity of a loaded module and that it has been isolated on a specific node of *IP*.
- *Secure communication.* A software provider can communicate with a specific software module on a specific node with confidentiality, integrity, and authenticity guarantees.
- *Secure linking.* A software module on a node can securely link with other modules on the same node. That is, whenever calling (or being called by) another module, a module can verify that the callee (caller) is the intended module. Also, the runtime interactions between these modules cannot be observed or tampered with by other software on the same node.
- *Confidential deployment.* If so desired, a software provider can deploy modules with the guarantee that no attacker will be able to access the module's code at any point in time.
- *Hardware breach confinement.* If an attacker manages to breach the hardware of a node, they may be able to manipulate or impersonate modules running on *that* node. However, such a breach should not allow them to do the same with modules running on *other* nodes.

Obviously, these security properties are not entirely independent of each other. For instance, it does not make sense to have secure communication but no isolation: given the power of our attackers, any message could then simply be modified right after its integrity was verified by a software module.

Note that the isolation property is valid at the machine code level of the underlying architecture. That is, the state of a module can be fully understood based on its machine code and the interactions with its entry points. If a module is compiled from a higher level programming language, special care must be taken to ensure that the expected isolation properties are preserved at the machine code level; especially for languages that allow for undefined behavior to occur. Agten et al. [2] show how one can securely compile higher level programming languages to an architecture that supports our isolation properties and our compiler (Section 4.2) implements many of their techniques.

Chapter 3

Design of Sancus

This chapter details the design of the Sancus architecture; that is, how we achieve the desired security properties in the context of our system- and attacker model (Chapter 2).

The main design challenge is to realize the desired security properties *without trusting any software on the nodes.* That is, software providers should not have to trust any software other than the modules they deployed or explicitly choose to trust. Another constraint is that nodes are low-end, resource-constrained devices. An important first design choice that follows from the resource-constrained nature of nodes is that we limit cryptographic techniques to symmetric key, in particular authenticated encryption. While public key cryptography would simplify key management, the cost of implementing it in hardware is too high [55]. Section 9.2 discusses the impact asymmetric cryptography would have on Sancus' design.

We first present some cryptographic primitives that will be used in the rest of this thesis (Section 3.1). Then, we give an overview of our design (Section 3.2) followed by the elaboration of its most interesting aspects (Sections 3.3 to 3.7). We conclude this chapter with an end-to-end example (Section 3.8).

3.1 Cryptographic Primitives

Throughout the design of Sancus, we assume the existence of three cryptographic primitives. First, a classical cryptographic hash function is used to compute digests of data. Second, a *key derivation function* is used to derive a cryptographic key from a master key and some diversification data:

$$K_{M,D} = \mathit{kdf}(K_M, D)$$

Third, an *authenticated encryption with associated data* primitive is used to simultaneously provide confidentiality, integrity, and authenticity guarantees on data. Such a primitive consists of two functions: one for encryption and one for decryption. The encryption function takes as input a key, plaintext, and associated data and produces ciphertext and an authentication tag. The ciphertext covers the given plaintext and the tag is a Message Authentication Code (MAC) over both the plaintext and the associated data:

$$C, T = \textit{aead-encrypt}(K, P, A)$$

The decryption function does the opposite operation and fails (i.e., produces no plaintext) if the tag is incorrect for the given ciphertext and associated data:

$$P = \textit{aead-decrypt}(K, C, A, T)$$

Note that this primitive can be used to compute a plain MAC over some data:

$$\textit{mac}(K, D) \equiv \pi_2(\textit{aead-encrypt}(K, \{\}, D))$$

(Here, we discard the ciphertext result of *aead-encrypt*.)

3.2 Overview

3.2.1 Nodes

Nodes are low-cost, low-power microcontrollers (our implementation is based on the TI MSP430). The processor in a node uses a Von Neumann architecture with a single address space for instructions and data. To distinguish actual nodes belonging to *IP* from fake nodes set up by an attacker, *IP* shares a symmetric key with each of its nodes. We call this key the *node master key*, and use the notation K_N for the node master key of node N. Given our attacker model where the attacker can control all software on the nodes, it follows that this key must be managed by the hardware, and it is only accessible to software in an indirect way.

3.2.2 Software Providers

Software providers are principals that can deploy software to the nodes of *IP*. Each software provider has a unique public ID *SP*.[1] *IP* uses a key derivation function *kdf* to compute a key $K_{N,SP} = \textit{kdf}(K_N, SP)$, which *SP* will later use to setup secure communication with its modules. Since node N has key K_N, it can compute $K_{N,SP}$ for any *SP*. The node will include a hardware implementation of *kdf* so that the key can be computed without trusting any software.

[1]Throughout this text, we will often refer to a software provider using its ID *SP*.

3.2.3 Software Modules

Software modules are essentially simple binary files containing two mandatory sections: a *text section* containing code and constants and a *data section* containing a module's runtime data. As we will see later, the contents of the latter section are not attested and are therefore vulnerable to malicious modification before hardware protection is enabled. Therefore, the processor will zero-initialize its contents at the time the protection is enabled to ensure an attacker cannot have *any* influence on a module's initial state. Next to the two protected sections discussed above, a module can opt to load a number of *unprotected sections* containing code or data. This is useful to, for example, limit the amount of code that can access protected data. Indeed, allowing code that does not need it access to protected data increases the possibility of bugs that could leak data outside of the module. In other words, this gives developers the opportunity to keep the trusted code of their own modules *as small as possible*. Each section has a header that specifies the start and end address of the section.

The *identity* of a software module consists of a hash of (1) the contents of the text section and (2) the start and end addresses of the text and data sections. We refer to this second part of the identity as the *layout* of the module. It follows that two modules with the exact same code and data can coexist on the same node and will have different identities as their layout will be different. We will use notations such as SM or SM_1 to denote the identity of a specific software module.

Software modules are always loaded on a node on behalf of a specific software provider SP. A software module is deployed by loading each of the sections of the module in memory at the specified addresses. For each module, the processor maintains the layout information in a *protected storage* area inaccessible from software. It follows that the node can compute the identity of all modules loaded on it: the layout information is present in protected storage and the contents of the text section are in memory.

An important sidenote here is that the loading process is *not* trusted. It is possible for an attacker to intervene and modify the module during loading. However, this will be detected as soon as the module communicates with its provider (Section 3.5).

Finally, the node computes a symmetric key $K_{N,SP,SM}$ that is specific to the module SM loaded on node N by provider SP. It does so by first computing $K_{N,SP} = kdf(K_N, SP)$ as discussed above, and then computing $K_{N,SP,SM} = kdf(K_{N,SP}, SM)$. All these keys are computed by hardware (Section 3.4) and kept in the protected storage, and will only be available to software indirectly by means of new processor instructions we discuss later. Table 3.1 gives an overview of the keys used by Sancus.

Table 3.1: Overview of the keys used in Sancus, how they are created and who can access them. Note that derived keys are also accessible by any entity that has access to their master keys but this is not explicity mentioned.

Key	Creation	Accessible by
K_N	Random	IP, N
$K_{N,SP}$	$kdf(K_N, SP)$	SP
$K_{N,SP,SM}$	$kdf(K_{N,SP}, SM)$	SM (indirectly)

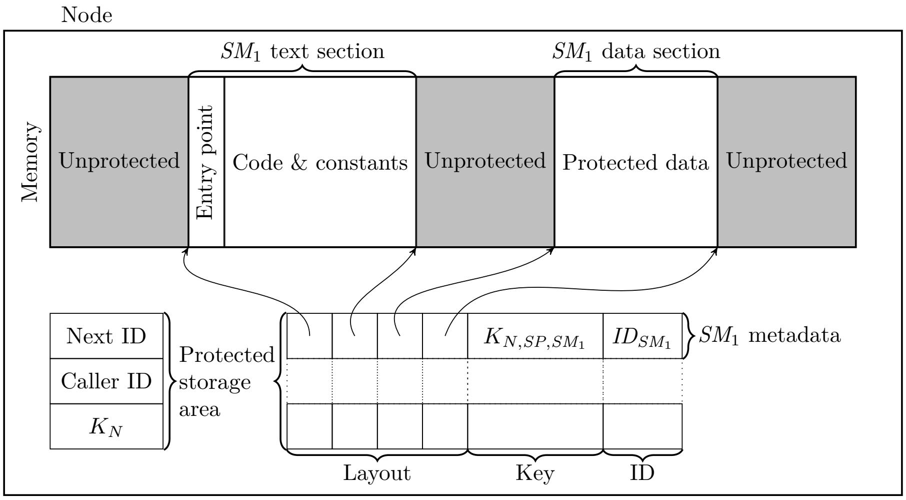

Figure 3.1: A node with a software module loaded. The left part of the protected storage area is global while the right part is per module metadata. Sancus ensures the keys can never leave the protected storage area by only making them available to software in indirect ways through new processor instructions.

Note that the provider SP can also compute $K_{N,SP,SM}$, since he received $K_{N,SP}$ from IP and since he knows the identity SM of the module he is loading on N. This key will be used to attest the presence of SM on N to SP and to secure the communications between SM and SP.

Figure 3.1 shows a schematic of a node with a software module loaded. The picture also shows the keys and the layout information that the node has to manage. Section 3.8 details the loading process through an end-to-end example.

3.2.4 Memory Protection on the Nodes

The various modules on a node must be protected from interfering with each other in undesired ways by means of some form of memory protection. Our design relies on PCBAC [84], as this memory access control model has been shown to support strong isolation [83], as well as remote attestation [29]. Roughly speaking, isolation is implemented by restricting access to the data section of a module such that it is only accessible while the program counter is in the corresponding text section of the same module. Moreover, the processor instructions that use the keys $K_{N,SP,SM}$ will be program counter-dependent. Essentially, the processor offers special instructions to access the cryptographic capabilities. If such an instruction is invoked from within the text section of a specific module *SM*, the processor will use key $K_{N,SP,SM}$. Moreover, these instructions are only available after memory protection has been enabled for module *SM*. It follows that only a well-isolated *SM* installed on behalf of *SP* on *N* can compute cryptographic primitives with $K_{N,SP,SM}$, and this is the basis for implementing both remote attestation and secure communication.

3.2.5 Remote Attestation and Secure Communication

In order to provide a confidential, integrity-protected, and authenticated communication channel between a software provider and its modules, Sancus includes an authenticated encryption primitive. New instructions are provided to encrypt and decrypt messages using the key of the calling module. When *SP* receives a message encrypted with $K_{N,SP,SM}$, they will have high assurance that the message has been produced by *SM* since, as mentioned above, only *SM* is able to use this key. Note that since *SM* is the *identity* of the module that *SP* is communicating with, this primitive also provides for remote attestation.

3.2.6 Secure Linking

A final aspect of our design is how we deal with secure linking. When a software provider sends a module SM_1 to a node, this module can specify that it wants to link to another module SM_2 on the same node, so that SM_1 can call services of SM_2 locally. SM_1 specifies this by including the identity (i.e., a hash) of SM_2 in its text section.[2] The processor includes a new instruction that SM_1 can call to check that (1) there is a module loaded (with memory protection enabled) at the address of SM_2 and (2) the identity of that module has the expected value.

[2]Note that if SM_2 also wants to link to SM_1, this method creates a circular dependency between their identities. This can be resolved by not including the other's identity in the text section but having the software provider securely send it after deployment and storing it in the data section.

A similar mechanism can be used by SM_2 to verify that it is indeed called by SM_1 (*caller authentication*). In its entry point, SM_2 can call a new instruction that verifies the identity of the module that called the entry point. For this to work, the processor keeps track of the *previously* executing software module.

Fortunately, this expensive hash calculation over a potentially large text section is only needed during the first authentication. Section 3.7 discusses a more efficient procedure for subsequent authentications.

3.3 Key Management

We handle key management without relying on public-key cryptography. *IP* is a trusted authority for key management. All keys are generated and/or known by *IP*. There are three types of keys in our design (Table 3.1):

- Node master keys K_N shared between node N and *IP*.
- Software provider keys $K_{N,SP}$ shared between a provider *SP* and a node N.
- Software module keys $K_{N,SP,SM}$ shared between a node N and a provider *SP*, and the hardware of N makes sure that only *SM* can use this key.

We have considered various ways to manage these keys. A first design choice is how to generate the node master keys. We considered three options: (1) using the same node master key for every node, (2) randomly generating a separate key for every node using a secure random number generator and keeping a database of these keys at *IP*, and (3) deriving the master node keys from an *IP* master key using a key derivation function and the node identity N.

We discarded option (1) because for this choice the compromise of a single node master key breaks the security of the entire system, hence violating *hardware breach confinement* (Section 2.3). Options (2) and (3) are both reasonable designs that trade off the amount of secure storage and the amount of computation at *IP*'s site. Our prototype uses option (2).

The software provider keys $K_{N,SP}$ and software module keys $K_{N,SP,SM}$ are derived using a key derivation function as discussed in the overview section.

Finally, an important question is how compromised keys can be handled in our scheme. Since any secure key derivation function has the property that deriving the master key from the derived key is computationally infeasible, the compromise of neither a module key $K_{N,SP,SM}$ nor a provider key $K_{N,SP}$ needs to lead to the revocation of K_N. If $K_{N,SP}$ is compromised, provider *SP* should receive a new name *SP'* since an attacker can easily derive $K_{N,SP,SM}$ for any *SM* given $K_{N,SP}$. If $K_{N,SP,SM}$ is compromised, the provider can still safely deploy other modules. *SM* can also still be deployed if the provider makes a change to the text section of SM.[3] If K_N is compromised, it needs to be revoked. Since K_N is different for every node, this means that only one node needs to be either replaced or have its key updated.

3.4 Memory Access Control

Memory can be divided into (1) memory belonging to modules, and (2) the rest, which we refer to as unprotected memory. Memory allocated to modules is divided into two sections, the text section, containing code and constants, and the data section containing all the data that should remain confidential and should be integrity protected. Modules can also have an unprotected data section that is considered to be part of unprotected memory from the point of view of the memory access control system.

Apart from application-specific data, runtime metadata such as the module's call stack should typically also be included in the data section. Indeed, if a module's stack were to be shared with untrusted code, confidential data may leak through stack variables or control-data might be corrupted by an attacker. It is the module's responsibility to make sure that its call stack and other runtime metadata is in its data section, but our implementation comes with a compiler that ensures this automatically (Section 4.2).

The memory access control logic in the processor enforces that (1) the data section of a module is only accessible while code in the text section of that module is being executed, (2) the text section can only be executed by jumping to a well-defined entry point, and (3) the text section cannot be written and can only be read while code in that section is being executed. The second part is important since it prevents attackers from misusing code chunks in the text section to extract data from the data section. For example, without this guarantee, an attacker might be able to launch a Return-Oriented Programming (ROP) attack [17] by selectively combining gadgets found in the text section. Of course, if a module contains a bug that allows an attacker to divert its control-flow, he might still be able to launch such an attack; enforcing an entry point prevents these attacks being launched from code outside of the module. Note that, as shown in Figure 3.1, our design allows modules to have a single entry point only. This may seem like a restriction but, as we will show in Section 4.2, it is not since multiple logical entry points can easily be dispatched through a single physical entry point. Table 3.2 gives an overview of the enforced access rights.

Besides memory access control, the processor also ensures that modules cannot be interrupted while being executed to prevent register contents from

[3]For example, a random byte could be appended to the text section without changing the semantics of the module.

Table 3.2: Memory access control rules enforced by Sancus using the traditional Unix notation. Each entry indicates how code executing in the "from" section may access the "to" section.

From/to	Entry	Text	Data	Unprotected
Entry	`r-x`	`r-x`	`rw-`	`rwx`
Text	`r-x`	`r-x`	`rw-`	`rwx`
Unprotected/ Other SM	`--x`	`---`	`---`	`rwx`

leaking outside the module. Supporting interruptible modules is orthogonal to our goals and is left as future work.

Memory access control for a module is enabled at the time the module is loaded. First, untrusted code (for instance the node's operating system) will load the module in memory as discussed in Section 3.2. Then, a special instruction is issued:

`protect` *layout, SP*

This processor instruction has the following effects:

- the layout is checked not to overlap with existing modules, and a new module is registered by storing the layout information in the protected storage of the processor (Figure 3.1);
- memory access control is enabled as discussed above; and
- the module key $K_{N,SP,SM}$ is calculated – using the text section and layout of the actually loaded module – and stored in the protected storage.

This explains why we do not need to trust the OS that loads the module in memory: if the content of the text section or the layout[4] would be modified before execution of the `protect` instruction, then the key generated for the module would be different, and subsequent attestations or authentications performed by the module would fail. Once the `protect` instruction has succeeded, the hardware-implemented memory access control scheme ensures that software on the node can no longer tamper with *SM*.

The only way to lift the memory access control is by calling the processor instruction:

`unprotect` *continuation*

[4]Note, however, that it is still possible to dynamically link modules at load time. See Section 4.3 for details.

The effect of this instruction is to lift the memory protection of the module *from which the* `unprotect` *instruction is called.* To prevent the leakage of confidential data, this instruction also clears the module's code- and data sections. Since the `unprotect` instruction itself is part of the code section, the programmer has to provide a pointer to the code where the execution is to be continued in the *continuation* argument.

Finally, it remains to decide how to handle memory access violations. We opt for the simple design of resetting the processor and clearing memory on every reset. This has the advantage of clearly being secure for the security properties we aim for. However an important disadvantage is that it may have a negative impact on availability of the node: a bug in the software may cause the node to reset and clear its memory. An interesting avenue for future work is to come up with strategies to handle memory access violations in less severe ways. Invalid reads could return some default value as in secure multi-execution [23]. Invalid writes or jumps could be dropped or modified to actions that are allowed as in edit-automata [60]. For instance, an invalid memory read might just return zero, and an invalid jump might be redirected to an exception handler.

3.5 Remote Attestation and Secure Communication

We extend the processor with two more instructions that are used for remote attestation and secure communication:

`encrypt` *plaintext, associated data, ciphertext (output), tag (output)[, key]*
`decrypt` *ciphertext, associated data, tag, plaintext (output)[, key]*

These instructions have the same semantics as the *aead-encrypt* and *aead-decrypt* functions, respectively (Section 3.1).

As can be seen from the signatures above, both instructions have the key as an optional argument. If none is given, the module key of the invoking module is implicitly used (or an error code is returned if invoked by unprotected code). This is the *only* way for software to access a module key and the key $K_{N,SP,SM}$ will *only* be used when invoked by the module with identity *SM* deployed by *SP* on node N. Note that, besides being able to access the module key, these instructions are *not* privileged and the same memory access rules are enforced as for any instruction that accesses memory.

These instructions can be used to provide confidentiality, integrity, and authenticity guarantees of data exchanged between modules and their providers. The ciphertext plus the corresponding tag can be sent using the untrusted operating system over an untrusted network. If the tag verifies correctly (using $K_{N,SP,SM}$) upon receipt by the provider *SP*, they can be sure that the decrypted plaintext indeed comes from *SM* running on N on behalf of *SP* as the node's hardware makes sure only this specific module can use the module key $K_{N,SP,SM}$. The reasoning is equivalent for data sent to the module.

To implement remote attestation, we only need to add a freshness guarantee (i.e., protect against replay attacks). Provider *SP* sends a fresh nonce *No* to the node N, and the module *SM* returns the MAC of this nonce using the key $K_{N,SP,SM}$, computed with the `encrypt` instruction (Section 3.1 explains how this can be done). This gives the *SP* assurance that the correct module is running on that node at this point in time.

Building on this scheme, we can also implement secure communication. Whenever *SP* wants to receive data from *SM* on N, it sends a request to the node containing a nonce *No* and possibly some input data I that is to be provided to *SM*. This request is received by untrusted code on the node which passes *No* and I as arguments to the function of *SM* to be called. When *SM* has calculated the output O, it asks the processor to calculate $\mathit{aead\text{-}encrypt}(K_{N,SP,SM}, O, \mathit{No}||I)$ using the `encrypt` instruction. The resulting ciphertext and tag are then sent to *SP*. By verifying the tag with its own copy of the module key, the provider has strong assurance that O has been produced by *SM* on node N given input I.

3.6 Confidential Loading

If, besides the integrity guarantees provided through remote attestation, one wants to have *confidentiality guarantees* for a module's text section, more architectural support is necessary. Indeed, although a module's text section is not readable by other modules (Table 3.2), this is only enforced after enabling a module; i.e., up to that point an attacker can easily read the module's text section.

Therefore, we provide a second way to use the `protect` instruction:

`protect` *layout*, *SP*, *MAC*

In this form, the `protect` instruction behaves exactly the same (Section 3.4) except that, before calculating the module key, the module's text section is decrypted in place using $K_{N,SP}$. If the integrity check using the given MAC, i.e., an authenticated encryption tag, fails, the text section is cleared and the protection disabled.

Note that $K_{N,SP}$ is now used to encrypt confidential modules, as well as for key derivation. However, the uniqueness of both operations can be guaranteed by domain separation, i.e., by setting the first bit of the associated data to 0 or 1 for the encryption and key derivation respectively.

It should be mentioned that the integrity check is not strictly necessary for confidential loading, since any subsequent remote attestation will also verify the module's integrity. However, it could be used as a simple form of module authentication: by disabling the non-decrypting form of the `protect` instruction, only entities possessing a valid software provider key can install modules on the system.

3.7 Secure Linking and Local Communication

In this section, we discuss how we assure the secure linking property mentioned in Section 2.3. More specifically, we consider the situation where a module SM_1 wants to call another module SM_2 and wants to be ensured that (1) the integrity of SM_2 has not been compromised, and (2) SM_2 is correctly protected by the processor.

In our design, if module SM_1 wants to link securely to SM_2, SM_1 should be deployed with the identity of SM_2. The processor provides a special instruction to check the existence and integrity of a module at a specified address:

`attest` *address, expected hash*

This instruction will:

- verify that a module is loaded (with protection enabled) at the provided address;
- compute the identity of that module (i.e., a hash of its text section and layout);
- compare the resulting hash with the *expected hash* parameter of the instruction; and
- if the hashes are equal, return the module's ID (to be explained below), otherwise return zero.

Using this processor instruction, a module can securely check for the presence of another expected module, and can then call that other module.

Since this authentication process is relatively expensive (it requires the computation of a hash), our design also includes a more efficient mechanism for repeated authentication. The processor will assign sequential IDs[5] to modules that it loads, and will ensure that – within one boot cycle – it never reuses these IDs. This can be implemented by storing the ID to be used for the next module in a register ("Next ID" in Figure 3.1), incrementing it after a new module is enabled, and generating a violation when it overflows. A processor instruction:

[5]To avoid confusion between the two different identity concepts used in this text, we will refer to the hardware-assigned number as *ID* while the text section and layout of a module is referred to as *identity*.

`get-id` *address*

checks that a protected module is present at *address* and returns the ID of the module. Once a module has checked using the initial authentication method that the module at a given address is the expected module, it can remember the ID of that module, and then for subsequent authentications it suffices to check that the same module is still loaded at that address using the `get-id` instruction.

For caller authentication, the processor keeps track of the previously executing module by recording its ID in a register ("Caller ID" in Figure 3.1). This register is updated whenever execution enters a new module. Modules can attest their caller through two instruction: `attest-caller` and `get-caller-id`. These instructions behave similar to `attest` and `get-id` respectively but use the previously executing module implicitly.

3.8 An End-to-End Example

To make the discussion in the previous sections more concrete, this section gives a small example of how our design may be applied in the area of sensor networks. Figure 3.2 shows our example setup. It contains a single node to which a sensor S is attached; communication with S is done through memory-mapped I/O. The owner of the sensor network, IP, has deployed a special module, SM_S, that is in charge of communicating with S. By ensuring that the data section of SM_S contains the memory-mapped I/O region of S, IP ensures that no software outside of SM_S is allowed to configure or communicate directly with S; all requests to S need to go through SM_S. Section 7.3.1 goes into more detail on how to write secure I/O drivers using Sancus.

Figure 3.2 also shows a number of software providers $(SP_1, \ldots, SP_n)$ who have each deployed a module $(SM_1, \ldots, SM_n)$. In the remainder of this section, we walk the reader through the life cycle of a module in this example setup.

The first step for a provider SP is to contact IP and request permission to run a module on the sensor node. If IP accepts the request, it provides SP with its provider key for the node, $K_{N,SP}$.

Next, SP creates the module SM that he wants to run on the processor and calculates the associated module key $K_{N,SP,SM}$. Since SM will communicate with SM_S, SP requests the identity of SM_S from IP. This identity is included in the text section of SM, so that SM can use it to authenticate SM_S. Then SM is sent to the node for deployment.

Once SM is received on the node, it is loaded – by untrusted software like the operating system – into memory and the processor is requested to protect SM, using the `protect` processor instruction. As discussed, the processor enables memory protection, computes the key $K_{N,SP,SM}$, and stores it in hardware.

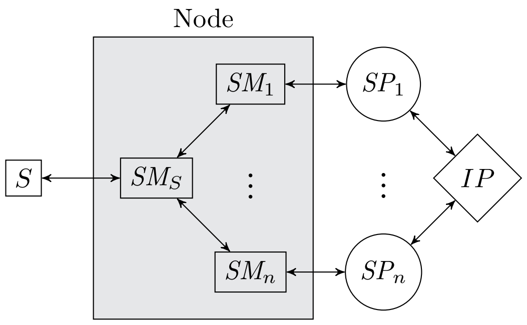

Figure 3.2: Setup of the sensor node example discussed in Section 3.8. Sancus ensures only module SM_S is allowed to directly communicate with the sensor S. Other modules securely link to SM_S to receive sensor data in a trusted way.

Now that *SM* has been deployed, *SP* can start requesting data from it. We will assume that *SM*'s function is to request data from S through SM_S, perform some transformation, filtering, or aggregation on it, and return the result to *SP*. The first step is for *SP* to send a request containing a nonce *No* to the node. Once the request is received (by untrusted code) on the node, *SM* is called passing *No* as an argument.

Before *SM* calls SM_S, it needs to verify the integrity of module SM_S. It does this by executing the `attest` instruction passing the address of the expected identity of SM_S (included in *SM*'s text section) and the address of the entry point it is about to call. The ID of SM_S is then returned to *SM* and, if it is non-zero, *SM* calls SM_S to receive the sensor data from S. *SM* will usually also store the returned ID of SM_S in its data section so that future authentications of SM_S can be done with the `get-id` instruction.

Once the received sensor data has been processed into the output data O, *SM* will request the processor to calculate $\mathit{aead\text{-}encrypt}(K_{N,SP,SM}, O, \mathit{No})$ using the `encrypt` instruction. *SM* then passes the ciphertext C and tag T to the (untrusted) network stack to be sent to *SP*. When *SP* receives the output of *SM*, it can verify its integrity by calculating $\mathit{aead\text{-}decrypt}(K_{N,SP,SM}, C, \mathit{No}, T)$.

Chapter 4

Implementation

This chapter discusses the implementation of Sancus. We have implemented hardware support for all security features discussed in Chapter 3, as well as a compiler that can create software modules suitable for deployment on the hardware.

4.1 The Processor

Our hardware implementation is based on an open source implementation of the TI MSP430 architecture, the openMSP430 from the OpenCores project [40]. We have chosen this architecture because both GCC and LLVM support it, and there exists a lot of software running natively on the MSP430, for example the Contiki [27] operating system.

The discussion is organized as follows. First, we explain the features added to the openMSP430 in order to implement the isolation of software modules. Then, we discuss how we added support for the cryptographic operations. Finally, we describe the modifications we made to the openMSP430 core itself.

4.1.1 Isolation

This part of the implementation deals with enforcing the access rights shown in Table 3.2. For this purpose, the processor needs access to the layout of every software module that is currently protected. Since the access rights need to be checked on every instruction, accessing these values should be as fast as possible. For this reason, we decided to store the layout information in special registers inside the processor. Note that this means the total number of software modules that can be protected at any particular time has a fixed upper bound. This upper bound, N_{SM}, can be configured when synthesizing the processor.

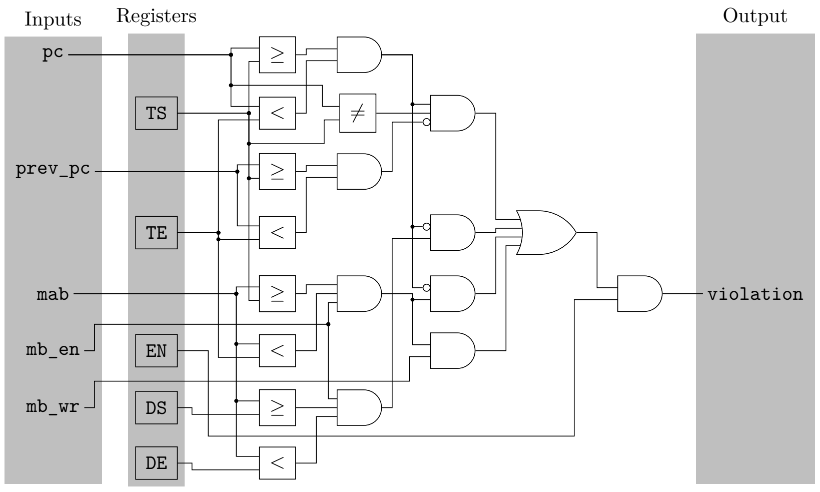

Figure 4.1: Schematic of the Memory Access Logic circuit, the hardware used to enforce the memory access rules for a single protected module.

Figure 4.1 gives an overview of the Memory Access Logic (MAL) circuit used to enforce the access rights of a single software module. This MAL circuit is instantiated N_{SM} times in the processor. It has five inputs: `pc` and `prev_pc` are the current and previous values of the program counter, respectively. The input `mab` is the memory address bus – the address currently used for load or store operations[1] – while `mb_en` indicates whether the address bus is enabled for the current instruction and `mb_wr` indicates whether the access is a write. The MAL circuit has one output, `violation`, that is asserted whenever one of the access rules is violated.

Apart from the input and output signals, the MAL circuit also keeps state in registers. The layout of the protected software module is captured in the `TS` (start of text section), `TE` (end of text section), `DS` (start of data section) and `DE` (end of data section) registers. The `EN` register is set to 1 if there is currently a module being protected by this MAL circuit instantiation. The layout is saved in the registers when the `protect` instruction is called, at which time `EN` is also set. When the `unprotect` instruction is called, we just unset `EN` which disables all checks.

Since the circuit is purely combinatorial, no extra cycles are needed for the enforcement of access rights. As explained above, this is exactly what we want since these rights need to be checked for every instruction. The only downside this approach might have is that the large combinatorial circuit adds to the

[1]Of course, this includes implicit memory accesses like a `call` instruction.

length of the critical path of the processor. We will explore the implications our design has on the processor's critical path in Section 5.1.

Apart from hardware circuit blocks that enforce the access rights, we also added a single hardware circuit to control the MAL circuit instantiations. It implements four tasks: (1) combine the `violation` signals from every MAL instantiation into a single signal; (2) keep track of the value of the current and previous program counter; (3) keep track of the currently and previously executing *SM*; and (4) when the `protect` instruction is called, select a free MAL instantiation to store the layout of the new software module and assign it a unique ID.

4.1.2 Cryptography

As explained in Section 3.1, a hardware implementation of three cryptographic primitives is needed to implement our design: authenticated encryption, key derivation and hashing. Since our implementation is based on a small micro-processor, one of our main goals here is to make the implementation of these features as small as possible.

We have chosen to build these cryptographic primitives on the SpongeWrap [12] authenticated encryption construction using spongent [13] as the underlying sponge function. Since keyed sponge functions are shown to be pseudorandom functions [8], we can reuse SpongeWrap to calculate MACs, and consequently for key derivation. Since the security of SpongeWrap relies on the soundness of the sponge function it uses, it can also be used as a hashing function by calling *aead-encrypt*$(\{\}, \{\}, M)$.

Besides being able to use it for all necessary primitives, there are several reasons we use SpongeWrap with spongent. Since the security of SpongeWrap is proportional to the capacity of the underlying sponge function[2], and spongent is defined for a large range of capacities, we can create an implementation with a selectable security parameter. More specifically, our core can be synthesized with a security parameter between 16 and 256 bits, although values less than 80 bits should be avoided. Since the security parameter influences the core's area (Section 5.2), it is a trade-off between cost and security.

As we will see later, all module keys are stored in hardware making the key size an important design parameter regarding area. Another advantage of SpongeWrap is that the key size may be as small as the security parameter whereas some other lightweight authenticated encryption primitives require a key that is twice as long, e.g., APE [7].

A downside of SpongeWrap is that uniqueness of the associated data is required for confidentiality, and no security guarantees can be given when a

[2]To be precise, for a sponge function with a capacity of c bits, SpongeWrap has a security of $c/2$ bits [12].

nonce is reused. More specifically, if two ciphertext messages are captured that are encrypted with the same key and associated data, part of the XOR of the corresponding plaintext message may be leaked (see [12] for details). Therefore, the user of this primitive should ensure that, for a specific key, the associated data is unique, i.e., that it includes a nonce. Note that this is only necessary when encrypting data and there is no nonce requirement for creating MACs. In contrast, *nonce-misuse resistant* authenticated encryption algorithms (e.g., APE mentioned above) limit information leakage about the message when the nonce is reused, but this comes at an additional implementation cost.

It is a software provider's responsibility that the nonce requirement is fulfilled by the modules it deploys. In our prototypes, this is achieved by having *SP* send an initial counter value as nonce in its first message to a newly deployed module. For subsequent messages, modules can simply increment the counter and use that value as the next nonce. Alternatively, if *SP* never wants to send messages to a module, the initial counter value can be included in the module's text section.

The node key K_N is fixed when the hardware is synthesized and should be created using a secure random number generator. When a module *SM* is loaded, the processor will first derive $K_{N,SP}$ using the SpongeWrap implementation which is then used to derive $K_{N,SP,SM}$. The latter key will then be stored in the hardware MAL instantiation for the loaded module. Note that we have chosen to cache the module keys instead of calculating them on the fly whenever they are needed. This is a trade-off between size and performance which is justified because, when using 128 bit keys, SpongeWrap needs about 90 cycles per input byte (Section 5.1). Since the module key is needed for every remote attestation and whenever the module's output needs to be encrypted, having to calculate it on the fly would introduce a runtime overhead that we expect to be too high for most applications.

Because of the associated data uniqueness requirement explained above, our implementation of confidential loading is slightly different from its design (Section 3.6). Since modules deployed on N by *SP* are always encrypted using $K_{N,SP}$, the `protect` instruction takes an extra argument, *nonce*, to be able to fulfill the nonce requirement. This argument is used as the associated data input for the decryption routine.

4.1.3 Core Modifications

The largest modification that had to be made to the core is the decoding of the new instructions. We have identified a range of opcodes, starting at `0x1380`, that is unused in the MSP430 instruction set and mapped the new instructions in that range.

Further modifications include routing the needed signals, like the memory address bus, into the access rights modules as well as connecting the violation signal to the internal reset. Note that the violation signal is stored into a register before connecting it to the reset line to avoid the asynchronous reset being triggered by combinatorial glitches from the MAL circuit.

Since our experience has shown that developing applications on a system that resets on violations is rather tedious, we added a synthesis option to generate a non-maskable interrupt instead. If this option is enabled, the memory backbone will disable all memory accesses when a violation is generated and the frontend will initiate the IRQ sequence. Although this may superficially seem secure, it brings with it a number of problems (e.g., if a module generates a violation, its register contents will be leaked) we have not dealt with yet. Therefore, it currently is not enabled by default and should not be used in production environments.

Figure 4.2 gives an overview of the added hardware blocks when synthesized with support for two protected modules. In order to keep the figure readable, we did not add the input and output signals of the MAL blocks shown in Figure 4.1.

4.2 The Compiler

Although the hardware modifications enable software developers to create protected modules, doing this correctly is tedious, as the module can have only one entry point, and as modules may need to implement their own call-stack to avoid leaking the content of stack allocated variables to unprotected code or to other modules. Hence, we have implemented a compiler extension based on LLVM [62] that deals with these low-level details. We have also implemented a support library that offers an API to perform some commonly used functions like calculating a MAC of data.

Our compiler compiles standard C files.[3] To benefit from Sancus, a developer only needs to indicate which functions should be part of the protected module being created, which functions should be entry points and what data should be inside the protected section. For this purpose, we offer three attributes – `SM_FUNC`, `SM_ENTRY` and `SM_DATA` – that can be used to annotate functions and global variables.

4.2.1 Entry Points

Since the hardware supports a single entry point per module only, the compiler implements multiple logical entry points on top of the single physical entry point by means of a jump table. The compiler assigns every logical entry point a unique ID. When calling a logical entry point, its ID is placed in a register before jumping to the physical entry point of the module. The code at the

[3]We use Clang [61] as our compiler frontend. This means any C-dialect accepted by Clang is supported.

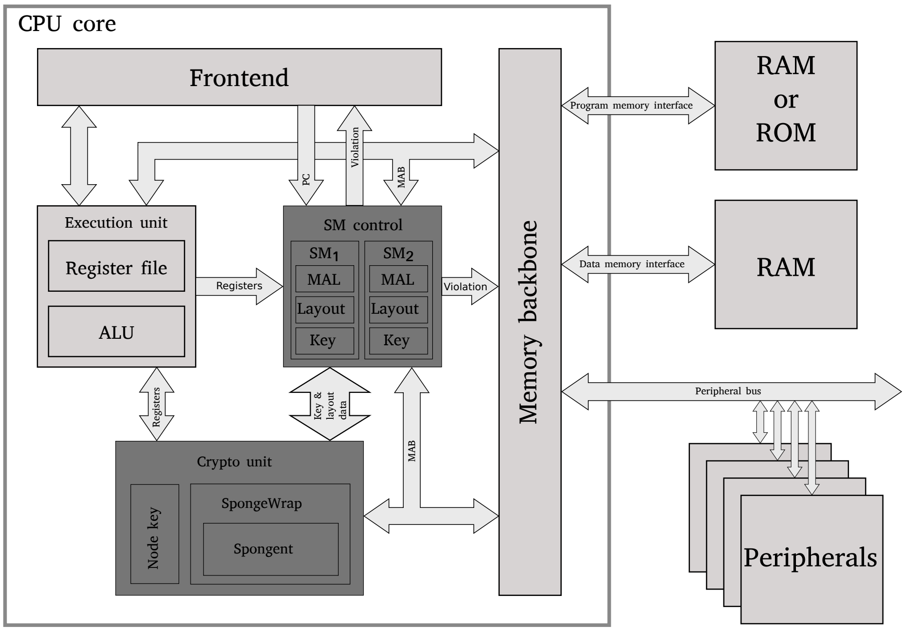

Figure 4.2: Overview of the hardware blocks in the Sancus core. Lightly shaded blocks are part of the original openMSP430 design while the darkly shaded ones are added specifically for Sancus. Remember that, while we only draw two *SM* blocks for clarity, this number (N_{SM}) can be chosen when synthesizing the core. Notice how the *SM* control unit takes the program counter (PC) and the memory address bus (MAB) as input to produce the violation signal using the MAL circuits.

physical entry point then jumps to the correct function based on the ID passed in the register.

When a module calls an external function, the same entry point is also used when this function returns. This is implemented by using a special ID for the "return entry point". If this ID is provided when entering the module, the address to return to is loaded from the module's stack. Of course, this is only safe if stack switching is enabled.

4.2.2 Stack Switching

As discussed in Section 3.4, it is preferable to place the runtime stack of software modules inside the data section. Our compiler automatically handles everything needed to support multiple stacks. For every module, space is reserved at a fixed location in its protected section for the stack. The first time a module is entered, the stack pointer is loaded with the address of this start location of the stack. When the module is exited, the current value of the stack pointer is stored in the protected section so that it can be restored when the module is reentered.

4.2.3 Exiting Modules

Our compiler ensures that no data is leaked through registers when exiting from a module. When a module exits, either by calling an external function or by returning, any register that is not part of the calling convention is cleared. That is, only registers that hold a parameter or a return value retain their value.

4.2.4 Secure Linking

Calls to protected modules are automatically instrumented to verify the called module. This includes automatically calculating any necessary module identities. The compiler also automatically uses the optimization described in Section 3.7.

4.3 Deployment

Since the identity of a module is dependent on its load addresses on node N, SP must be aware of these addresses in order to be able to calculate $K_{N,SP,SM}$. Moreover, any identity hashes needed for secure linking will also be dependent on the load addresses of other modules. Enforcing static load addresses is obviously not a scalable solution given that we target systems supporting dynamic loading of software modules by third-party software providers.

Given these difficulties, we felt the need to develop a proof-of-concept software stack providing a deployment solution. Our stack consists of two parts: a set of tools used by SP to deploy SM on N and host software running on N. Note that this host software is *not* part of any protected module and, hence, does not increase the size of the TCB.

We will now describe the deployment process implemented by our software stack. First, SP creates a relocatable Executable and Linkable Format (ELF) file of SM and sends it to N. The host software on N receives this file, finds a free memory area to load SM and relocates it using a custom made dynamic ELF loader. Then, hardware protection is enabled for SM and a symbol table is sent back to SP. This symbol table contains the addresses of any global functions[4] as well as the load addresses of all protected modules on N. Using this symbol table, SP is able to reconstruct the exact same image of SM as the one loaded on N which can then be used to calculate $K_{N,SP,SM}$.

[4]For example, `libc` functions and I/O routines.

Note that an alternative linking strategy is for *SP* to first request the node's symbol table, link the module locally and then send it to the node to be loaded. This would simplify the node since the custom ELF loader is not needed in this scheme. However, since our toolchain does not support position-independent code, this would mean that the memory locations where the module is going to be loaded need to be reserved while *SP* links the module. We feel that this two-phase deployment scheme adds more overall complexity than a simple ELF loader.

Since the dynamic loader needs to inspect and update parts of the text section of modules, this process does not work when confidential loading is used. Although our tools currently do not support the fully automatic loading of encrypted modules, it can be implemented as follows. First, *SP* sends a request to N indicating the sizes of the sections of the module it wants to load. Then, the host software allocates memory for those sections and replies with a handle identifying the allocated memory and a symbol table. Using this symbol table, *SP* links *SM* locally and sends the resulting image, together with the memory handle, back to N. The host software on N then loads it in the pre-allocated memory sections and enables its protection.

After *SM* has been deployed, the host software on N provides an interface to be able to call its entry points. This can be used by *SP* to attest that *SM* has not been compromised during deployment and that the hardware protection has been activated.

This interface is used to upload the identity hashes to *SM* of the modules it securely links to. To this end, *SP* either calculates these hashes after it received the symbol table or, if it concerns modules belonging to a different software provider that use confidential loading, receives them from their respective providers. Then, *SP* encrypts those hashes using $K_{N,SP,SM}$ and sends them to *SM* using the interface described above.

Chapter 5

Evaluation

In this chapter we evaluate Sancus in terms of runtime performance, impact on chip size, and provided security. All experiments were performed using a Xilinx XC6SLX25 Spartan-6 FPGA with a speed grade of -2, synthesised using Xilinx ISE Design Suite.

5.1 Performance

There are two important performance aspects to consider with our design. First, since we made changes to the CPU core, we evaluate the impact on its critical path, i.e., the maximum frequency it can run at. Second, we measure the runtime overhead of the added instructions, as well as the code transformations performed by the compiler.

5.1.1 Critical Path

Since the Xilinx tools offer no direct way to find the critical path of a design, we measured it indirectly. Using timing constraints, one can specify what clock rate certain signals should be able to sustain; the tools will then err when the constraint cannot be met. By varying the constraint on the input clock signal, we can get a measure on how fast the design will be able to run and, thus, on the length of the critical path.

We found that the unmodified openMSP430 core can run at 51 MHz with our setup. For our modifications, there are two parameters that may influence the critical path: the security level (i.e., the size of the keys) and the number of supported modules N_{SM}. The reason these parameters may influence the critical path is the same for both. The keys are stored in the MAL circuits and routed to the crypto unit through a multiplexer. Both the key size and N_{SM}

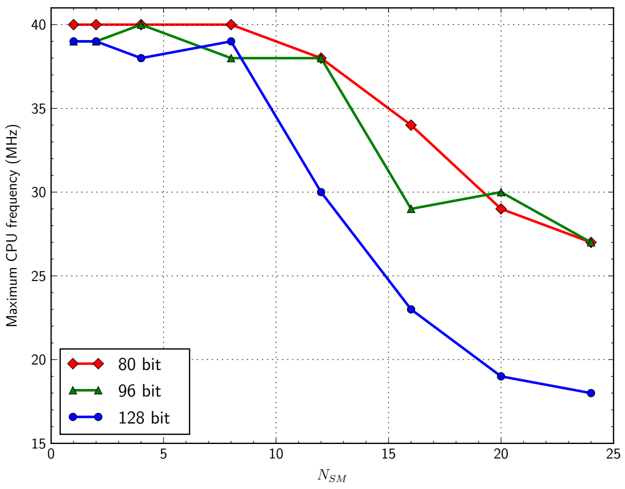

Figure 5.1: The maximum frequency for which the core can be synthesized in function of the number of modules (N_{SM}) for a number of security levels. The maximum frequency decreases with N_{SM} due to the large multiplexer needed to get the module key out of the MAL circuits. This also explains why the maximum frequency decreases much faster when the key size is larger. Note that the unmodified openMSP430 core can be synthesized at 51 MHz.

will increase the size of this multiplexer, and hence increase the length of the critical path.

Figure 5.1 shows the maximum obtainable frequency in function of N_{SM} for a number of different security levels. Note that although our implementation allows for security levels up to 256 bits, 128 bits are ample for our target platforms, and we therefore do not evaluate higher security levels.

The influence of the security level and N_{SM} should be clear from the figure. However, it should also be clear that the maximum frequency for any security level is not influenced much by values of N_{SM} up to 8. The reason for the maximum frequency for low numbers of N_{SM} being smaller than the unmodified core (51 MHz) is because of the MAL circuits (Figure 4.1), which add a number of comparators to the path of the memory address bus.

It should be noted that it is the global trend in Figure 5.1 which should interest the reader the most, not each individual value. Indeed, as pointed out by Drimer [26, Chapter 5], speed measurements reported by FPGA tools exhibit a very large variability. We acknowledge this difficulty and follow the advice in [26, Section 5.2.1] by fully disclosing the source code of our implementation.

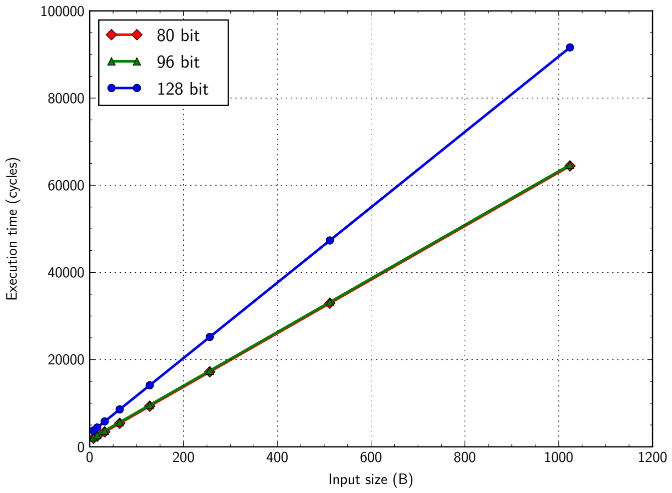

Figure 5.2: Execution time of the `encrypt` instruction for a number of security levels. The cost of the other cryptographic instructions is similar with the exception of `enable` for confidential loading, which is twice as big. Notice how the performance for the 80- and 96 bit security levels is almost equal because they use the same SPONGENT variant.

5.1.2 Microbenchmarks

To quantify the impact on performance of our extensions, we first performed microbenchmarks to measure the cost of each new instruction. To this end, we added a custom timestamp counter peripheral to the CPU core that allowed us to conveniently measure the amount of cycles passed since power up. It should also be noted that all measurements are completely noiseless and thus accurate. Consequently, it is not necessary to calculate an average value over multiple measurements.

The `get-id` and `unprotect` instructions are very fast: they both take one clock cycle. The other instructions perform cryptographic operations on their input, and hence their runtime cost depends linearly on the size of the input they handle. Remember that all cryptographic operations are implemented using the same underlying primitive (Section 4.1.2), which means their runtime cost is almost exactly the same. The only exception is the `enable` instruction when confidential loading is used. In this case the underlying primitive is called twice: once for decryption and once for key derivation. Its cost is therefore twice as big as the other instructions. Figure 5.2 shows the measurements for

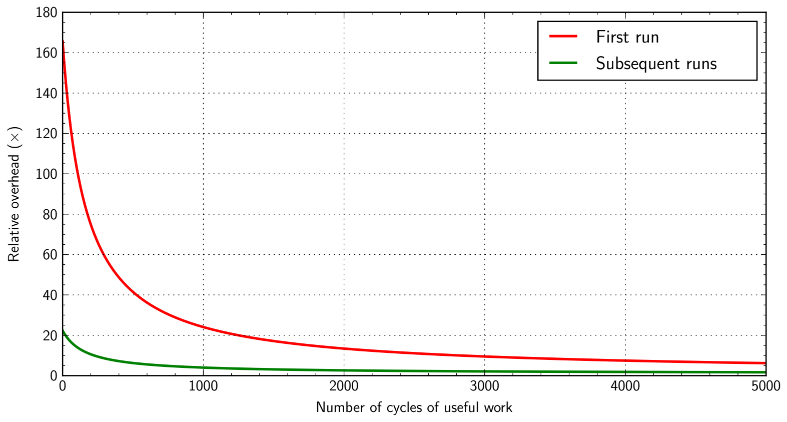

Figure 5.3: Although the absolute overhead Sancus imposes on our example setup (Figure 3.2) is quite large, the relative overhead quickly drops when the modules perform some useful work. Notice how the subsequent runs are much faster than the first due to the optimization discussed in Section 3.7.

the `encrypt` instruction for a number of different security levels.

Note that the performance of the 80- and 96 bit implementations is almost the same. The reason for this is that the performance of our crypto unit is determined by the underlying SPONGENT variant. Recall that SPONGENT is defined in different variants and we select the correct variant for the required security level (Section 4.1.2). The 80- and 96 bit security levels require the same SPONGENT variant.

5.1.3 Macrobenchmark

To give an indication of the impact on performance in real-world scenarios, we performed the following macro benchmark. We synthesized our processor with 128-bit keys and configured it as in the example shown in Figure 3.2. We measured the time it takes from the moment a request arrives at the node until the response is ready to be sent back. More specifically, the following operations are timed: (1) the request is passed, together with the nonce, to SM_i; (2) SM_i requests SM_S for sensor data; (3) SM_i performs some transformation on the received data; and (4) SM_i encrypts its output together with the nonce. The overhead introduced by Sancus is due to a call to `attest` in step (2) and a call to `encrypt` in step (4) as well as the entry and exit code introduced by the compiler. Since this overhead is fixed, the amount of computation performed in step (3) will influence the *relative overhead* of Sancus. Note that the size of the text section of M_S is 230 bytes, and that nonces and output data encrypted by M_i both have a size of 16 bits.

We measured the fixed overhead to be $26,834$ cycles for the first time data is requested from the module. Since the call to `attest` in step (2) is not needed after the initial verification (Section 3.7), we also measured the overhead of any subsequent requests, which is $3,481$ cycles. Given these values, the relative overhead can be calculated in function of the number of cycles used during the computation in step (3). The result is shown in Figure 5.3.

5.2 Area

We measured the area of our design in terms of registers and LUTs using the Map Report generated by the Xilinx tools. The unmodified openMSP430 core uses 700 registers and $2,032$ LUTs. The area of Sancus in function of the number of modules N_{SM} for a number of security levels is shown in Figure 5.4.

We also evaluated the ASIC area of our design using Synopsys Design Compiler v2013.12 with the UMC 130nm and NanGate 15nm standard-cell libraries. The default ASIC synthesis settings for the openMSP430 were used, except for disabling clock gating and DFT insertion. The unmodified openMSP430 core measures 11kGE and 15kGE, respectively, using these libraries. The results are shown in Figure 5.5.

If computational overhead is of lesser concern, the area can be reduced by computing the module key on the fly instead of storing it in registers. Exploring other improvements is left as future work.

5.3 Security

We provide an informal security argument for each of the security properties Sancus aims for (Section 2.3). First, *software module isolation* is enforced by the memory access control logic in the processor. Both the access control model as well as its implementation are sufficiently simple to have a high assurance in the correctness of the implementation. Moreover, Agten et al. [2] and Patrignani et al. [74] have shown that higher-level isolation properties (similar to isolation between Java components) can be achieved by compiling to a processor with PCBAC. Sancus does *not* protect against vulnerabilities in the implementation of a module. If a module contains buffer overflows or other memory safety related vulnerabilities, attackers can exploit them using well-known techniques [30] to get unintended access to data or functionality in the module. Dealing with such vulnerabilities is an orthogonal problem, and a wide range of countermeasures for addressing them has been developed [90].

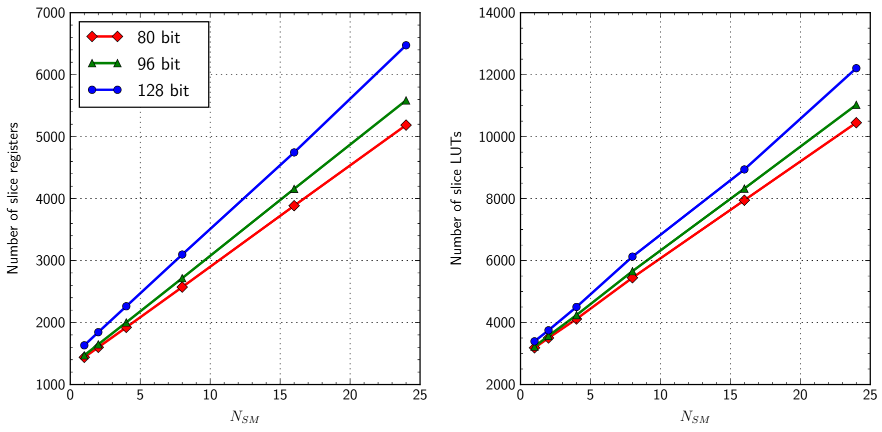

Figure 5.4: The total number of registers and LUTs in function of N_{SM} when synthesizing the openMSP430 core with Sancus extensions for different security levels.

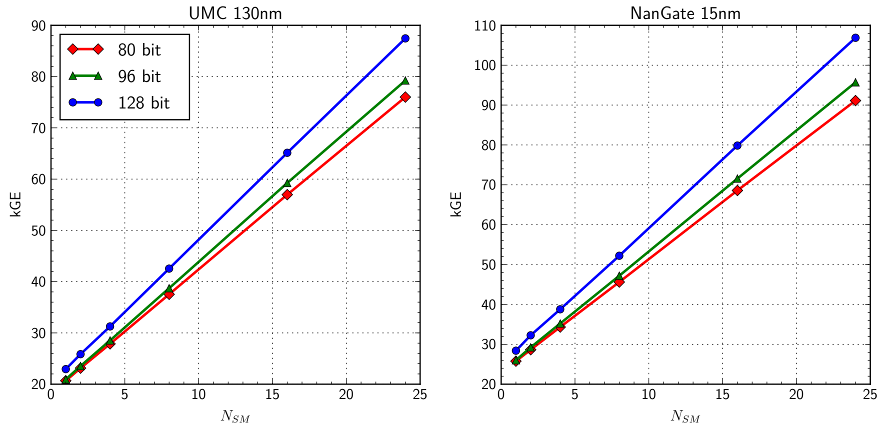

Figure 5.5: The area of the whole openMSP430 core with Sancus extensions when synthesizing for different security levels using the UMC 130nm and NanGate 15nm standard-cell libraries. Synthesis was done for a target clock frequency of 25MHz.

The security of *remote attestation* and *secure communication* follows from the following key observation: the computation of MACs with the module key is only possible by a module with the correct identity running on top of a processor configured with the correct node key (and, of course, by the software provider of the module). As a consequence, if an attacker succeeds in completing a successful attestation or communication with the software provider, he must have done it with the help of the actual module. In other words, within our attacker model, only API-level attacks against the module are possible, and it is indeed possible to develop modules that are vulnerable to such attacks, for instance if a module offers an entry point to compute MACs with its module key on arbitrary input data. But if the module developer avoids such API-level attacks, the security of Sancus against attackers conforming to our attacker model follows.

If a module has access to the correct identity of another module it wants to call, the security of *secure linking* follows from the definition of the `attest` instruction (Section 3.7). Indeed, this instruction will only succeed if a module with the given identity is enabled at the given location. This means that an attacker can only force the instruction to succeed by either (1) loading the correct module; or (2) constructing a different module with the same identity. The latter amounts to finding a hash collision, which our attacker model precludes.

The identity used for secure linking must not be stored in unprotected memory where an attacker can easily manipulate it. There are two options to provide the identity securely to a module. First, it can be stored in a module's text section. Although, if confidential loading is not used for this module, an attacker can manipulate the text section before protection is enabled, this manipulation will be detected when its provider performs remote attestation. Second, the identity can be sent using secure communication after deployment and stored in the module's data section. This is the technique that our implementation uses (Section 4.3).

The security of *confidential loading* follows from two observations. First, before the `enable` instruction is called, the module's text section is encrypted using the vendor key, which the attacker does not have access to. Second, after the instruction is finished, Sancus' access rules (Table 3.2) will deny any access to the text section from outside the module. Therefore, only API-level attacks would enable an attacker to read (parts of) the text section of modules that use confidential loading.

Finally, *hardware breach confinement* follows from the fact that we use independent master keys on all nodes (Section 3.3).

Chapter 6

Related Work

Ensuring strong isolation of code and data is a challenging problem. Many solutions have been proposed, ranging from hardware-only to software-only mechanisms, both for high-end and low-end devices.

6.1 Isolation in High-End Devices

The Multics [20] operating system marked the start of the use of protection rings to isolate less trusted software. Despite decades of research, high-end devices equipped with this feature are still being attacked successfully. More recently, research has switched to focus on the isolation of software modules with a minimal TCB by relying on recently added hardware support. McCune et al. [66] propose Flicker, a system that relies on a Trusted Platform Module (TPM) chip and trusted computing functionality of modern CPUs, to provide strong isolation of modules with a TCB of only 250 Lines Of Code (LOC). Subsequent research [65, 9, 78, 83] focuses on various techniques to reduce the number of TPM accesses and significantly increase performance, for example by taking advantage of hardware support for virtual machines.

ARM TrustZone [3] implements hardware based access control to use a physical core as two virtual processors so as to execute security critical applications in their own "world", in isolation from the normal world. The secure world runs its own OS, libraries and applications, which mutually trust each other.

More recently, Intel started shipping x86 processors equipped with Software Guard Extensions (SGX) [67] that allows the execution of security-critical code via hardware-enforced individually isolated *enclaves* in a shared address space, managed by an untrusted OS. SGX also provides functionality for local and remote attestation and for data sealing [5].

The Programmable Unit for Metadata Processing (PUMP) machine supports per-word metadata tags of arbitrary size together with software defined rules operating on these tags [24]. This allows complex policies, such as taint tracking or control-flow integrity, to be defined by software. Relevant to us is that it has also been shown to be able to implement compartmentalization of software modules [4].

The idea of deriving module specific keys from a master key using (a digest of) the module's code is also used by the On-board Credentials project [52]. They use existing hardware features to enforce the isolated execution of *credential programs* and securely store secret keys. Only one credential program can effectively be loaded at any single moment, but the concept of *families* is introduced to be able to share secrets between different programs. Although secure communication is implemented using symmetric cryptography, they rely on public key cryptography to set it up.

6.2 Isolation in Low-End Devices

While recent research results on commodity computing platforms are promising, the hardware components they rely on require energy levels that exceed what is available to many embedded devices such as pacemakers [45] and sensor nodes. A lack of strong security measures for such devices limits how they can be applied and vendors may be required to develop closed systems or leave their system vulnerable to attack.

Sensor operating systems and applications, for example, were initially compiled into a monolithic and static image without safety or security considerations, as in early versions of TinyOS [58]. The reality that sensor deployments are long-lived, and that the full set of modules and their detailed functionality is often unknown at development time, resulted in dynamic modular operating systems such as SOS [46] or Contiki [28]. As stated in the introduction of this thesis, the availability of networked modular update capability creates new threats, particularly if the software modules originate from different stakeholders and can no longer be fully trusted. Many ideas have been put forward to address the safety concerns of these shared environments, and solutions to provide memory protection, isolation, and (fair) multithreading have appeared. t-kernel [44] rewrites code on the sensor at load time. Coarse-grained memory protection (basically MMU emulation) is available for the SOS operating system by sandboxing in the Harbor system [53] through a combination of backend compile time rewriting and runtime checking on the sensor. Safe TinyOS [19] equally uses a combination of backend compile time analysis and minimal runtime error handlers to provide type and memory safety. Java's language features and the Isolate mechanism are used on the Sun SPOT embedded platform using the Squawk VM [80]. SenShare [56] provides a virtual machine for TinyOS applications. While these proposed solutions do not require any hardware modifications, they all incur a software-induced overhead. Moreover, third-party software providers must rely on the infrastructure provider to correctly rewrite modules running on the same device.

To increase security of embedded devices, Strackx, Piessens, and Preneel [84] introduced the idea of a program counter-based access control model, but without providing any implementation. Agten et al. [2] prove that isolation of code and data within such a model only relies on the vendor of the module and cannot be influenced by other modules on the same system. Eldefrawy et al. [29] implemented hardware support for allowing attestation that a module executed correctly without any interference, based on a similar access control model. While this is a significant step forward, it does not provide isolation, as sensitive data cannot be kept secret from other modules between invocations. Trustlite [51], on the other hand, features an Execution-Aware Memory Protection Unit (EA-MPU) that records program counter-based memory access rules in a configurable hardware table. Compared to Sancus, this allows for more complex policies, such as multiple private data sections per module, or protected data sharing between two or more modules. Trustlite, however, relies on a trusted Secure Loader software entity to initialise the EA-MPU table at boot time, and does not allow modules to be unloaded at runtime. More recently, the TyTAN [15] architecture extends Trustlite with dynamic loading, and local and remote attestation guarantees for isolated tasks from mutually distrusting stakeholders. Their approach to attestation resembles Sancus' in that they derive keys from task identities and a hardware-level platform key. In contrast to Sancus however, TyTAN relies on a trusted software runtime to measure task identities, and to guard inter-module authenticated communication.

6.3 Capability-Based Addressing

One of the earliest memory protection techniques is segmentation, where memory is divided in segments with associated access rights. Instead of referring to memory locations using linear addresses, a segment descriptor was used together with an offset within the segment. When addressing a memory location, the hardware would use the information from the segment descriptor to decide whether the access is allowed. The first such descriptor architecture, the Burroughs B5000 [59, Section 2.2] developed in 1961, supported different descriptor types. Data descriptors, for example, were used to provide bounds checking for array accesses and program descriptors ensured that functions could only be executed from their beginning, not unlike Sancus' entry points. Today, segmentation is still supported on the x86 architecture although it is hardly used. One notable exception is PaX [85], which uses the segmentation features of the x86 to implement non-executable pages on systems that do not support the NX-bit.

From these descriptor architectures later evolved capability-based addressing [31], starting with Dennis and Van Horn's Supervisor [22]. A capability is a protected version of a segment descriptor; one that cannot be forged by unprivileged software. This usually means that only a restricted set of operations are allowed on capabilities; e.g., reducing the referenced memory segment or the access rights is allowed. If all addressing is done through capabilities, one can restrict what memory a module can access by providing it with just the capabilities it needs. Thus, capabilities provide a way to sandbox software. Note that this is really the opposite of Sancus' isolation goals where we want to restrict what outside code can do to a module; i.e., we want to provide a reverse sandbox. This will be discussed in some more detail in Sections 9.3.2 and 9.4.3.

Most capability systems provide a way to implement protected subsystems that have access to a private set of capabilities. Although the exact mechanisms differ from system to system, protected subsystems are usually implemented through what is called an enter capability in the Plessey System 250 [59, Section 4.7]. Here, a protected subsystem is represented by a memory segment containing its code and data capabilities. An enter capability, then, refers to such a segment without giving explicit access to it. A holder of such a capability can only use it by calling it, giving the offset of one of the contained code capabilities as argument. This would cause the processor to transfer control to the beginning of the specified code segment and transform the enter capability into a read capability, giving the protected subsystem access to all capabilities stored in the segment. Note how similar this is to entry points in Sancus, although enter capabilities alone are not quite enough to implement reverse sandboxing (Section 9.4.3).

More recently, CHERI [89] introduced capabilities to modern architectures by implementing them on a 64-bit MIPS processor. Their work was based on the implementation of guarded pointers, another term for capabilities, on the M-Machine [16]. However, where the M-Machine only allowed power-of-two-sized segments to compress the size of capabilities, CHERI allows arbitrary sized segments to support true fine-grained memory protection. Another key difference is that the M-Machine is a pure capability machine while CHERI adds capability support on top of the existing memory protection features of the processor, allowing for incremental adoption of the capability model.

6.4 Secure Compilation

An interesting application of PMAs, and thus of Sancus, as that of secure, or fully abstract, compilation [1]. Roughly speaking, compilation of a program is fully abstract if an attacker interacting with the compiled program is not able to do anything he would not be able to with the program as specified in the source language.[1] A simple example of a failure of full abstraction is that of the `private` access modifier in C++ programs: at the source level, one cannot access class members marked as such but, to the best of our knowledge, this restriction is lost when compiled to machine language by any current compiler. The concept of secure compilation is interesting because it allows reasoning about the security of programs at the source level without having to worry about low-level details.

Agten et al. [2] show how to securely compile a simple object-oriented language to an architecture supporting PCBAC and provide a prototype implementation based on Fides [83]. These results were later extended to more realistic source languages by Patrignani et al. [74]. On a different architecture, Juglaret et al. [48] propose a fully abstract compilation scheme for the PUMP machine. Although the Sancus compiler (Section 4.2) is not fully abstract, its design is heavily based on these results. For example, the ideas of having a per-module call stack and clearing registers when exiting a module are taken from the suggested fully abstract compilation schemes.

[1] Although this definition is inaccurate, it will do for the present discussion.

Part II

Applications

Chapter 7

Authentic Execution of Distributed Event-Driven Applications

This chapter studies the problem of securely executing distributed applications on shared infrastructure with a small TCB. We want to provide the owner of an application running on the shared infrastructure with strong assurance that their application is executing securely. We focus on (1) *authenticity* and *integrity* properties of (2) *event-driven* distributed applications, because for this security property and class of applications, it is relatively easy to specify the exact security guarantees offered by our approach. But we believe our approach to be valuable for any kind of distributed application (event-driven or not). In particular, our prototype supports arbitrary C code for building distributed applications.

Roughly speaking, our notion of *authentic execution* is the following: if the application produces a physical output event (for instance, turns on an LED), then there must have happened a sequence of physical input events such that that sequence, when processed by the application (as specified in the high-level source code), produces that output event. Let us elaborate a bit what this means.

First, it is clear that authentic execution gives *no* availability guarantees: if the execution never produces any output, then it is vacuously secure. Extending our approach with availability guarantees is a challenging direction for future work. Second, authentic execution also specifies *no* confidentiality guarantees: the property does not prevent attackers from observing all events in the system. Our approach also offers significant confidentiality properties, but this is not the focus of this chapter. Third, authentic execution *does* provide strong integrity guarantees: it rules out both spoofed events as well as tampering with the execution of the program. Informally, if the executing program produces an output event, it could also have produced that same event if no attacker was present. Any physical output event can be explained by means of the untampered code of the application, and the actual physical input events that have happened.

The main contribution of this chapter is the design, implementation and evaluation of an approach for the authentic execution of event-driven distributed applications with a small TCB and under a strong attacker model. More specifically, the contributions are:

- The design of an approach for authentic execution of event-driven programs under the assumption that the execution infrastructure offers specific security primitives (essentially standard protected modules plus support for secure I/O).
- A novel technique for implementing such support for secure I/O by means of protected driver modules on small microprocessors like the MSP430.
- A prototype implementation of the approach on Sancus where all security primitives are implemented by hardware only, and hence a distributed application can execute authentically with a hardware-only TCB.
- An evaluation of the performance and security aspects of that implementation.

The rest of this chapter is structured as follows. First, we introduce the kind of applications we support with our approach together with a running example and our security objectives (Section 7.1). Then we describe the design of our approach together with the assumptions made of the underlying architecture (Section 7.2). This section is meant as a blueprint to be able to implement our design on a PMA. We then show how we implemented this design on Sancus (Section 7.3) followed by an evaluation of this implementation (Section 7.4). We conclude by discussing some possible directions for future work (Section 7.5).

7.1 Running Example, Infrastructure & Objectives

Figure 7.1 illustrates a simple example of the kind of system we consider. The figure shows a sensor network on a parking lot with two parking spots. This infrastructure can be reused for a variety of applications which can be provided by different stakeholders. Applications include parking guidance, parking lot utilization analysis, or detection of cars that violate parking rules. We show two of these applications: one (A_{Vio}) that detects and displays parking violations, and another (A_{Avl}) that displays the number of available parking spots. Their

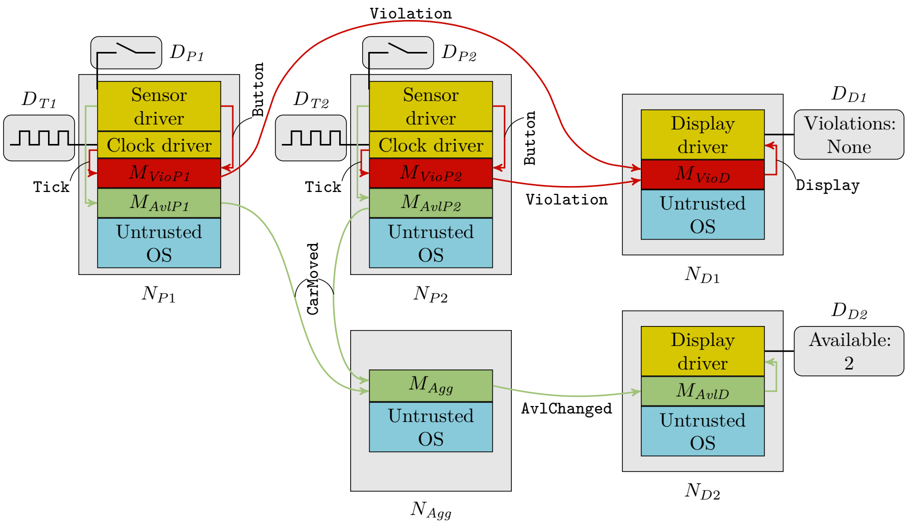

Figure 7.1: A simple example system with two applications, A_{Avl} (green) and A_{Vio} (red). Gray parts (i.e., the hardware) are trusted by all applications. The OS (blue) as well as the interconnecting network are completely untrusted. As an example, the A_{Vio} deployer creates the three red software modules with a trusted compiler (source code in Figure 7.2a), inspects the correctness of the shared parking sensor-, clock- and display drivers (yellow) and sets-up connections between the modules. Using our techniques, the deployer can be assured of authentic execution of the A_{Vio} application.

source code is shown in Figure 7.2. We now describe the different aspects more precisely, using Figure 7.1 as a running example.

7.1.1 The Shared Infrastructure

The infrastructure is a collection of *nodes* (N_i), where each node consists of a processor, memory, and a number of *I/O devices* (D_i). The infrastructure is shared among a number of mutually distrusting stakeholders that deploy and execute distributed *applications* (A_i). For simplicity, we assume processors are simple microprocessors such as the MSP430 used in our prototype.

An I/O device interfaces the processor with the physical world and supports (1) sensing some physical quantity (for instance the state of a switch), (2) influencing some physical quantity (such as an LED), and (3) notifying the processor of some state change (e.g., a key being pressed) by issuing an interrupt.

In our running example, there are 5 nodes. Two of these (N_{P1} and N_{P2}) are each attached to two input devices (a clock D_{Ti} and a car presence detector

```
module VioP1
on Button(pressed):
  if pressed:
    taken = 1
  else:
    taken = 0
    count = 0
    Violation(0)

on Tick():
  if taken:
    count += 1
  if count > MAX:
    Violation(1)

module VioP2
# Similar to VioP1

module VioD
on Violation1(v):
  v1 = v
  if v1: Display(1)
  if v2: Display(2)

on Violation2(v):
  # Similar to
  # Violation1
```

(a) A_{Vio}

```
module AvlP1
on Button(pressed):
  CarMoved(pressed)


module AvlP2
# Similar to AvlP1


module Agg
on CarMoved1(entered):
  p1 = entered
  num_avl = NUM_PARKINGS
  if (p1):
    num_avl -= 1
  if (p2):
    num_avl -= 1
  AvlChanged(num_avl)


on CarMoved2(entered):
  # Similar to CarMoved1


module AvlD
on AvlChanged(num_avl):
  Display(num_avl)
```

(b) A_{Avl}

Figure 7.2: Source code of the example applications (Figure 7.1) in a Python-like syntax. Modules are declared using the `module` keyword and last until the next module or the end of the file. The `on` keyword starts an event handler which can be connected, using the deployment descriptor, to an output event of another module or to a physical I/O channel (Section 7.1.2). Output channels are implicitly declared when invoked through a function call-like syntax.

D_{Pi}) and are installed on parking spots. Two other nodes (N_{D1} and N_{D2}) are connected to display devices (D_{Di}) and show the output of the applications. One node (N_{Agg}) is not connected to any I/O device and can be used by applications to perform general purpose computation (e.g., aggregating data from multiple sensor nodes).

7.1.2 Modules & Applications

Since we use an event-driven application model, *modules* (M_i) contain input- and output channels. Upon reception of an event on an input channel, the corresponding event handler is executed atomically and this may result in new events on the module's output channels.

An application, then, is a collection of modules together with a *deployment descriptor*. This descriptor specifies on which nodes the modules should be installed as well as how all the module's channels should be connected. Channels can be connected in two ways. First, one module's output channel can be connected to another's input channel. Such a connection behaves as a buffered queue of events. Second, the infrastructure can provide a number of *physical* I/O channels which can be connected to a module's I/O channels. The infrastructure must ensure that the events on such channels correspond to physical events. For example, an event received on a physical input could correspond to a button press or an event produced on a physical output could turn on an LED. A key contribution of this chapter is a way to securely connect modules to physical I/O channels (Section 7.3.1).

To elaborate on one of the example applications (Figure 7.1), A_{Vio} consists of three modules: two (M_{VioP1} and M_{VioP2}) are deployed on parking spots and detect single violations and one (M_{VioD}) aggregates and displays all violations. The two parking spot modules have two inputs that are connected to input devices provided by the infrastructure: one that produces events for cars entering and exiting the parking spot (D_{Pi}) and another that sends periodical timer events (D_{Ti}). As the source code (Figure 7.2a) shows, these modules wait for a car enter event, then for a maximum number of timer events and then produce an output event indicating a violation. These output events are connected to the inputs of M_{VioD} which in turn produces output events for all violations and sends them to the output display device D_{D1}.

7.1.3 Security Objective

We are now in position to state the security objective of this chapter. We want to provide security guarantees to the *deployer* of an application. The deployer uses his own (trusted) computing infrastructure to compile the application A and to deploy the compiled modules to the nodes in the shared infrastructure, and to configure connections between modules, and between modules and physical input and output channels.

At runtime, an actual trace of physical I/O events will happen, and the deployer can observe an actual sequence of physical output events. We say that this sequence of outputs is *authentic* for a deployed application A if it is allowed by A's modules and deployment descriptor in response to the actual trace of input events. In other words: the source code of A explains the physical outputs on the basis of actual physical inputs that have happened.

For instance for A_{Vio}, suppose we have physical events where a car arrives on parking 1, `MAX` clock ticks pass and then the display shows a 1. The trace of outputs is an authentic trace for A_{Vio}, because its source code allows for this display event given the physical input events. A trace for the same sequence of physical events, but now ending with the display showing a 2, is *not* authentic.

Our objective is to design a deployment algorithm such that the deployer can efficiently check authenticity of traces. If the deployer observes a trace of physical output events, and the authenticity check of the deployer succeeds, then our approach guarantees that this trace of output events is authentic.

Note that this security notion rules out a wide range of attacks, including attacks where event transmissions on the network are spoofed or reordered, and attacks where malicious software that tampers with the execution of modules is injected. Other relevant attacks are *not* covered by this security objective. As discussed earlier, there are no availability guarantees – e.g., the attacker can suppress all network communication. There are also no confidentiality guarantees: the attacker is not prevented from observing events flowing in the system. However, although this is not the focus of our design, our implementation *does* come with substantial protection of the confidentiality of the application's state as well as the information in events (Section 7.5.1).

7.2 Design of Authentic Execution

In this section we explain how to accomplish the goals described in Section 7.1.3. We start by giving an overview of what we expect of the underlying architecture in terms of security properties and features. Then we show how these features can be used to meet our security goals.

7.2.1 Underlying Architecture

Given the shared nature of the infrastructure assumed in our system model (Section 7.1.1), we require the ability to isolate source modules from other code running on a node. Since an important non-functional goal is to minimize the TCB, relying on a classical omnipotent kernel to provide isolation is ruled out. Therefore, we assume the underlying architecture is a PMA [82].

Although the details may differ between PMAs, isolation of a software module is generally understood to include the following two properties. First, a module should be able to specify memory locations containing data that is only accessible by the module's own code (*data isolation*). Second, the code of a module should not be writable and a module should be able to specify a number of *entry points* through which its code must be executed (*code isolation*). For simplicity we will further assume that both a module's code and data are located in contiguous memory areas called, respectively, its *code section* and its *data section*.

We also expect the availability of a compiler that targets PMs on the underlying architecture. The input to this compiler is as follows: (1) a list of entry point functions; (2) a list of non-entry functions; (3) a list of variables that should be allocated in the data section; and (4) a list of constants that should be allocated in the code section. The output of the compiler should be a PM suitable for isolation on the underlying architecture.

Besides isolation, we expect the PMA to provide a way to *attest* the correct isolation of a PM. Attestation provides proof that a PM with a certain identity has been isolated on the node, where the *identity* of a PM should give the deployer assurance that this PM will behave as the corresponding source module.

We also expect that, after enabling isolation, the PMA is capable of establishing a confidential, integrity protected and authenticated communication channel between a PM and its deployer. Although the details of how this works may differ from one PMA to another, for simplicity we assume the PMA establishes a shared secret between a PM and its deployer and provides an authenticated encryption primitive. We will refer to this shared secret as the *module key*. Note that the authentication property of the communication channel refers to a PM's identity and hence to attestation. Indeed, the PMA should ensure that if a deployer receives a message created with a module key, it can only have been created by the corresponding, correctly isolated, PM.

7.2.2 Mapping source modules to PMs

To map a source module to a PM, we use the following procedure. First, each of the source module's inputs and outputs is assigned a unique *connection identifier*. The format of this identifier is unimportant as long as it uniquely specifies a particular input or output.

A table (`KeyTable`) is added to the PM's variables that maps connection identifiers to symmetric keys. Using this table, every connection has one key associated with it. These keys will be initialized to all zeros by the underlying architecture which is interpreted as an unconnected input or output. For establishing a connection, an entry point is generated (`SetKey`). This entry point takes a connection identifier and a key – encrypted using the module key – as input and updates the corresponding mapping in `KeyTable` if it is not already set (Figure 7.3).

```
def SetKey(payload):
  try:
    conn_id, key = Decrypt(payload)
    if KeyTable[conn_id] == 0:
      KeyTable[conn_id] = key
  except: pass
```

Figure 7.3: Pseudocode of the `SetKey` entry point using a Python-like syntax. Note that `Decrypt` uses the module key to decrypt the payload and throws an exception if the operation failed (i.e., the payload's MAC is incorrect).

Since every connection needs to be protected from reordering and replay attacks, the compiler adds another table (`NonceTable`) to the PM's variables. This table maps connection identifiers to the current nonce for each connection. How nonces are used is explained below.

All the module's event handlers are marked as non-entry functions. A callback table (`CbTable`) is added to the PM's constants that maps connection identifiers of inputs to the corresponding event handlers. This table is used by the entry point `HandleInput`, which is called whenever an event is delivered to the PM (Section 7.2.3). `HandleInput` takes two arguments: a plain-text connection identifier and an encrypted payload. If `KeyTable` has a key for the given identifier it is used to decrypt the payload (using the *expected* nonce as associated data), which is then passed to the callback function stored in `CbTable`. If any of these operations fails, the input event is ignored (Figure 7.4). This means that, from a programmer's point of view, an input callback is only called with an event that was generated by an entity that has access to the correct connection key.

Each call to an output is replaced by a call to the new non-entry wrapper function `HandleOutput`. This function takes a connection identifier and a payload, encrypts the payload together with the current connection nonce (which is incremented afterwards) using the corresponding connection key and publishes it to the event manager (via `HandleLocalEvent`, Section 7.2.3), passing it the connection identifier. If the output is currently unconnected, the output event will be dropped (Figure 7.5).

To summarize, the following PM definition will be given as input to the PMA compiler (Section 7.2.1): (1) `SetKey` and `HandleInput` as entry points; (2) each input event handler and `HandleOutput` as non-entry functions; (3) `KeyTable`, `NonceTable`, and module globals as variables; and (4) `CbTable` and module constants as constants. Figure 7.6 shows the compiled memory layout of one of the example modules.

```
def HandleInput(conn_id, payload):
  try:
    key = KeyTable[conn_id]
    if key != 0:
      cb = CbTable[conn_id]
      nonce = NonceTable[conn_id]
      cb(Decrypt(nonce, payload, key))
      NonceTable[conn_id] += 1
  except: pass
```

Figure 7.4: Pseudocode of the `HandleInput` entry point. Note that erroneous accesses to the tables, as well as errors during `Decrypt`, cause exceptions. This means that such events, as well as those for which no input key has been set, are ignored. Notice the use of `Decrypt` here: it takes a key and the expected associated data as arguments.

```
def HandleOutput(conn_id, data):
  key = KeyTable[conn_id]
  if key != 0:
    nonce = NonceTable[conn_id]
    NonceTable[conn_id] += 1
    payload = Encrypt(nonce, data, key)
    HandleLocalEvent(conn_id, payload)
```

Figure 7.5: Pseudocode of the generated output wrapper. Note that since the compiler generates calls to this function and it cannot be called from outside the module, the connection identifier is always valid and no error checking is necessary.

7.2.3 Untrusted Software on the Nodes

To support the deployment of modules and the exchange of events between modules, a number of untrusted software components need to be installed on the nodes. This section given an overview of these components.

Module Loader

The module loader is an untrusted software component running on every node. It provides services for external entities to interact with modules on its node. To this end, it listens for two types of remote requests: `LoadModule` and `CallEntry`. The former takes a compiled PM as input, loads it into the PMA and returns the

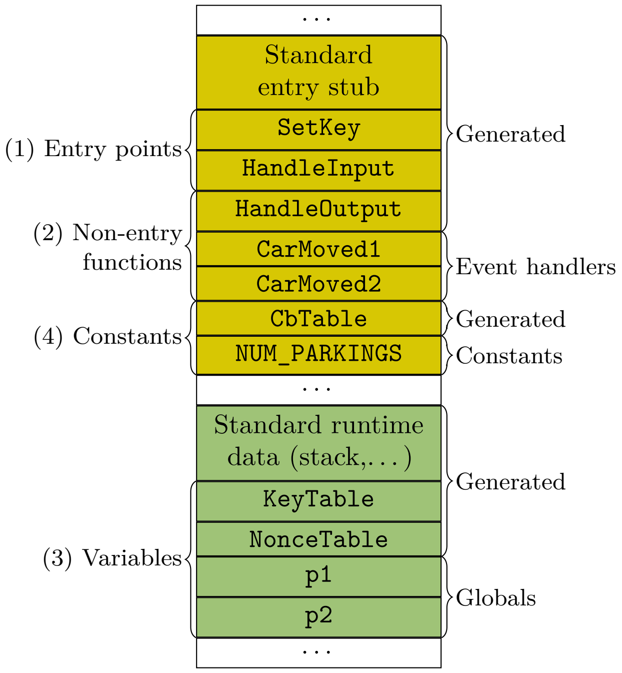

Figure 7.6: Memory layout of the compiled version of M_{Agg} of A_{Avl} (Figure 7.2b). The code- and data sections are shaded in yellow and green, respectively. The numbers on the left labels correspond to the compiler inputs (Section 7.2.1) while the right labels indicate whether parts are implicitly generated by the compiler or correspond to source code.

module's unique identifier together with all information necessary for attestation and module key establishment. (What exactly this information is and how the attestation and key establishment is performed is specific to the used PMA.) `CallEntry` takes a PM's identifier, the identifier of an entry point and potentially some arguments and calls the entry point with the given arguments.

It should be stressed that the module loader is *completely* untrusted. Indeed, a loaded module can be verified using remote attestation and an entry can be called securely by using the secure communication channel provided by the PMA.

Event Manager

The event manager is another untrusted software component running on every node that is used to route events from outputs to inputs. It recognizes three types of requests: `AddConnection`, `HandleLocalEvent` and `HandleRemoteEvent`. A deployer can invoke `AddConnection` remotely to connect the output of a module to the input of another. How exactly inputs and outputs are identified is implementation specific but it will in some form involve specifying (1) a node address (e.g., an IP address); (2) a PM identifier; and (3) a connection identifier. As will become clear later, `AddConnection` only needs to be called on the event manager of the node where the output source module is deployed.

`HandleLocalEvent` is used by modules that want to publish an event; i.e., inside the output wrappers (Section 7.2.2). The arguments are the module- and connection identifiers and the event payload. Based on the identifiers the event manager looks up the destination event manager and invokes its `HandleRemoteEvent` API, providing the identifiers of the input to which the request should be routed. For a `HandleRemoteEvent` request, the event manager will check if the destination module exists and, if so, invoke its `HandleInput` entry point, passing the connection identifier and payload as arguments.

7.2.4 Physical Input and Output Channels

We assume that the infrastructure offers physical input and output channels using *protected driver modules* that translate application events into physical events and vice versa. For input channels, these modules should generate events that correspond to physical events and provide a way for application modules to authenticate the generated events. For output channels, a driver module (M_D) should have exclusive access to its device (D) and allow an application module (M_A) to take exclusive access over the driver. That is, the driver will only accept events – and hence translate them to physical events – from the application module currently connected to it. The infrastructure should also provide a way for the deployer of M_A to attest that it has exclusive access to M_D and that M_D also has exclusive access to D. Moreover, the deployer should be able to attest M_D itself to ensure that it indeed only accepts events from the module currently having exclusive access and that it does not release this exclusive access without being asked to do so by the module itself.

7.2.5 Deployment

Deployment is the act of installing all application modules on their respective nodes and setting up the connections between outputs and inputs. All computations described in this section are run on the deployer's infrastructure and are therefore completely trusted. Of course, any requests to the nodes are performed over an untrusted network.

In phase 1, the deployer starts by compiling each source module (Section 7.2.2) into a loadable image. Then, it uses the deployment descriptor to find the node on which the module should be deployed and sends its module loader a

`LoadModule` request with the image as argument. It then performs the PMA-specific method of attestation and setting up the module key. At the end of this step, the deployer has a secure communication channel with each of its deployed source modules.

To complete phase 1 of the deployment, the deployer sets up the connections between modules (not yet the connections to physical I/O channels). To this end, it will generate a unique connection key and send it to both endpoints of the connection. Sending the key to a PM is done by first encrypting it together with the connection identifier (Section 7.2.2) using the module key. This payload is then passed to the `SetKey` entry point using the module loader's `CallEntry` API.

Next (phase 2) the deployer sets up the connections to the physical I/O channels. The deployer first sets up the connection to physical outputs (phase 2a). This is the point in time from which we know that outputs will be authentic (Section 7.2.6). Finally, all connections to physical inputs are set up (phase 2b).

7.2.6 Security Argument

We now present a security argument for the construction proposed in this section. Recall that, informally, our goal is to ensure that all physical output events can be explained by the application's source code and the observed physical input events (Section 7.1.3).

More precisely, we show the following: *Consider the time frame starting at the end of phase 2a of deployment of the application (Section 7.2.5), and ending at a point where the deployer starts a new attestation of a specific protected driver module for an output device D_O. If this attestation succeeds, and if the deployer has observed a specific sequence of physical output events on D_O in the considered time frame, then there have been contiguous sequences of physical input events on the input devices connected to the application such that the observed outputs follow from these inputs according to the application source code semantics.*

As an example, consider A_{Vio} (Figure 7.2a). If there appears a "1" on the display output device after the application has been deployed, and if after appearance of that "1" attestation confirms that the driver of the display is still in the expected state, then there must have been physical input events of a car arriving on parking spot 1 and `MAX` clock ticks.

We argue here that these output events can only be produced by the application's protected modules; the assumption of a correct compiler then leads to the desired property.

Since (1) a physical output event can only be produced by the corresponding device (D_O); (2) output drivers have exclusive access to their device; and (3) a protected module (M_O) has exclusive access to the driver (Section 7.2.4); only M_O can initiate physical output events on D_O. The deployer's successful attestation of the output driver module after the outputs have been observed ensures that this exclusive access was maintained over the entire considered time frame.

The construction of PMs (Section 7.2.2) ensures that a module can only be invoked through its two entry points. Of these two, only `HandleInput` can result in output events (Figures 7.3 and 7.4). Since `HandleInput` authenticates its input, output events are always the result of correct input events. Since our deployment scheme only allows for two types of correct input events, physical input events and outputs from other modules, our security property follows.

7.3 Implementation

We have created a fully functional prototype of our design based on Sancus. We provide both the necessary compiler extensions to compile source modules to Sancus PMs and the runtime components to deploy full applications on Contiki [27] based networks. A novel contribution of this implementation are *driver PMs* that facilitate secure I/O.

7.3.1 Secure I/O on Sancus

This section describes how protected drivers can be implemented using Sancus. Recall that for output channels, we want an application module to have exclusive access to a driver (Section 7.2.4). This, in turn, implies that the driver should have exclusive access to the physical I/O device. Although for input channels the requirements are less strict – we only need to authenticate a device – for simplicity, we also use exclusive device access here.

Exclusive Access to Device Registers

Sancus, being based on the MSP430 architecture, uses Memory-Mapped I/O (MMIO) to communicate with devices. Therefore, providing exclusive access to device registers is supported out of the box by mapping the driver's private section over the device's MMIO region. There is one difficulty, however, caused by the private section of Sancus modules being contiguous and the MSP430 having a fixed MMIO region (i.e., the addresses used for MMIO cannot be remapped). Thus, a Sancus module can use its private section either for MMIO or for data but not for both. Therefore, a module using MMIO cannot use *any* memory, including a stack, severely limiting the functionality this module can implement.

The obvious solution is to add a second private section to Sancus modules which can be used for MMIO. However, since this would mean that two new registers should be added to the hardware representation of every Sancus module

(Section 4.1.1), even to those that do not need to perform I/O, we have chosen a different approach.

Driver modules can be split in two separate modules, one performing only MMIO (`mod-mmio`) and one using the API provided by the former module to implement the driver logic (`mod-driver`). The task of `mod-mmio` is straightforward: for each available MMIO location it implements entry points for reading and/or writing this location and ignores calls to those entry points by modules other than `mod-driver`. This task is simple enough to be implemented using only registers for data storage, negating the need for an extra data section.

This technique lets us implement exclusive access to device registers on Sancus without changing its underlying hardware representation of modules. However, this incurs a non-negligible performance impact because `mod-mmio` has to attest `mod-driver` on *every* call to one of its entry points. The optimization described earlier (Section 3.7), doing the attestation once and only checking the module identifier on subsequent calls, is not applicable because it requires memory for storing the identifier. We address this by hard coding the *expected* identifier of `mod-driver` in the code section of `mod-mmio`. During initialization, `mod-driver` will check if it was assigned the expected identifier and abort otherwise. At this point, `mod-driver` also attests `mod-mmio` to verify that module's integrity and to ensure that the device's MMIO registers are mapped within `mod-mmio`'s data section. If the attestation fails, `mod-driver` aborts as well. This procedure ensures that, after an application module successfully attested `mod-driver`, it either has exclusive access to its device, or it is aborted. How this is used by applications is shown later (Section 7.3.1).

Secure Interrupts

On the MSP430, interrupt handlers are registered by writing their address to a specific memory location called the interrupt vector. Therefore, handling interrupts inside PMs is supported out of the box by registering the module's entry point as an interrupt handler. However, if the PM also wants to support "normal" entry points, we would need a way to detect when the entry point is called in response to an interrupt.

More generally, we need a way to identify *which* interrupt caused an interrupt handler to be executed. Indeed, otherwise an attacker might be able to inject events into an application by spoofing calls to an interrupt handler. To this end, we extended the technique used for caller authentication (Section 3.7). Whenever an interrupt occurs, the processor stores a special value specific to that interrupt in the register that keeps track of the previously executing module. This way, an interrupt handler can identify by which interrupt it was called in the same way modules can identify which module called one of their entry points. Note that the processor obviously ensures that these special values used to identify interrupts are never assigned to any PM.

Interfacing with Applications

There are a number of possible ways to interface protected driver modules with application modules. This section discusses one possibility that uses Sancus' secure linking feature (Section 3.7) for efficiency. The downside of this approach is that the application module has to be deployed on the same node as the driver. Note that for simplicity this section discusses drivers for single physical events but the described techniques can easily be extended to drivers supporting multiple physical events.

Input drivers provide an entry point to register a callback function to be called whenever a physical event happens (`RegisterInputCb`). During the deployment phase, application modules call this entry point – using Sancus' secure linking feature – to register one of their entry points as a callback. The driver's identifier, which is the result of a successful secure linking step, is stored in the module's private data section. Whenever the module's callback entry point is called, this identifier is compared with the result of the `get-caller-id` instruction to verify it was called by the expected driver.

Output drivers provide an entry point that allows modules to gain exclusive access (`AcquireOutput`). When called, this entry point checks if some module already has exclusive access and, if not, uses `get-caller-id` to store the identifier of the requesting module. It also offers an entry point for posting events which will check, again using `get-caller-id`, if the module posting the event has exclusive access. During deployment, an application module firsts attests the output driver, storing its module identifier, and then calls `AcquireOutput`, aborting on failure. For attestation, the application modules provides an entry point for the deployer that attests that the module has exclusive access. This is implemented by comparing the driver module's current module identifier with that of the module located at the location where the driver module was loaded at the time of deployment. Note that this implements the attestation referred to in our informal security argument (Section 7.2.6).

The reason this attestation procedure is secure is as follows. When an application module (M_A) attests a driver module (M_D) during deployment, M_A checks the correctness of M_D's code. This encompasses, among other things, that the code only allows a single module to have access to the driver, and that it does not release this access without the module having exclusive access asking for it. If M_A records the module identifier of M_D after having attested it, M_A can later check if M_D still exists by simply checking the identifier of the PM loaded at the location where M_D was loaded during deployment. This works because Sancus ensures module identifiers are unique within a boot cycle (Section 3.7). If M_A calls `AcquireOutput` on M_D and it succeeds, and later it verifies that M_D still exists, M_A can be sure it still has exclusive access to M_D. Note that this procedure also ensures that M_D has exclusive access to its underlying device.

```
SM_OUTPUT(Violation);
SM_DATA int taken, count;
SM_INPUT(Button, data, len) {
 if (data[0]) {
   taken = 1;
 } else {
   taken = count = 0;
   char event = 0;
   Violation(&event, sizeof(event));
 }
}
SM_INPUT(Tick, data, len) {
 if (taken && ++count > MAX) {
   char event = 1;
   Violation(&event, sizeof(event));
 }
}
```

Figure 7.7: Possible translation of module M_{VioP1} (Figures 7.1 and 7.2a) to C using the annotations understood by our compiler.

Indeed, because of M_D's initialization sequence (Section 7.3.1), it would have aborted otherwise, in which case `AcquireOutput` always fails.

7.3.2 Compiler and Untrusted Runtime

Our compiler implementation is a literal translation of the design outlined in Section 7.2.2. All modifications to the Sancus compiler (Section 4.2) are *extensions* meaning that all original Sancus features are still available to programmers (e.g., calling external functions or other PMs).

On top of the existing annotations provided by the Sancus compiler for specifying entry points (`SM_ENTRY`), internal functions (`SM_FUNC`) and private data (`SM_DATA`), we added two new annotations: `SM_INPUT` and `SM_OUTPUT` for specifying inputs and outputs respectively. Figure 7.7 shows an example module written in C using our annotations.

`SM_OUTPUT` expects a name as argument (more specifically, a valid C identifier). For every output, the compiler generates a function with the following signature: `void name(char* data, size_t len)`. This function can be called to produce an output event. For input handlers, `SM_INPUT` generates a function with the same signature as above. Inside this function, the programmer has access to a buffer containing the (unwrapped) payload of the event that caused its

Table 7.1: Size of the software for running the evaluation scenario. The shaded components are part of the TCB.

Component	LOC	Binary size (B)
Contiki	38386	14880
Event manager	598	1730
Module loader	906	1959
Buttons Driver	338	1016
LCD Driver	137	640
Parking Sensor	43	1383
Aggregator	84	1970
Display	31	1333
Deployment Descriptor	57	n/a

executions. For both inputs and outputs, the name provided to the annotation is the identifier used in the deployment descriptor.

The untrusted runtime consists of two components deployed on every node: the module loader and the event manager (Section 7.2.3). Both components are implemented as TCP servers running as regular Contiki [27] processes. This means that our solution can be deployed alongside conventional Contiki applications.

7.4 Evaluation

To assess the runtime overhead and the size of the software TCB we have implemented and deployed A_{Avl} (Figures 7.1 and 7.2b). Each computing node is configured to provide 64 bits of security.[1] On it, we installed a Contiki kernel, our module loader and event manager, and PM drivers for the devices it needs: a button driver for the car sensors and a serial LCD driver for the display. Then the application modules are deployed.

Table 7.1 shows the sizes of the different software components deployed on nodes. As can be seen, the majority of the code running on a node – about 40k LOC – is untrusted in our model. A total of 633 LOC comprising of drivers and the actual application code is compiled to PMs and needs to be trusted, together with 57 LOC of the deployment descriptor. That is, only 1.7% of the deployed code base is part of the software TCB.

[1]Note that, although it is probably not recommended in production systems (Section 4.1.2), 64 bits of security is enough for the present evaluation. Using the information in Figure 5.2, the results in Figure 7.8 can easily be extrapolated to higher security levels.

When looking at the binary sizes of the these software components, the difference between infrastructure components (18.1 KiB) versus TCB (6.2 KiB, 25.5%) appears less prominent, which is due to a large number of conditionally compiled statements in Contiki as well as compiler generated entry points and stub code in the PMs. Nevertheless, the reduction of the TCB when using our approach is substantial, leading to a considerably reduced attack surface on each node, and – importantly – the application owner does not need to trust *any* infrastructural software if he reviews the driver modules that his application uses.

We also performed a detailed performance analysis of this example application; the results of which are shown in Figure 7.8. The sequence diagram corresponds to the A_{Avl} application but, for simplicity, shows a variant where there is a direct connection between the sensor module and the display module (i.e., the aggregator is ignored). Notice, however, that the given timing information can easily be extrapolated to the complete version of the application and, indeed, to any application.

For fast devices like the button sensors, the overhead of our secure I/O approach can be quite large: the protected driver executes about 13 times as slow as the unprotected one. However, not much effort was put in optimizing our implementation and we expect that significant performance gains are possible (e.g., the wrapper for encrypting output events uses `malloc` to create a buffer which contributes about $30\mu s$ to the total overhead). For slower devices like the serial LCD, it is clear that the relative overhead drops significantly: the protected driver executes about 7% slower than the unprotected one.

Another type of overhead generated by our approach is the increased size of events. The sequence diagram shows that an event containing 2 bytes of useful data, (the size of an integer on the MSP430) will be 6 times as large. In general, the representation of events has a constant overhead of 10 bytes: 2 for the nonce and 8 for the MAC.

Whether these overheads are acceptable will depend heavily on the application. Yet, they are reasonable in the light of the security guarantees and TCB reduction provided by our approach.

7.5 Discussion

7.5.1 Integrity versus Confidentiality

We have focused our security objective on integrity and authenticity, and an interesting question is to what extent we can also provide confidentiality guarantees. It is clear that, thanks to the isolation properties of protected modules and to the confidentiality properties of authenticated encryption, our prototype already provides substantial protection of the confidentiality of both

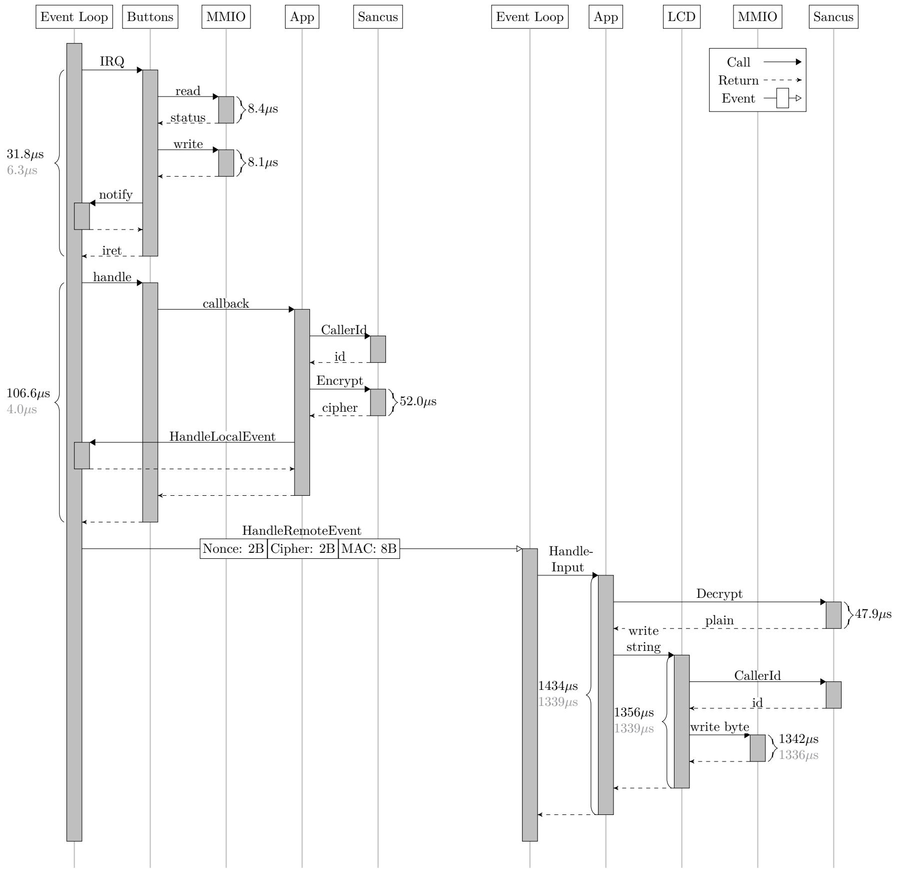

Figure 7.8: Sequence diagram showing the control flow and timings of part of the A_{Avl} application in Figure 7.1. The diagram shows how a physical input is handled starting from the moment an IRQ is generated for it by an input device up to the moment a physical output is shown on an output device. The timings for the protected application are shown in black while those for an unprotected version are shown in grey. For the lifelines, "Event Loop" corresponds to our event manager; "Buttons" and "LCD" to the `mod-driver` part of the protected drivers (Section 7.3.1); "MMIO" to the `mod-mmio` part (which is subsumed in the driver part for the unprotected application); "App" to the application modules; and "Sancus" to the Sancus core (which obviously does not exists for the unprotected application). Notice how, similar to many OSs, the "Buttons" ISR is kept as short as possible by splitting it in two levels. The first level will notify the event loop to call the second level before returning from the ISR.

the state of the application as well as the information contained in events. However, providing a formal statement of the confidentiality guarantees offered by our approach is non-trivial: some information leaks to the attacker, such as for instance when (and how often) modules send events to each other. This in turn can leak information about the internal state of modules or about the content of events. The ultimate goal would be to make compilation and deployment fully abstract [1] (indicating, roughly, that the compiled system leaks no more information than can be understood from the source code), but our current approach is clearly not fully abstract yet. Hence, we decided to focus on guaranteeing strong integrity first, and understanding exactly what confidentiality guarantees can be offered is future work.

7.5.2 Metaprogramming

The critical reader may have observed that programming many devices in our reactive programming language (or in C) will quickly get tedious. For instance, all the parking sensors will run similar code in the A_{Vio} application. In realistic deployment scenarios where thousands of devices need to be programmed, one clearly would not want to program all these devices manually. We believe our approach would integrate well with approaches for metaprogramming such applications (such as for instance the Flask approach [64]).

7.5.3 Hardware Attacks and Side-Channels

Although hardware attacks and side-channels are explicitly ruled out by our attacker model (Section 2.2), it is interesting to discuss the impact an attacker would have given access to such techniques.

An attacker that successfully circumvents the hardware protections on a node would be able to manipulate and impersonate all modules running on *that node*. That is, the attacker would be able to inject events into an application but only for those connections that originate from the compromised node. The impact on the application obviously depends on the kind of modules that run on the node. If it is an output module, the application is completely compromised since the attacker can now produce any output they want. If, on the other hand, it is one among many sensor nodes that get aggregated on another node, the impact may be minor.

Given the kind of small microprocessors that we target, many side-channels such as cache timing attacks ([54, 47]) or the “page fault channel” ([91]) are simply irrelevant to our implementation. However, analysis of our implementation in terms of side-channels is an interesting direction for future work.

7.5.4 Towards a Formal Security Proof

The main focus of this chapter is the design, implementation and evaluation of our system, and a formalization of our approach is out of scope. We believe however that a formalization of both the security property and the security argument are feasible. In this section, that can safely be skipped by readers not interested in such a formalization, we outline a path towards a formal security proof.

First, the source programming language needs to be formalized. A *source module* m declares a name, zero or more identifiers for input channels, and zero or more identifiers for output channels. For each declared input channel, an event handler to process events arriving on that input channel is defined. A program p then consists of zero or more modules, and zero or more connections, where a connection $id_1 \rightarrow id_2$ specifies that the outputs sent on output channel id_1 should be delivered to input channel id_2. Connections behave as buffered queues. The input and output channels that are not connected to anything are the *primitive* input and output channels of the application. These are the channels that will be connected to I/O devices on deployment. The semantics of programs is then relatively straightforward: execution is triggered by an event $?id(n)$ on a primitive input channel. This leads to the execution of the corresponding handler for that event. Execution of this handler generates output events, that can be (1) internal (unobservable, silent) events like internal computation within a module or transmissions over connections between modules, or (2) primitive output events $!id(n)$. On completion of the handler, a next event is handled – either another primitive input event, or one of the transmitted events buffered in one of the connections. Using standard techniques, the semantics of a program can be defined to be a labeled transition system where the labels are events α of the form $?id(n)$ and $!id(n)$. This labeled transition system defines precisely the traces of events $\overline{\alpha}$ that a source program p has. We will use the notation $\overline{\alpha} \downarrow_{id}$ to project a trace $\overline{\alpha}$ to the subtrace that only contains events on channel id; that is, events of the form $!id(n)$ if id is an output channel and $?id(n)$ if it is an input channel.

Second, the runtime infrastructure needs to be modelled, including support for loading, isolating and attesting protected modules, and also including a model of the cryptographic primitives used in our approach. It is less straightforward to build a suitable model here. Important design choices include how to model cryptography (symbolic or computational model) and at what level of detail to model the machine code of protected modules. The semantics of the runtime infrastructure will define exactly when physical input events $?pi(n)$ and physical output events $!po(n)$ occur.

Third, our approach needs to be formalized as (1) a compiler that compiles source level modules to runtime protected modules, and (2) an implementation of the deployment algorithm on the model of the runtime infrastructure. With an appropriate model of the runtime infrastructure in place, we expect this to be straightforward.

Fourth, the security property we aim to achieve needs to be formalized. This will look roughly as follows. Let $p(in_i, out_j)$ be a program with primitive input channels in_i and primitive output channels out_j. Deployment connects in_i to physical input channels pi_i and out_j to physical output channels po_j. We use the notation $\lceil\rceil$ to relate runtime physical events to the corresponding source-level event, i.e., $\lceil ?pi_i(n)\rceil = ?in_i(n)$ and $\lceil !po_j(n)\rceil = !out_j(n)$. Our security property now becomes: For the time frame starting at the end of phase 2a of deployment, and ending at a point where the deployer starts a new successful attestation of the protected driver module for po_j, let ρ_{out_j} be the sequence of output events on po_j. Then p has a trace $\overline{\alpha}$ such that (1) $\overline{\alpha}\downarrow_{out_j} = \lceil \rho_{out_j} \rceil$, and (2) for each primitive input channel in_i, there has been a contiguous sequence of inputs ρ_{in_i} on pi_i with $\overline{\alpha}\downarrow_{in_i} = \lceil \rho_{in_i} \rceil$.

Fifth and finally, the theorem needs to be proven formally. This is likely to be a substantial effort, and hence is out of scope for this thesis, but remains an interesting topic for future work.

Chapter 8

Trust Assessment Modules for the Internet of Things

In order to benefit from Sancus' security features, one must annotate the module's source code and compile it using our compiler (Section 4.2). Although we have tried to make this process as non-intrusive as possible, it might not be feasible to run every software system within a Sancus module. In some cases, for example, the source code might simply not be available to recompile. In other cases, the limit on the number of supported modules (N_{SM}, Section 4.1.1) may make it impossible to provide protection for all modules that need it.

This chapter explores a way to use Sancus to provide security guarantees for software that, itself, is not able to be protected by Sancus. By deploying so-called *trust assessment modules* alongside unaltered and unprotected code, we are able to measure certain security relevant properties in a secure way. Of course, given our strong attacker model (Section 2.2) and the fact that we do not protect the measured code, the obtained security properties are of a heuristic nature.

8.1 Trust Assessment Modules

Our approach to trust assessment is designed to integrate seamlessly with the deployment of low-cost and low-power hardware in Wireless Sensor Networks (WSNs) and in the IoT. In particular, we make use of a Sancus-enabled CPU to run a protected trust assessment module and to facilitate secure and authenticated communication with a remote operator of this module. This operator can be, for example, a human operator with a particular interest in inspecting a specific device, or a trust management system that keeps track of the integrity and trustworthiness of a larger network of devices. Our trust assessment module executes as a PM, in isolation from a base of largely unmodified and generally untrusted OS and application code. Yet, our approach partially relies on services provided by this untrusted code, e.g., networking, scheduling and memory management, in a way such that failure is detected by the trust assessment module or by the remote operator. Trust assessment modules are capable of inspecting and modifying the state of the untrusted OS and applications autonomously or on request, giving the operator a trustworthy means of assessing the integrity of the software on a node and to take actions accordingly.

In this section we describe the process of deploying and communicating with Sancus-protected trust assessment modules and discuss inspection targets and trust metrics. We further outline weaknesses and attack scenarios to our approach. While the examples in this section are given with respect to the Contiki OS and its internals, we believe that our approach can be easily adapted to support other OSs in the domain of the IoT, such as TinyOS [57] or FreeRTOS [11].

8.1.1 Module Deployment

To deploy trust assessment modules, we implemented the standard Sancus deployment strategy (Section 4.3) on Contiki. This gives the trust management system a way to attest the correct deployment of trust assessment modules as well as a secure communication channel. Figure 8.1 gives an overview of the deployment process.

After this process, the trust assessment module is now ready to execute on the computing node and may access all data and code on that node, with the exception of data belonging to other PMs. Consequentially, the module may inspect arbitrary address ranges and report its findings to the operator as an indication of the trustworthiness of the node. In the following section we discuss a number of these trust indicators in detail.

8.1.2 Trust Indicators

Our approach to trust assessment readily supports measuring a number of trust indicators as listed and explained in detail below. Importantly, our system is not limited to these indicators and we believe that additional or alternative indicators may be more suitable for specific application scenarios. Research, in particular in the context of software aging and software rejuvenation [21] names many such indicators that may be securely measured using our approach.

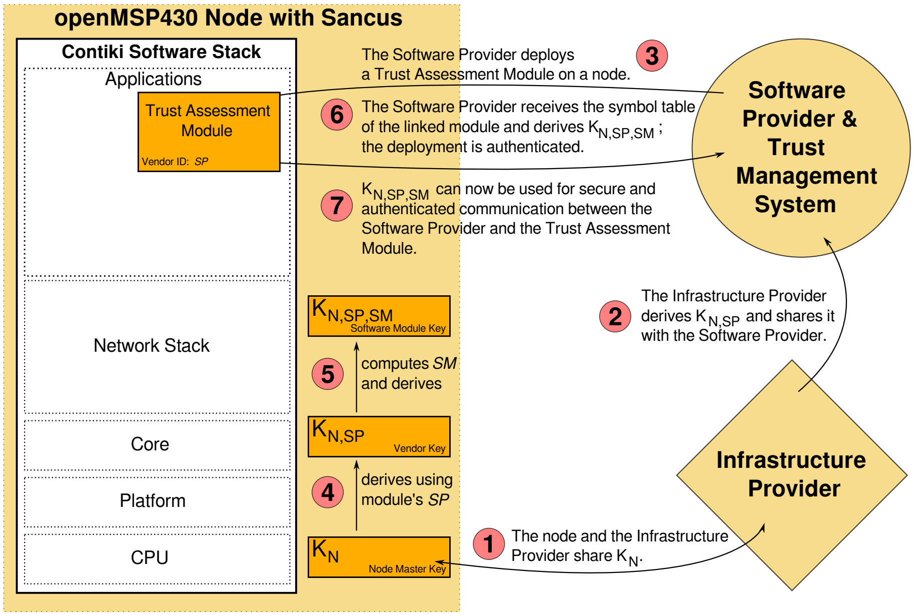
openMSP430 Node with Sancus
Contiki Software Stack
Applications
Trust Assessment Module
Vendor ID: SP
Network Stack
Core
Platform
CPU
$K_{N,SP,SM}$ Software Module Key
5 computes SM and derives
$K_{N,SP}$ Vendor Key
4 derives using module's SP
K_N Node Master Key
3 The Software Provider deploys a Trust Assessment Module on a node.
6 The Software Provider receives the symbol table of the linked module and derives $K_{N,SP,SM}$; the deployment is authenticated.
7 $K_{N,SP,SM}$ can now be used for secure and authenticated communication between the Software Provider and the Trust Assessment Module.
Software Provider & Trust Management System
2 The Infrastructure Provider derives $K_{N,SP}$ and shares it with the Software Provider.
Infrastructure Provider
1 The node and the Infrastructure Provider share K_N.

Figure 8.1: Deployment of a trust assessment module on a Sancus node. The TCB, from the perspective of the operator, is shaded in orange.

Code Integrity

A particularly useful measurement is code integrity. Sancus-enabled hardware features a cryptographic primitive to compute a MAC of a section of memory using the module key (Sections 3.1 and 3.5). This MAC may then either be reported to the remote operator or be compared with a MAC stored in the secret section of the trust assessment module in autonomous operation. Code integrity checks with a MAC are used by the trust assessment module to establish whether a particular section of code has been modified, which is then securely communicated to the operator. Unexpected code modifications may be caused by an attack against, or a malfunction of, the unprotected software on the device. Candidates for integrity checks are core functions of the OS such as the scheduler, the memory management system or the network stack, or application code. Integrity checking all code sections is technically feasible but may impose unacceptable computational overheads.

OS Data Structures

Trust assessment modules are further capable of inspecting and reporting the content of internal data structures of the OS. Interesting candidates for this are the process table or the interrupt vector table. Similar to code integrity checks, unexpected changes of these data structures are a strong indication of a malfunction or a successful attack against a device.

Available Resources

A group of indicators that is heavily used in the domain of software aging is the availability of resources such as memory and swap space: as software runs for extended periods of time, small memory leaks can accumulate and degrade performance, eventually leading to failure. In the context of Contiki and the MSP430 we use the general availability of program memory and data memory and the size of the largest available chunks of these as trust indicators. The chunk size is an important characteristic as our architecture does not feature a MMU that could mitigate the fragmenting effect of repeated allocation and deallocation. Importantly, reliably measuring the availability of program and data memory requires implementing part of the allocator, typically an OS component, as part of the trust assessment TCB.

Application Data Structures

Similar to monitoring OS data structures, we have experimented with using application data as trust indicators. For example, on WSN nodes that run a webserver, activity can be measured by monitoring the length of the request queue. Also static content that is used to compile dynamic websites can be inspected to detect modification due to a bug or a malicious attempt. Generally all these measures are highly specific with respect to critical use cases of a node.

Event Occurrence and Timing

A key feature of our trust assessment infrastructure is to monitor and attest intentional activity on a node. More specifically, by integrating part of the OS's scheduler into the TCB, our approach can attest when critical code on a node has been executed. This allows an operator to infer which parts of a node are behaving within expected parameters.

Combined Indicators

In particular in the context of autonomous operation of a trust assessment module, combining trust indicators is desired so as to automatically adapt to changing deployment scenarios. In particular, we have experimented with modules that combine the inspection of OS data structures, i.e., the process table, and periodically performing integrity checks on the functions associated with each process. This can be interleaved with measuring the frequency of process invocation and execution times, giving the operator a detailed picture of the behaviour of a computing node and allowing for specific autonomous responses to faults.

8.1.3 Fault Recovery

As a trust assessment module or the operator detect anomalies on a node, the module is even capable of responding to the situation. Responses may range from a simple reset of the node over a more thorough investigation of the fault up to actively manipulating the system state and restoring damaged code and data.

8.2 Evaluation

We have implemented the approach described in the previous section as a number of flexibly configurable trust assessment modules that can be loaded into a Contiki OS at runtime. In this section we evaluate our implementation with respect to overheads in terms of module sizes and run time. We further discuss security gains, attack scenarios and their mitigation.

8.2.1 Scenario & Implementation

Our prototypic implementation is based on a developmental version of Contiki 3.x running on Sancus. We evaluate an application scenario in which the trust assessment module regularly reports on the application processes running on a node, periodically checks the integrity of a number of code sections of these processes and integrates with Contiki's scheduler to detect and log process invocations. We have further added a public entry point to the trust assessment module that allows an application to register invariant address ranges, which are then included in periodic integrity checks. This section gives an overview of entry points and the internal behaviour of our trust assessment module and the demo scenario.

As outlined in Table 8.1, our example module provides a number of entry points to be called from unprotected code. Most importantly, the `TAMainFunc` is invoked by the scheduler. In a first run, it will initialise internal data structures of the module and then populate these data structures with initial measurements from the unprotected OS. This involves shadowing part of the scheduler's process list and calculating MACs of the process functions and the interrupt vector table. Subsequent invocations result in the current state of the unprotected OS being compared with the internal state of the module. In addition, `TASecureCallProcess` is used by the scheduler to start process functions. As this function is part of the trust assessment module, it can securely log which process is invoked and keep track of meta data. Of course, all data,

Table 8.1: Entry points of our trust assessment module.

Function Name	Description
`TAMainFunc`	Main entry point controlling initialisation and periodic behaviour.
`TARegisterInvar`	Can be used by application code and internally to register an address range for regular integrity checks.
`TASecureCallProcess`	Used by the OS scheduler to invoke application functions; the trust assessment module extends the call with counting the number of invocations and measuring time.
`TAInvarsStatus`	Returns an encrypted status report on the integrity checked address ranges.
`TAProcessStatus`	Returns an encrypted status report on the processes currently running on a node.

including MACs and meta data on process invocations is stored in the trust assessment module's private data section. The functions `TAInvarsStatus` and `TAProcessStatus` return a snapshot of this data, encrypted with the module key and using a nonce to guarantee freshness. Thus, the module's state can be reported to the operator for further assessment.

To test the effectiveness of our trust assessment module, our scenario integrates a number of trivial application processes and a "malicious" process that aims to perform alterations to OS data and application code. Specifically, our attacker is invoked by an event timer. With every invocation it performs one of the following random actions: do nothing, modify a function pointer in the process list, remove an entry from the process list, overwrite a process function, or modify an entry in the interrupt vector. Event timing typically results in alternating invocation of the attacker and the trust assessment module. Expectedly, all changes performed by the attacker are detected with the next invocation of the trust assessment module.

8.2.2 Overheads

Our evaluation shows at what expenses the alterations made by the attacker are detected. In Table 8.2 we list measurements of the size of our trust assessment components and these components' execution time. All code of the demo scenario is compiled either with MSP430-GCC 4.6.3 if no Sancus features are involved, or with the Sancus toolchain (Section 4.2). The trust assessment

Table 8.2: Size and execution time of different trust assessment components on an MSP430 running at 20 MHz: 1 cycle corresponds to 50 ns. Function sizes include protected helper functions.

Function	Size (B)	Run time (cycles)	Description
`TACoreEnable`	58	236,440	Enables module protection and initiates key generation.
`TAMainFunc`	430	578	Main function, initialisation.
		73,678	… validation run (5 processes, 9 integrity checks).
`TARegisterInvar`	402	1,242	Stores meta-data and MACs of 32 B.
		10,762	… 199 B.
		19,930	… 399 B.
`TACheckInvars`	498	69,659	Checks integrity of 9 address ranges (1833 B).
`TAAddProcess`	568	$\leq$ 18,374	Shadows an entry from the process list and determines length of process function.
`TACheckProcesses`	288	2,371	Checks shadowed process data against process list (5 processes).
`TASecureCallProcess`	392	266	Process invocation with no logging.
		$\leq$ 731	… logs time and number of invocations.
`TAInvarsStatus`	202	10,254	Encrypts meta-data on integrity-checked code and data (160 B + 16 B nonce).
`TAProcessStatus`	202	17,488	Encrypts meta-data on running processes (320 B + 16 B nonce).
Total	3,742	n/a	Code and data.

module is executed on Sancus configured with 41 KiB of program memory[1] and 16 KiB of data memory, running at 20 MHz. Synthesis is done on a Xilinx Spartan-6 FPGA. In Table 8.2 we report execution times in terms of CPU cycles. With the given clock speed, 1 cycle corresponds to 50 ns.

As can be seen from Table 8.2, our approach does imply non-negligible overheads. Whether these overheads are acceptable depends largely on the constraints on reactivity and energy consumption versus safety and security requirements in a specific deployment scenario. Our trust assessment module is designed to keep the cost of periodic validation tasks small, typically below 70,000 cycles (3.5 ms), at the expense of incurring higher initial overheads. Overall, most overheads are caused by the use of Sancus-provided cryptographic operations. The performance of these operations is evaluated in detail in Section 5.1.2.

As mentioned in Section 8.1.2 certain trust indicators, such as logging process invocations, required us to modify the Contiki core. These modifications are always small, i.e., replacing a call to a Contiki internal function with a PM-equivalent. Yet, the resulting overhead is considerably high due to switching protection domains – 26 cycles for an unprotected call and return versus 160 cycles for calling a protected entry point function. Due to passing arguments, return values, and logging the function invocation with a time stamp, process invocations through `TASecureCallProcess` incurs an overhead of 731 cycles.

With respect to runtime performance it is important to mention that Sancus does not support interruption of protected code execution. Thus, protected modules run with interrupts disabled, which may lead to important interrupts not being served by the OS and certain properties of the unprotected code potentially being broken. Examples for this could be real-time deadlines not being met due to extensive integrity checks. This issue can be mitigated by splitting up periodic validation tasks, e.g., do not perform all integrity checks but only one per scheduled invocation of the trust assessment module. Similar approaches have been used to perform expensive validation tasks in desktop and server environments [37]. Ongoing research aims to resolve this issue by making Sancus PMs fully interruptible and re-entrant. Mechanisms for securely handling interrupts in the context of PMAs have been discussed in [18, 51]. The non-interruptibility of Sancus PMs also makes it necessary to use trampoline functions that re-enable and again disable interrupts when transferring control to an application process in `TASecureCallProcess`, incurring relatively high overheads for scheduled process invocations.

We neither evaluate nor provide an integration with a trust management system. In particular, we do not evaluate the infrastructure that has to be

[1]ROM is often used as program memory in embedded devices. However, on platforms that support module deployment at runtime, as we do, program memory is writable.

in place to load a software module at runtime, and to communicate with a PM on the OS level. This infrastructure performs fairly generic tasks, yet its implementation is highly dependent on the deployment scenario. Contiki and many other embedded OSs provide module loaders and a network stack that is fully sufficient to implement the required functionality. Yet, the performance of these components depend on the storage and communication hardware connected to the CPU and is, thus, beyond the scope of this thesis.

8.2.3 Security Evaluation

Bootstrapping Autonomous Operation

An obvious issue of the scenario presented and evaluated here is with respect to the suggested autonomous mode of operation: the trust assessment module automatically discovers running processes and then periodically checks the discovered data structures and code sections for unexpected changes. Of course, an attacker may tamper with these sections at or before boot time, effectively preventing detection in regular checks. In our scenario it would be the responsibility of the operator to request and evaluate the output of `TAInvarsStatus` and `TAProcessStatus` to detect such modifications. Alternatively, a trust assessment module may also be provided with a list of expected processes and MACs by the operator at runtime, using secure communication.

Communication Failure

While the code and the internal state of the PM cannot be tampered with, it is of course possible that malfunctions or a successful attack against the node prevent the trust assessment module from successfully communicating with the operator or from executing altogether. Yet, this is detected by the operator who then may conduct actions appropriate for the deployment scenario.

Preventing Invocation of the Trust Assessment Module

In the evaluated application scenario, the trust assessment module is invoked by the scheduler and its entry point is stored in the unprotected process list. This gives the attacker process the opportunity to tamper with the pointer to the entry point, allowing it to disable execution of the trust assessment module. Alternatively, an attacker or a malfunction may disable interrupts while preventing control flow from returning to the scheduler. In our evaluation scenario this attack would not be detected by the trust assessment module directly but rather by the operator who would not be able to communicate with the module.

For autonomous operation, this attack can be easily mitigated by configuring the trust assessment module to be invoked as an interrupt service routine for a non-maskable timed interrupt using our secure I/O techniques (Section 7.3.1). Since the MSP430 does not have a non-maskable timer interrupt, we simulated this behaviour by using the watchdog as a source of timed interrupts, which we have implemented as an optional configuration option in our evaluation scenario. To ensure that the module will complete its tasks, this approach requires the worst-case execution time of the trust assessment module to be smaller than the interrupt rate. It is possible to guarantee that the watchdog configuration is not modified by an attacker by making the control register and the respective entry in the interrupt vector table part of the secret section of a PM. In combination with extensive integrity checks, this approach also hinders stealthy attacks where malicious code would attempt to restore a valid system state before the trust assessment module is executing. Yet, using a non-maskable interrupt to invoke the trust assessment infrastructure requires some consideration: It must be possible to determine the worst-case execution time of the trust assessment module and it must be acceptable to interrupt application code for that time as the PM itself is non-interruptible. Using a scheduler to invoke the trust assessment module allows for more permissible policies that prevent starvation of applications.

Attacker Adaptation

A stealthy attacker that is well-adapted to a specific trust assessment module may be able to hide code or data in address ranges that are not inspected by the module. The attacker may also restore inspected memory content to the state that is expected by the trust assessment module right before inspection takes place. Our approach to trust assessment counters these attacks by allowing the operator to deploy trust assessment modules at runtime, confronting the attacker with an unknown situation. Alternatively, a generic module may inspect targets by request from the operator rather than controlled by a deterministic built-in policy.

Process Accounting

Our trust assessment module features logging and reporting a time stamp of the latest invocation and the total number of invocations of scheduled processes. Of course, these numbers are only exact as long as processes are called through the scheduler, which passes the call through our trust assessment module. As processes may be invoked without using the scheduler, the numbers reported by our module represent a lower bound on the actual number of invocations. If more precise measures are needed for a particular process, this process should be implemented as a PM and perform its own accounting.

Extending the TCB

Of course, the safety and security of a node could be improved greatly by implementing larger parts of the OS, e.g., the scheduler, or applications as PMs. PMAs imply a number of complications that are a direct consequence of the strong isolation guarantees provided: resource sharing between components is generally prohibited, yet it is often desired for efficiently implementing communication between components. In Sancus, for example, one would have to explicitly copy the protected state of a module so as to share it with another module or unprotected code. While technically feasible, we believe that it is not trivial to re-implement a more complex code base as a set of neatly separated PMs. Alternatively one could think of compiling an entire embedded OS together with its applications as a single PM. This would ensure integrity but does not provide for isolation between components and severely restricts runtime extensibility and the use of dynamic memory. Thus, the trust assessment modules provided here present a pragmatic approach to measure and improve safety and security of an IoT node while not interfering with the existing code on that node. This results in low development overheads and runtime overheads that should be acceptable for many deployment scenarios.

8.3 Related Work

This section discusses some related work in the domains of WSNs and trust assessment on high-end systems. Where applicable, we compare the work with our contributions. Note that this section is not meant as an exhaustive exposition on trust assessment – a domain that can be interpreted rather broadly – but as an overview of the work that we consider closely related to ours.

8.3.1 Trust Management in Wireless Sensor Networks

Many schemes for trust management in WSNs have been devised by researcher over the years [34, 43, 63]. Most of these schemes deal with the problem of distributing trust management over a network of sensor nodes. Individual nodes usually obtain trust values about neighboring nodes by observing their externally visible behavior. These trust values are then propagated through the network allowing nodes to make decisions based on the trustworthiness of other nodes.

Although our approach does not deal directly with distributed networks, it can be used to enhance trust metrics used by existing trust management systems. Indeed, our trust assessment modules can provide nodes with a detailed view on the internal state of their neighbors; allowing them to make better informed decisions about their trustworthiness. Moreover, since the produced trust metrics are attested, the bar is significantly raised for existing attacks on trust management systems where malicious nodes try to impersonate good nodes.

8.3.2 Trust Assessment on Desktop & Server Systems

Techniques similar to our trust assessment modules have been described for the domain of desktop and server systems. Copilot [75] and Gibraltar [10] employ specialised PCI hardware to access OS kernel memory with negligible runtime overhead. Both systems detect and report modifications to kernel code and data.

A number of approaches use virtualisation extensions of modern general purpose CPU. Here, a hypervisor is employed to inspect a guest operating system. SecVisor [79] protects legacy OSs by ensuring that only validated code can be executed in kernel mode. Similarly, NICKLE [76], shadows physical memory in a hypervisor to store authenticated guest code. At runtime, kernel mode instructions are then only loaded from shadow memory and an attempt to execute code that is not shadowed is reported as an attack. Hello rootKitty [37] inspects guest memory from a hypervisor to detect and restore maliciously modified kernel data structures. Due to frequent transitions between execution hypervisor and guest code, and expensive address translation between those domains, these inspection systems typically incur significant performance overheads. HyperForce [36] mitigates this problem by securely injecting the inspection code into the guest and forcing guest control flow to execute this code.

Our approach to trust assessment using PMs on a Sancus-enabled TI MSP430 provides isolation guarantees that are equivalent to executing the trust assessment code in a hypervisor. Yet, our PM executes in the same address space as the OS and application, which makes expensive domain switches and address mapping unnecessary. In addition, Sancus provides attestation features in hardware that the above systems do not employ. On modern desktop architectures, these features can be implemented using the TPM, or by using SGX.

Sancus enables the implementation of effective security mechanisms on extremely light-weight and low-power hardware. In terms of inspection abilities and isolation guarantees, these mechanisms are similar to the state-of-the-art in the desktop and server domain. Our approach to trust assessment modules illustrates that, using Sancus, comprehensive inspection mechanisms can be implemented efficiently, incurring runtime overheads that should be acceptable in many deployment scenarios with stringent safety and security requirements.

8.4 Conclusion

In this chapter we presented an approach to trust assessment for extremely light-weight and low-power computing nodes as they are often used in the IoT. Instead of relying on the externally observable behaviour of a node, we deploy flexible trust assessment modules directly on the node. These modules are executing in isolation from an unprotected OS and application code. Yet, the modules are capable of inspecting the unprotected domain and report measurements that are indicative for the trustworthiness of a node to a trust management system. We employ Sancus to guarantee isolation, to facilitate remote attestation of the correct deployment of a trust assessment module, and to secure communication between a module and a trust management system. In terms of inspection abilities and isolation guarantees, Sancus-protected trust assessment modules are similar to using virtualisation technology or specialised security hardware in the desktop and server domain.

We have implemented our approach to trust assessment modules on a Sancus-enabled TI MSP430 microcontroller. Our results demonstrate that, using Sancus, comprehensive inspection mechanisms can be implemented efficiently, incurring runtime overheads that should be acceptable in many deployment scenarios with stringent requirements with respect to safety and security. Indeed, we believe that our approach enables many state-of-the-art inspection mechanisms and countermeasures against attacks to be adapted for IoT nodes and in the domain of WSNs, which are in dire need of modern security mechanisms [77]. These mechanisms include integrity checks and data structure inspection as discussed in this chapter. Yet, more complex mechanisms such as automatic invariant detection and validation [37], stack inspection [33] or protection against heap overflows [70] are in scope for our approach.

Part III

Conclusion

Chapter 9

Discussion on some Design Decisions and Future Work

We will conclude this thesis by exploring some of the design decisions made for Sancus that could have been done differently. This chapter can be used by readers interested in designing a security architecture similar to Sancus to gain insight in the different alternatives that can be used to accomplish Sancus' security guarantees.

More specifically, we will investigate how a different approach to software provider keys may lead to easier secure communication with modules belonging to a different provider (Section 9.1); how public-key cryptography could be used by Sancus and the advantages it would bring (Section 9.2); how the layout of modules can be made more flexible (Section 9.3); and how the hardware limit on the number of concurrently loaded modules can be lifted (Section 9.4).

9.1 Software Provider Keys

During early design stages, Sancus did not have the concept of software provider keys (Section 3.2.2 and Table 3.1). Instead, module keys were supposed to be generated by the infrastructure provider. Software providers would be required to request a module key – by submitting an image of the module to the infrastructure provider – before deployment.

Although this scheme would support the same security features as Sancus' current design, there is an obvious practical objection to it: software providers would not be able to deploy modules on their own. This issue led us to add another level to the key hierarchy: software provider keys only needed to be issued once by the infrastructure provider and allowed software providers to deploy modules without interaction with third parties.

9.1.1 Problems with the Current Design

Recall that module keys are derived at the time the module's protection is enabled and kept unchanged in the protected storage area in hardware from then on. The implication of this design is that every module is permanently tied to a certain software provider. Since it so natural to think of modules to always belong to a specific software provider, this implication was not given much thought during Sancus' initial design.

An issue that *was* given some thought was the difficulty for a software provider to communicate with modules that were not deployed by them. Indeed, remote attestation and secure communication (Section 3.5) works by deriving a shared secret from the software provider key making it unavailable to third-parties that do not have access to this key.

Note that there is an interesting asymmetry here with secure linking (Section 3.7) where modules belonging to different providers *are* able to attest and securely link with each other. This property can be used to provide a workaround for the secure communication issue above. If SP_i wants to securely communicate with SM_j belonging to SP_j, SP_i can deploy SM_i on the same node and let this module attest SM_j. Then, SM_i can either establish a shared key between SP_i and SM_j or act as a proxy between them. Although this scheme provides a way for software providers to securely communicate with modules not belonging to them, there are obvious downsides to it in terms of code size. Especially on the kind of resource constraint devices that Sancus targets, installing a module just for the sake of establishing a secure communication channel might be infeasible.

For completeness, it should be mentioned that there is another possible workaround: SP_i could simply ask SP_j to establish a shared secret with SM_j using their own secure communication abilities. At first sight, this might seem a reasonable solution since it does not involve the code size overhead as the scheme discussed above. However, note that SP_i looses the ability to attest the correctness of SM_j and instead has to completely trust SP_j for doing so correctly and honestly. Therefore, this workaround is strictly inferior to the previous one.

9.1.2 Multi-Vendor Protected Modules

The problem described above is caused by software providers not having a native shared secret with modules not deployed by them. This, in turn, is a consequence of the earlier observation that modules are tied to a single provider. Since this was an accidental design choice rather than a necessary one, we are now thinking of alternatives that make secure communication easier to implement for application developers.

If modules would be able to use module keys derived from another *SP* than the one they were deployed by, secure communication with different software providers would be directly supported. Therefore, we propose to add an optional argument, *sp*, to the `encrypt` and `decrypt` instructions. When provided, Sancus will first derive $K_{N,SP}$, then $K_{N,SP,SM}$, and then use that key for the operation. Since *SP* itself is also able to calculate $K_{N,SP,SM}$ given the identity of *SM*, these primitives enable the establishment of a shared key between *SM* and any *SP*.

An important question is whether these new instructions introduce any weaknesses in Sancus' security guarantees. Since we propose to give modules (restricted) access to the provider key of a provider they were not deployed by, one may wonder if this would enable this module to impersonate this provider. To see why this is not the case, note that a module *SM* only has access to key $K_{N,SP}$ to derive the key $K_{N,SP,SM}$ with its own identity. Since *SP* would only be using $K_{N,SP,SM}$ if they wanted to communicate with a module with identity *SM* on node N, no malicious impersonation can be performed.

A last interesting question is whether modules should still somehow be bound to the provider that deployed them. If we would make the *sp* argument to the `encrypt` and `decrypt` instruction mandatory, there would be no need for modules to be bound to any particular provider. If provider *SP* needs to communicate with *SM*, they would simply encrypt their message using $K_{N,SP,SM}$ putting *SP* in the associated data (Section 3.1). This way, *SM* is able to decrypt the message with the correct key.

However, in many cases module sharing would not be desirable and a provider wants to have exclusive access to its modules. In other cases, sharing may be desirable but the provider still wants to have ultimate control over its modules; e.g., to decide when to stop sharing its functionalities. If modules were not bound to any specific provider, these use cases would be impossible to implement. Indeed, all providers would be considered completely equal by the module. Therefore, binding modules to the provider that deployed them is an essential feature to have. Indiscriminate sharing of a module can then easily be implemented by ignoring this binding.

9.1.3 Module Identity and its Main Vendor

The previous section raises an interesting question: If a module is bound to its deploying software provider and hence has a special relation to this provider, should this somehow be reflected in its identity? Due to the significance that this special relation may have for a module's behavior, it should definitely be possible to discover which provider deployed a module. For the deploying provider, this can be accomplished by making the module use its default key during the attestation procedure since this key should be derived from the provider's key. For other providers, making the deploying provider's identity part of a module's identity would be a way to attest this.

However, in some cases it might not matter which provider deployed a module. For those cases making the deploying provider's identity part of the module's identity would make the remote attestation procedure more cumbersome for other providers. Indeed, the provider would first need to ask the module for its deploying provider in order to be able to calculate the module key before the remote attestation procedure can start. Therefore we propose to not add the deploying provider's identity to the module's identity. When this information *is* important, the module can simply add it to the initial remote attestation message.

9.1.4 Remote Attestation Procedure

To summarize, let us show the messages exchanged during the attestation procedure in both cases. Here, $A \rightarrow B : \{P\}$ means A sends a message with payload P to B. $SM : R =$ `instr` means that SM executes an instruction `instr` and the result is stored in R. No is a (non-secret) nonce.

First, the case that SP_i wants to attest a module SM_i that he deployed (this is the same procedure as described in Section 3.5):

$$SP_i \rightarrow SM_i : \{No\}$$

$$SM_i : mac(K_{N,SP_i,SM_i}, No) = \texttt{encrypt}\ \{\}, No$$

$$SM_i \rightarrow SP_i : \{mac(K_{N,SP_i,SM_i}, No)\}$$

Here, K_{N,SP_i,SM_i} is the module's default key so the MAC is produced by the `encrypt` instruction *without* the new *sp* argument.

Then, for SP_i to attest a module SM_j deployed by SP_j:

$$SP_i \rightarrow SM_j : \{No \mid SP_i\}$$

$$SM_j : mac(K_{N,SP_i,SM_j}, No \mid SP_j) = \texttt{encrypt}\ SP_i, \{\}, No \mid SP_j$$

$$SM_j \rightarrow SP_i : \{SP_j \mid mac(K_{N,SP_i,SM_j}, No \mid SP_j)\}$$

Note the use of the new form on the `encrypt` instruction here. If it is not important that SM_j was deployed by SP_j, this information can simply be left out of the MAC calculation and the last message.

9.1.5 Secure Linking Revisited

We have made the case above (Section 9.1.3) that when adding support for multi-vendor modules, it is important to be able to attest which provider deployed a module. This is also important when modules securely link to each other. As an example, say some module *SM* wants to use the services of a secure logging module SM_l. The latter module accepts data from local modules and sends it to SP_l to be securely stored in a database. Clearly, the identity of SP_l is important and it should be possible to attest it during the secure linking step.

Note that the above also holds for single-vendor modules; i.e., in Sancus' current design. However, Sancus currently provides no way to attest the provider a module was deployed by. Adding the provider's identity to the module's identity is, for the same reasons as above, probably not an appropriate solution. Therefore, we propose to add a new instruction to get the identity of the provider that deployed a module.

9.2 Public-Key Cryptography for Sancus

While designing Sancus in 2012, we dismissed the use of public-key cryptography because it would be too expensive to implement on the resource constrained devices that we wanted to target (Chapter 3). Since there has been a lot of development in low-cost public-key cryptography since then – Elliptic Curve Cryptography (ECC) in particular is promising [81] – it is worth to evaluate what the impact on Sancus would be if we were able to use it. We will start with describing a possible way to leverage public-key cryptography in Sancus en then describe its advantages.

9.2.1 Public-Key Protocol for Sancus

This section describes a possible way of using public-key cryptography to support Sancus' security features. In particular, the described protocol supports remote attestation and secure communication. Note that we do not claim that this is the only or best way to use public-key cryptography in Sancus; we merely want to demonstrate how it *could* be used.

We will use the same notation for cryptographic operations as in the rest of this text (Section 3.1) with the addition of the following. We write PK_i respectively PK_i^{-1} to denote the public and private parts of a key pair. To encrypt respectively sign some data D using the key PK_i we write $\textit{pk-encrypt}(PK_i, D)$ and $\textit{pk-sign}(PK_i^{-1}, D)$. For ease of notation, *pk-sign* outputs both the signature and the signed data.

In our public-key protocol, every node N is manufactured with a private key PK_N^{-1} in protected storage, much like the node key K_N in our current design. When an infrastructure provider *IP* buys a node, they are provided with the corresponding public key PK_N. *IP* will be the Certificate Authority (CA) for the public keys of the nodes they own by signing key certificates using PK_{IP}^{-1} and making these keys together with their certificates (publicly) available.

Whenever a software provider *SP* wants to deploy a module, they first ask *IP* for PK_{IP}.[1] They will then retrieve – and verify the certificate of – PK_N for the node N they want to deploy a module on. After compiling the module *SM* into a binary file B_{SM}, *SP* sends to following message to N:

$$SP \rightarrow N : \{No \mid PK_{SP} \mid B_{SM}\}$$

When the node receives this message, it will load *SM* in memory, enable memory protection, and generate a key pair (PK_{SM}, PK_{SM}^{-1}). The keys PK_{SP} and PK_{SM}^{-1} are stored in the protected storage area of *SM*. Then, N sends the following message to *SP*:

$$N \rightarrow SP : \{\textit{pk-sign}(PK_N^{-1}, No \mid PK_{SP} \mid PK_{SM} \mid SM)\}$$

where *SM* is the identity of the installed module and *No* is a nonce used for freshness. Note that this message can be used as a certificate to verify PK_{SM}. In addition, this certificate provides the same guarantees as the remote attestation procedure of Sancus: a module with identity *SM* has been correctly deployed on node N by software provider *SP* (identified by its public key PK_{SP}). It is also clear that since *SP* has access to the keys (PK_{SP}^{-1}, PK_{SM}) and *SM* to (PK_{SM}^{-1}, PK_{SP}), secure communication is easily accomplished:

$$SP \rightarrow SM : \{\textit{pk-sign}(PK_{SP}^{-1}, No \mid \textit{pk-encrypt}(PK_{SM}, I))\}$$

$$SM \rightarrow SP : \{\textit{pk-sign}(PK_{SM}^{-1}, No \mid \textit{pk-encrypt}(PK_{SP}, O))\}$$

where *No* is a nonce used for freshness, and I and O are input and output data, respectively. Of course, for efficiency reasons it might make sense to establish a symmetric session key after deploying a module and use that for further communication.

Before going into the advantages of this protocol, notice that we have basically defined a Public Key Infrastructure (PKI) for PMs. The root certificate binds PK_{IP} to *IP* and is assumed to be available to the software providers. The corresponding private key PK_{IP}^{-1} is used to create certificates binding node keys PK_N to nodes N. Private node keys PK_N^{-1} are in turn used to sign certificates binding module keys PK_{SM} to a module with identity *SM* deployed on node N by a software provider with public key PK_{SP}.

9.2.2 Advantages of a Public-Key Protocol

Easier Handling of Secrets at the Infrastructure Provider

Because an infrastructure provider acts as a CA for the nodes they own, software providers only need to securely receive PK_{IP} in order to be able to install

[1]Doing this securely is a difficult problem but out-of-scope.

modules on any node. Compare this with Sancus' current design, where a different symmetric key is needed for all nodes a software provider wants to install modules on. Using a public key protocol would thus mean much less interaction between IP and SP and this interaction only needs to be integrity protected.

For the same reason, the only secret stored by IP is PK_{IP}^{-1} and this secret only needs to be accessed when a new node is deployed to sign its public key. The symmetric key design of Sancus forces infrastructure providers to have access to a node's key whenever a new software provider registers for that node. Therefore, using public-key cryptography would mean that less access is needed to secrets which would make it more viable to air-gap this secret for added security.

Easier Inter-Provider Module Authentication

Imagine a software provider SP_i wanting to receive authenticated data from SM_j deployed by SP_j. Currently, SP_i would need to securely negotiate a shared secret with SP_j which SP_j then needs to send, using its secure communication channel, to SM_j. Using the described public-key protocol, however, SP_i can independently verify SM_j's certificate to start an authenticated communication session.

However, the authentication only works in one direction since SM_j has no way to authenticate SP_i. Using a TLS-like handshake SM_j could verify that all messages come from the same source but not that this source is SP_i. If two-way authentication is required, SP_i would need to ask SP_j to send PK_{SP_i} to SM_j using its secure communication channel. Therefore, the advantages of the public-key protocol would be lost.

Note that the same considerations hold for communication between two modules deployed by different providers: Inter-provider communication is only necessary if both modules need to authenticate each other.

9.3 Rigid Access Control Rules

Sancus' access control rules (Section 3.4) are rigid in two ways: (1) every module has exactly one data section; and (2) every data section belongs to maximum one module. We have already already encountered the inflexibility of (1) when designing our secure device drivers (Section 7.3.1). It is clear that (2) makes it difficult to share large amounts of data between modules. Although solutions have been proposed to make the latter easier for the user [87], they are not particularly efficient because the only way to share data between modules is by passing it through registers.

We will now discuss two ways to reduce this rigidity: one by lifting the restrictions of the current access control rules and another by decentralizing the rules through the use of memory capabilities.

9.3.1 A Many-to-Many Mapping Between Code and Data

Trustlite [51] solves both rigidities by allowing a many-to-many mapping between code- and data sections. Since every mapping can specify the kind of accesses that are allowed, this provides a memory access control scheme that is more generic, and hence flexible, than that of Sancus.

However, while Trustlite assumes a static set of access control rules, one of Sancus' goals is to support dynamic systems where modules are loaded and unloaded at runtime. Although there is no technical reason for the Trustlite model not to support dynamically changing the access control rules, from a security perspective there are some things that need to be thought through.

First note that, while overlapping access rules may exists, they should not exist for a newly created module. That is, when enabling a module, it should be ensured that no other modules on the node have access to the same memory regions. This is obviously necessary to support Sancus' reverse sandbox guarantees.

Another important consideration is how new access rules should be allowed to be defined. Since allowing any module to add any mapping would completely circumvent the memory access rules, a policy needs to be defined on how rules can be added to the system. A first observation is that new rules can always be added for memory regions that have no rules associated with them. This can then be used by modules to dynamically extend their data sections to, for example, incorporate MMIO regions. To support efficient data sharing, modules should also be allowed to give access to (a part of) their data regions to other modules. This means the system should have a policy on how to add access rules for memory regions that already have some other rules associated with them.

Clearly, for a module to be allowed to add a mapping for a memory region, it should at least have access to it itself. However, simply having access to a region is probably not enough for most use cases. Take, for example, a module SM_A that shares a private memory buffer with a module SM_B. If SM_B would be allowed to create new mappings for the buffer, it would be impossible for SM_A to fully reclaim the buffer. To solve this, a notion of memory ownership could be defined and only owners of a memory region should be allowed to create new mappings for this region. This could simply be implemented by adding an extra "owner" access bit to the hardware representation of access rules. Whenever a new mapping is created for an as of yet unmapped memory region, this owner bit should be set. If the owner later creates new mappings for (a part of) this region, the bit could either be automatically cleared or the owner could be given the option to keep it set.

A last important question to be answered is what exactly a module's identity is in a context where its memory regions can dynamically be changed. The easiest way is probably to define it in exactly the same way as Sancus currently defines it: the contents of its text section together with its layout information *at the time protection is enabled.* Although the text section and the layout information can now obviously change over time, using the ownership concept defined above, this can only be initiated by the module itself. This means that by attesting the initial state of a module, its provider will have the same guarantees of its secure execution as in Sancus' current design.

9.3.2 Sharing Data Using Capabilities

In capability-based addressing, access to a memory region is provided through an unforgeable reference. This reference is much like a classical pointer with additional information on the size of the referenced memory region and the way it may be accessed. Since it is unforgeable, the memory access logic can be completely decentralized: access to a memory region is allowed if it is done through a proper capability.

Using capability-based addressing, sharing a memory region is as simple as passing a capability for that region around. Therefore, this could provide a solution to the efficient memory sharing problem in the current Sancus design. Moreover, adding data sections to a module might be implemented by giving the module access to additional memory regions by providing it with capabilities. Thus, capabilities might provide a solution to both access control rigidities addressed in this section.

Unfortunately, things are not quite so easy. For example, as mentioned above, we may want modules to be able to reclaim full control over memory regions they shared in the past. This would boil down to the ability to revoke shared capabilities, a notoriously difficult problem. Another problem that we glossed over above is that, although it seems easy to define a module's data sections through the capabilities it owns, we want to ensure that no other capabilities exist for those regions to support Sancus' reverse sandbox guarantees. Section 9.4.3 discusses these problems in some more detail.

9.4 Hardware Limit on the Number of Modules

As explained in Section 4.1.1, there is an upper limit on the number of modules that can be concurrently loaded. This limit, N_{SM}, has to be chosen when synthesizing the processor. From a security engineer's perspective, N_{SM} should be as large as possible since it allows for more fine-grained compartmentalization of applications. From a hardware vendor's perspective, however, it should not be too high since it has an adverse effect on the hardware cost and the maximum attainable frequency (Figures 5.1 and 5.4).

Of course, there will always be some limit on the number of concurrent modules. At the very least, it will be limited by the amount of available memory. Thus, if we talk about removing the hardware limit on N_{SM} in this section, we mean a limit smaller than the one imposed by the available memory.

9.4.1 Moving Metadata to Memory

The per-module metadata includes the module's layout, its key and its identifier (Figure 3.1). Since all this information is necessary to support Sancus' features, we need to find a way to represent it all *without* requiring hardware registers to overcome the limit on N_{SM}. One obvious idea might be to move all this information to memory.

Applying this idea to just the module keys might already alleviate most of the hardware cost issues since they constitute the largest part of a module's metadata for all reasonable security levels. The hardware could reserve a fixed part of a module's data section to store the key. To keep the same security guarantees as Sancus currently has, the hardware should ensure that this part of the data section is not directly accessible by the module itself, only through the cryptographic instructions. An important observation is that storing keys in memory would not incur any runtime overhead on Sancus due to the way key bytes are accessed by its crypto core.

The layout information, on the other hand, seems more difficult to move to memory. Without having to define the exact format of how this metadata would be represented in memory, it should be clear that it is undesirable to have to read memory in order to decide whether a memory access is allowed. Indeed, since at least one decision has to be made for every instruction – whether executing that instruction is allowed – this would incur an unacceptable overhead. This problem is amplified by the fact that the memory access logic needs *global* information; that is, it needs to know about *all* loaded modules in order to make a decision. Although a clever caching scheme might be able to make this much more efficient than just reading all this information from memory when it is needed, one of the selling points of Sancus is that it incurs *zero* overhead on application not using our extensions. Clearly, only a cache big enough to store the layout information of all loaded modules would be able to guarantee that kind of performance. This is exactly how Sancus is currently implemented.

9.4.2 Tagging Memory with Ownership Information

The main scalability problem with the current memory access logic seems to be that for every memory access decision, the layout information of *all* loaded modules needs to be consulted. Doing this efficiently, as pointed out above, is the reason Sancus currently has a hardware-imposed limit on N_{SM}. A solution to this problem might exists in finding a way to *decentralize* this information such that the memory access logic only needs to consult a small amount of local information.

The obvious way to decentralize layout information is to tag every memory location with the identifier of the module that owns it and the location's type (i.e., text, data or entry point). This would also allow modules to have a variable number of sections solving one of the rigid access control problems discussed earlier (Section 9.3). However, for the same performance reasons as above, this tag cannot reside in main memory because that would mean that every memory access requires one extra to read the tag.

It is clear that the overhead of memory tagging grows linearly with the amount of memory. How big this overhead is exactly, depends on a number of factors. First, the size of the tag is the size of a module identifier plus the number of bits for the memory location's type. Since, as described above, Sancus currently defines three types, two bits are sufficient to represent it. For the module identifiers, recall that they should never be reused within a boot cycle (Section 3.7). This means that the size of this identifier within a tag is not just a limit of the number of concurrently loaded modules but on the total number of modules loaded within a boot cycle. Although this number may be lowered to reduce the overhead, 14 bits for the identifier should be ample for most use cases, making it an even 16 bits in total for the tag.

The second factor in the total overhead is the size of the memory location covered by a tag. For the finest granularity, we would require one tag per byte for a total tag space that is twice the size of the memory space. This overhead can be reduced by increasing the size of memory locations but this will in turn increase the effects of memory fragmentation and restrict the number of modules that can be concurrently loaded. Thus, supporting a larger number of concurrently loaded modules means increasing the tag space, making this a non-solution to our problem.

We might therefore think that this memory tagging scheme is no better that simply synthesizing the current Sancus implementation with a large value for N_{SM}. However, besides the previously mentioned advantage of modules supporting multiple sections, there is another important difference that stems from the decentralization of layout information: The memory access logic no longer needs access to all layout information to decide whether a memory access is allowed. Indeed, only the tags of the executing instruction and the accessed memory location need to be consulted. This means we will only need a single MAL circuit (Figure 4.1, although the circuit will obviously be different) instead of one per supported module, potentially decreasing the processor's critical path substantially.

9.4.3 Reverse Sandboxing Using Capabilities

Since capabilities offer another way to decentralize access control, it might provide a solution to lift the hardware limit on the number of concurrent modules. This section continues the discussion on using capabilities for data sharing (Section 9.3.2) by expanding on some of the problems mentioned there.

If we want to employ capability-based addressing to implement Sancus' security guarantees, we should first define what a protected module is in this context. In classical capability systems, a protection domain is usually defined as all memory locations that are accessible through all reachable capabilities. There is a set of implicit capabilities, usually in the processor's register file, that are always accessible. Using these capabilities, the application is able to address memory locations and these locations may contain more capabilities. The current protection domain, then, is the set of all memory locations reachable from the implicit capability set.

Such a protection domain, however, does not define a protected module. Indeed, a protection domain defines to which memory locations executing code has access while a protected module defines to which memory locations outside code does not have access. In other words, a protection domain defines a sandbox while a protected module defines a reverse sandbox. It seems, therefore, that the goals of capabilities are the complete opposite of those of Sancus.

While this is true, note that Sancus accomplishes a reverse sandbox by having hardware logic to ensure that when a new module is loaded, it does not overlap with other modules. This hardware component is able to do its job by having a global view of all loaded modules. If we would be able to define a similar entity in a capability system – one that has a view of all existing capabilities – we are able to create reverse sandboxes. Indeed, this would allow us to load a module and then verify that no other capabilities exist for the memory regions allocated to that module. The simplest way to implement this would be to scan the whole address space, something that may be feasible for the small microcontrollers that Sancus targets (e.g., its prototype implementation has a 16-bit address space).

Before explaining how modules can be defined, we have to introduce one more concept from capability systems: *entry capabilities* [59, Chapter 4]. Although the details may differ between implementations, the basic idea of an entry capability is the following. Like any capability, an entry capability defines a region of memory. This region, however, is not accessible by a holder of the capability in any other way than by *calling* the capability. When such a capability is called, it is transformed into a read-execute capability and control is transfered to the first instruction in the memory region. This allows the called code to gain access to private memory regions by including capabilities in the memory region pointed to by the entry capability.

We will now show *one possible way* to create and attest reverse sandboxed modules using capabilities. An untrusted loader will obtain two capabilities from an untrusted memory manager: one to store the module's text section and one for the data section. The loader then loads the module's text section and appends the data capability to it. At this point, the `protect` instruction is called with the text capability as argument which performs the following steps: (1) verify that there is no other capability referring to the text section; (2) verify that the last location of the text section contains a capability and no other capabilities exist that refer to the same memory region; and (3) transform the text capability into an entry capability. For attestation purposes, this instruction could calculate a module key in exactly the same way as current Sancus does and store it at a fixed location in the module's data section (as discussed in Section 9.4.1).

This procedure ensures the following: (1) the only capability for the module's data section is located at the last location of the module's text section; and (2) the only capability for the module's text section is transformed into an entry capability. Therefore, the only way to execute a module is by calling its entry capability which guarantees the module can only be entered through its first instruction. Moreover, accessing the module's data capability is also only possible by executing its entry capability which means only the module's code is initially able to access its data section. Of course, the code may later decide to hand out capabilities to (parts of) its data section to other modules.

Chapter 10

Conclusion

The increased connectivity and extensibility of networked embedded devices in the IoT leads to exciting new applications, but also to significant new security threats. Recent incidents have shown the need for research on security in this context. This thesis proposed a novel security architecture called Sancus, that is low-cost yet provides strong security guarantees with a small, hardware-only, TCB.

Sancus' core feature is to provide memory isolation to software modules through PCBAC which ensures that (1) a module's data is only accessible when executing its code; and (2) its code can only be executed by jumping to a well-defined entry point. By implementing these access control checks in hardware, Sancus is able to enforce memory isolation without imposing an overhead, in terms of cycles, on the existing instruction set of the microprocessor. Although Sancus causes a small runtime overhead when entering or exiting modules to ensure a secure switching of protection domains, this is a minor price to pay for the security guarantees provided by PCBAC.

Isolation in itself, however, is not enough for a security architecture to be applicable to the shared infrastructures that Sancus aimed to support. Indeed, there is no point in deploying and isolating a module if we do not also have way of attesting the state of the module and to securely communicate with it. Therefore, Sancus adds a cryptographic core supporting authenticated encryption to the microprocessor's hardware. By using a key hierarchy that allows both the hardware and a module's deployer to derive the same key, Sancus is able to establish a confidential and integrity protected communication channel between a module and its deployer. Moreover, by using a key derivation scheme that generates keys based on, among others, a module's code section, this same channel also provides authenticity guarantees which means it implements remote attestation.

Sancus also supports secure communication between modules running on the same node. Secure linking, as it is called in Sancus, is implemented by allowing modules to identify other modules loaded at a certain memory location through a cryptographic hash of, among others, the module's code section. After this attestation procedure, a module can securely call another by jumping to its entry point and passing confidential data through registers. Sancus keeps track of the module that called an entry point to allow the called module to attest its caller. Although this procedure provides mutual authentication between modules, it is potentially expensive when large amounts of data need to be passed between them. We proposed some possible solutions to this problem in the discussion section.

Beside the core architecture, this thesis also discussed two applications of Sancus. First, we provided a way to employ Sancus in a distributed context. More specifically, we described a deployment strategy that allows us to gain trust of the outputs produced by distributed applications. That is, whenever an output is produced by such an application, then there must have happened a sequence of physical input events such that that sequence, when processed by the application as specified in the high-level source code, produces that output event. In other words, when a distributed application is deployed using our techniques and it produces an output, it could also have produced that output if no attacker was present.

An important contribution of this first application is the development of techniques to perform secure I/O from Sancus modules. We have shown how to write secure device drivers that allow us to authenticate physical devices. Future work might use these techniques to implement secure peripherals for desktop or server platforms.

In a second application, trust assessment modules, we have shown how to use a protected Sancus module to attest the state of unprotected software running on the same node. Such a module uses several heuristics – such as code integrity, available resources, or the timing of events – to gauge the health of the node and, using Sancus' secure communication features, reports this state back to a trust management system. This provides a way to add security guarantees to an otherwise unchanged legacy software stack, making it a useful tool when porting the software itself to Sancus is infeasible.

By concluding this thesis with a discussion on a number of design alternatives for Sancus, we have shown that there is no single *right* answer in security. Instead, there is a constant trade-off between security and efficiency. Nevertheless, this thesis has shown that not too many compromises need to be made since Sancus provides strong security guarantees at a moderate cost.

Bibliography

[1] Martín Abadi. “Protection in Programming-Language Translations”. In: *Secure Internet Programming, Security Issues for Mobile and Distributed Objects.* 1999, pp. 19–34.

[2] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. “Secure Compilation to Modern Processors”. In: *2012 IEEE 25th Computer Security Foundations Symposium (CSF 2012).* Los Alamitos, CA, USA: IEEE Computer Society, 2012, pp. 171–185. ISBN: 978-1-4673-1918-8.

[3] Tiago Alves and Don Felton. “TrustZone: Integrated hardware and software security”. In: *ARM white paper* 3.4 (2004), pp. 18–24.

[4] A. A. d. Amorim, M. Dénès, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. “Micro-Policies: Formally Verified, Tag-Based Security Monitors”. In: *2015 IEEE Symposium on Security and Privacy.* May 2015, pp. 813–830.

[5] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. *Innovative Technology for CPU Based Attestation and Sealing.* 2013.

[6] Ross J. Anderson and Markus G. Kuhn. “Low Cost Attacks on Tamper Resistant Devices”. In: *Proceedings of the 5th International Workshop on Security Protocols.* London, UK, UK: Springer-Verlag, 1998, pp. 125–136. ISBN: 3-540-64040-1.

[7] Elena Andreeva, Begül Bilgin, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha, and Kan Yasuda. “APE: Authenticated Permutation-Based Encryption for Lightweight Cryptography”. In: *Fast Software Encryption: 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers.* Ed. by Carlos Cid and Christian Rechberger. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 168–186.

[8] Elena Andreeva, Joan Daemen, Bart Mennink, and Gilles Van Assche. "Security of Keyed Sponge Constructions Using a Modular Proof Approach". In: *Fast Software Encryption: 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers.* Ed. by Gregor Leander. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 364–384.

[9] A.M. Azab, P. Ning, and X. Zhang. "SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms". In: *Proceedings of the 18th ACM conference on Computer and communications security.* ACM. 2011, pp. 375–388.

[10] A. Baliga, V. Ganapathy, and L. Iftode. "Detecting Kernel-Level Rootkits Using Data Structure Invariants". In: *Dependable and Secure Computing, IEEE Transactions on* 8.5 (2011), pp. 670–684.

[11] Richard Barry. *FreeRTOS: A portable, open source, mini Real Time kernel.* `http://www.freertos.org/`. 2010.

[12] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Duplexing the Sponge: Single-pass Authenticated Encryption and Other Applications". In: *Selected Areas in Cryptography.* Springer. 2011, pp. 320–337.

[13] Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. "SPONGENT: The Design Space of Lightweight Cryptographic Hashing". In: vol. 99. PrePrints. Los Alamitos, CA, USA: IEEE Computer Society, 2012, p. 1.

[14] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Eliminating Errors in Cryptographic Computations". In: *J. Cryptology* 14 (2001), pp. 101–119.

[15] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. "TyTAN: Tiny Trust Anchor for Tiny Devices". In: *Proceedings of the 52Nd Annual Design Automation Conference.* DAC '15. San Francisco, California: ACM, 2015, 34:1–34:6. ISBN: 978-1-4503-3520-1.

[16] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. "Hardware Support for Fast Capability-based Addressing". In: *SIGPLAN Not.* 29.11 (Nov. 1994), pp. 319–327.

[17] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. "On the difficulty of software-based attestation of embedded devices". In: *Proceedings of the 16th ACM conference on Computer and communications security.* CCS '09. Chicago, Illinois, USA: ACM, 2009, pp. 400–409. ISBN: 978-1-60558-894-0.

[18] R. de Clercq, F. Piessens, D. Schellekens, and I. Verbauwhede. "Secure interrupts on low-end microcontrollers". In: *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on.* IEEE, 2014, pp. 147–152.

[19] Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr. "Efficient memory safety for TinyOS". In: *Proceedings of the 5th international conference on Embedded networked sensor systems.* SenSys '07. Sydney, Australia: ACM, 2007, pp. 205–218. ISBN: 978-1-59593-763-6.

[20] FJ Corbato and VA Vyssotsky. "Introduction and overview of the Multics system". In: *Proceedings of the November 30–December 1, 1965, Fall joint computer conference, part I.* ACM. 1965, pp. 185–196.

[21] Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. "A Survey of Software Aging and Rejuvenation Studies". In: *J. Emerg. Technol. Comput. Syst.* 10.1 (2014), 8:1–8:34.

[22] Jack B. Dennis and Earl C. Van Horn. "Programming Semantics for Multiprogrammed Computations". In: *Commun. ACM* 9.3 (Mar. 1966), pp. 143–155.

[23] Dominique Devriese and Frank Piessens. "Noninterference Through Secure Multi-Execution". In: *Proceedings of the IEEE Symposium on Security and Privacy.* 2010, pp. 109–124.

[24] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight Jr., Benjamin C. Pierce, and Andre DeHon. "Architectural Support for Software-Defined Metadata Processing". In: *SIGARCH Comput. Archit. News* 43.1 (Mar. 2015), pp. 487–502.

[25] Danny Dolev and Andrew C. Yao. "On the Security of Public Key Protocols". In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208.

[26] Saar Drimer. *Security for volatile FPGAs.* UCAM-CL-TR-763. University of Cambridge, Computer Laboratory, Nov. 2009. URL: `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-763.pdf`.

[27] A. Dunkels, B. Gronvall, and T. Voigt. "Contiki - a lightweight and flexible operating system for tiny networked sensors". In: *Local Computer Networks, 2004. 29th Annual IEEE International Conference on.* `http://www.contiki-os.org/`. 2004, pp. 455–462.

[28] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. "Run-time dynamic linking for reprogramming wireless sensor networks". In: *Proceedings of the 4th international conference on Embedded networked sensor systems.* SenSys '06. Boulder, Colorado, USA: ACM, 2006, pp. 15–28. ISBN: 1-59593-343-3.

[29] Karim Eldefrawy, Aurélien Francillon, Daniele Perito, and Gene Tsudik. "SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust". In: *NDSS 2012, 19th Annual Network and Distributed System Security Symposium, February 5-8, San Diego, USA*. San Diego, UNITED STATES, Feb. 2012.

[30] Úlfar Erlingsson, Yves Younan, and Frank Piessens. "Low-Level Software Security by Example". In: *Handbook of Information and Communication Security*. Springer, 2010.

[31] R. S. Fabry. "Capability-based Addressing". In: *Commun. ACM* 17.7 (July 1974), pp. 403–412.

[32] Muhammad Omer Farooq and Thomas Kunz. "Operating Systems for Wireless Sensor Networks: A Survey". In: *Sensors* 11.6 (2011), pp. 5900–5930. ISSN: 1424-8220.

[33] H.H. Feng, O.M. Kolesnikov, P. Fogla, Wenke Lee, and W. Gong. "Anomaly detection using call stack information". In: *Security and Privacy, 2003. Proceedings. 2003 Symposium on.* USENIX Association, 2003, pp. 62–75.

[34] M.C. Fernandez-Gago, R. Roman, and J. Lopez. "A Survey on the Applicability of Trust Management Systems for Wireless Sensor Networks". In: *Security, Privacy and Trust in Pervasive and Ubiquitous Computing, 2007. SECPerU 2007. Third International Workshop on.* 2007, pp. 25–30.

[35] Aurélien Francillon and Claude Castelluccia. "Code injection attacks on Harvard-architecture devices". In: *Proceedings of the 15th ACM conference on Computer and communications security.* CCS '08. Alexandria, Virginia, USA: ACM, 2008, pp. 15–26. ISBN: 978-1-59593-810-7.

[36] Francesco Gadaleta, Nick Nikiforakis, JanTobias Mühlberg, and Wouter Joosen. "HyperForce: Hypervisor-enForced Execution of Security-Critical Code". In: *Information Security and Privacy Research.* Vol. 376. IFIP Advances in Information and Communication Technology. Springer, 2012, pp. 126–137.

[37] Francesco Gadaleta, Nick Nikiforakis, Yves Younan, and Wouter Joosen. "Hello rootKitty: A Lightweight Invariance-Enforcing Framework". In: *Information Security.* Vol. 7001. LNCS. Springer, 2011, pp. 213–228.

[38] Gartner. *Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020.* `http://www.gartner.com/newsroom/id/2636073`. 2013.

[39] Thanassis Giannetsos, Tassos Dimitriou, and Neeli R. Prasad. "Self-propagating worms in wireless sensor networks". In: *Proceedings of the 5th international student workshop on Emerging networking experiments and technologies.* Co-Next Student Workshop '09. Rome, Italy: ACM, 2009, pp. 31–32. ISBN: 978-1-60558-751-6.

[40] Olivier Girard. *openMSP430*. `http://opencores.org/project,openmsp430`. 2016.

[41] *GlobalPlatform*. `http://www.globalplatform.org/`. 2016.

[42] Johannes Götzfried, Tilo Müller, Ruan de Clercq, Pieter Maene, Felix Freiling, and Ingrid Verbauwhede. "Soteria: Offline Software Protection within Low-cost Embedded Devices". In: *Proceedings of the 31st Annual Computer Security Applications Conference*. 2015.

[43] Jorge Granjal, Edmundo Monteiro, and Jorge Sá Silva. "Security in the integration of low-power Wireless Sensor Networks with the Internet: A survey". In: *Ad Hoc Networks* 24, Part A (2015), pp. 264–287.

[44] Lin Gu and John A. Stankovic. "t-kernel: providing reliable OS support to wireless sensor networks". In: *Proceedings of the 4th international conference on Embedded networked sensor systems*. Boulder, Colorado, USA: ACM, 2006, pp. 1–14. ISBN: 1-59593-343-3.

[45] D. Halperin, T.S. Heydt-Benjamin, B. Ransford, S.S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W.H. Maisel. "Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses". In: *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. Ieee. 2008, pp. 129–142.

[46] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. "A dynamic operating system for sensor nodes". In: *Proceedings of the 3rd international conference on Mobile systems, applications, and services*. MobiSys '05. Seattle, Washington: ACM, 2005, pp. 163–176. ISBN: 1-931971-31-5.

[47] Ralf Hund, Carsten Willems, and Thorsten Holz. "2013 IEEE Symposium on Security and Privacy Practical Timing Side Channel Attacks Against Kernel Space ASLR". In: 2013.

[48] Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. "Towards a Fully Abstract Compiler Using Micro-Policies: Secure Compilation for Mutually Distrustful Components". In: *CoRR* abs/1510.00697 (2015).

[49] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '96. London, UK, UK: Springer-Verlag, 1996, pp. 104–113. ISBN: 3-540-61512-1.

[50] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '99. London, UK, UK: Springer-Verlag, 1999, pp. 388–397. ISBN: 3-540-66347-9.

[51] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. "TrustLite: A Security Architecture for Tiny Embedded Devices". In: *Proceedings of the Ninth European Conference on Computer Systems.* EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 10:1–10:14. ISBN: 978-1-4503-2704-6.

[52] Kari Kostiainen, Jan-Erik Ekberg, N. Asokan, and Aarne Rantala. "Onboard credentials with open provisioning". In: *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security.* ASIACCS '09. Sydney, Australia: ACM, 2009, pp. 104–115. ISBN: 978-1-60558-394-5.

[53] Ram Kumar, Eddie Kohler, and Mani Srivastava. "Harbor: software-based memory protection for sensor nodes". In: *Proceedings of the 6th international conference on Information processing in sensor networks.* IPSN '07. Cambridge, Massachusetts, USA: ACM, 2007, pp. 340–349. ISBN: 978-1-59593-638-7.

[54] Fangfei Liu;Yuval Yarom;Qian Ge;Gernot Heiser;Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical". In: San Jose, May 2015, pp. 605–622.

[55] Yong Ki Lee, K. Sakiyama, L. Batina, and I. Verbauwhede. "Elliptic-Curve-Based Security Processor for RFID". In: *Computers, IEEE Transactions on* 57.11 (Nov. 2008), pp. 1514–1527. ISSN: 0018-9340.

[56] Ilias Leontiadis, Christos Efstratiou, Cecilia Mascolo, and Jon Crowcroft. "SenShare: transforming sensor networks into multi-application sensing infrastructures". In: *Proceedings of the 9th European conference on Wireless Sensor Networks.* EWSN'12. Trento, Italy: Springer-Verlag, 2012, pp. 65–81. ISBN: 978-3-642-28168-6.

[57] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. "TinyOS: An Operating System for Sensor Networks". In: *Ambient Intelligence.* Springer, 2005, pp. 115–148.

[58] Philip Levis. "Experiences from a decade of TinyOS development". In: *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation.* OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 207–220. ISBN: 978-1-931971-96-6.

[59] Henry M. Levy. *Capability-Based Computer Systems.* Newton, MA, USA: Butterworth-Heinemann, 1984. ISBN: 0932376223.

[60] Jay Ligatti, Lujo Bauer, and David Walker. "Edit automata: enforcement mechanisms for run-time security policies". English. In: *International Journal of Information Security* 4.1-2 (2005), pp. 2–16. ISSN: 1615-5262.

[61] LLVM Developer Group. *Clang.* `http://clang.llvm.org/`. 2016.

[62] LLVM Developer Group. *LLVM.* `http://llvm.org/`. 2016.

[63] Javier Lopez, Rodrigo Roman, Isaac Agudo, and Carmen Fernandez-Gago. “Trust Management Systems for Wireless Sensor Networks: Best Practices”. In: *Comput. Commun.* 33.9 (2010), pp. 1086–1093.

[64] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. “Flask: Staged Functional Programming for Sensor Networks”. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming.* ICFP ’08. 2008.

[65] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. “TrustVisor: Efficient TCB Reduction and Attestation”. In: *Proceedings of the IEEE Symposium on Security and Privacy.* May 2010.

[66] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. “Flicker: An Execution Infrastructure for TCB Minimization”. In: *Proceedings of the ACM European Conference in Computer Systems (EuroSys).* ACM. Apr. 2008, pp. 315–328.

[67] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. “Innovative Instructions and Software Model for Isolated Execution”. In: *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy.* HASP ’13. Tel-Aviv, Israel: ACM, 2013, 10:1–10:1. ISBN: 978-1-4503-2118-1.

[68] Jan Tobias Mühlberg, Sara Cleemput, Mustafa A. Mustafa, Jo Van Bulck, Bart Preneel, and Frank Piessens. “An Implementation of a High Assurance Smart Meter using Protected Module Architectures”. In: *WISTP ’16.* LNCS. To appear. Heidelberg: Springer, 2016.

[69] Jan Tobias Mühlberg, Job Noorman, and Frank Piessens. “Lightweight and Flexible Trust Assessment Modules for the Internet of Things”. In: *ESORICS ’15.* Vol. 9326. LNCS. Heidelberg: Springer, 2015, pp. 503–520.

[70] Nick Nikiforakis, Frank Piessens, and Wouter Joosen. “HeapSentry: Kernel-Assisted Protection against Heap Overflows”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment.* Vol. 7967. LNCS. Springer, 2013, pp. 177–196.

[71] Job Noorman. *Sancus Supplementary Materials.* URL: `https://distrinet.cs.kuleuven.be/software/sancus/`.

[72] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base". In: *22nd USENIX Security symposium.* USENIX Association, Aug. 2013, pp. 479–494.

[73] Job Noorman, Nick Nikiforakis, and Frank Piessens. "There is safety in numbers: Preventing control-flow hijacking by duplication". In: *17th Nordic Conference on Secure IT Systems (NordSec 2012).* 2012.

[74] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. "Secure Compilation to Protected Module Architectures". In: *ACM Trans. Program. Lang. Syst.* 37.2 (Apr. 2015), 6:1–6:50. ISSN: 0164-0925.

[75] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. "Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor." In: *USENIX Security Symposium.* USENIX Association, 2004, pp. 179–194.

[76] Ryan Riley, Xuxian Jiang, and Dongyan Xu. "Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing". In: *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection.* Vol. 5230. LNCS. Springer, 2008, pp. 48–67.

[77] Rodrigo Roman, Pablo Najera, and Javier Lopez. "Securing the Internet of Things". In: *Computer* 44.9 (2011), pp. 51–58.

[78] Dewan P. Sahita R Warrier U. "Protecting Critical Applications on Mobile Platforms". In: *Intel Technology Journal* 13 (2 2009), pp. 16–35.

[79] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. "SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes". In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles.* ACM, 2007, pp. 335–350.

[80] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. "Java™ on the bare metal of wireless sensor devices: the squawk Java virtual machine." In: *VEE.* Ed. by Hans-Juergen Boehm and David Grove. ACM, Dec. 15, 2006, pp. 78–88.

[81] Sujoy Sinha Roy, Kimmo Järvinen, and Ingrid Verbauwhede. "Lightweight Coprocessor for Koblitz Curves: 283-Bit ECC Including Scalar Conversion with only 4300 Gates". In: *Cryptographic Hardware and Embedded Systems – CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings.* Ed. by Tim Güneysu and Helena Handschuh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 102–122.

[82] Raoul Strackx, Job Noorman, Ingrid Verbauwhede, Bart Preneel, and Frank Piessens. "Protected software module architectures". In: *ISSE 2013 Securing Electronic Business Processes.* Springer, 2013, pp. 241–251. URL: https://lirias.kuleuven.be/handle/123456789/430320.

[83] Raoul Strackx and Frank Piessens. "Fides: Selectively hardening software application components against kernel-level or process-level malware". In: *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS 2012),* ACM Press, Oct. 2012, pp. 2–13.

[84] Raoul Strackx, Frank Piessens, and Bart Preneel. "Efficient isolation of trusted subsystems in embedded systems". In: *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering: Security and Privacy in Communication Networks.* Vol. 50. Springer, Sept. 2010, pp. 1–18.

[85] The PaX Team. *PaX.* URL: https://pax.grsecurity.net/.

[86] Jo Van Bulck, Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. "Towards Availability and Real-Time Guarantees for Protected Module Architectures". In: *MASS '16, MODULARITY Companion Proceedings '16.* New York: ACM, 2016, pp. 146–151.

[87] Jo Van Bulck, Job Noorman, Tobias Mühlberg, and Frank Piessens. "Secure resource sharing for embedded protected module architectures". In: *Lecture Notes in Computer Science.* Vol. 9311. Springer, 2015, pp. 71–87.

[88] J. Viega and H. Thompson. "The State of Embedded-Device Security (Spoiler Alert: It's Bad)". In: *Security Privacy, IEEE* 10.5 (2012), pp. 68–70. ISSN: 1540-7993.

[89] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. "The CHERI Capability Model: Revisiting RISC in an Age of Risk". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture.* ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 457–468.

[90] Yves Younan, Wouter Joosen, and Frank Piessens. "Runtime countermeasures for code injection attacks against C and C++ programs". In: *ACM Comput. Surv.* 44.3 (June 2012), 17:1–17:28. ISSN: 0360-0300.

[91] Marcus Peinado Yuanzhong Xu Weidong Cui. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland).* IEEE – Institute of Electrical and Electronics Engineers, May 2015.

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